## Document Information

| | |
|---|---|
| **Analyzed document** | C Progamming - Complete Ebook.pdf (D165877814) |
| **Submitted** | 5/4/2023 1:04:00 PM |
| **Submitted by** | Mumtaz B |
| **Submitter email** | mumtaz@code.dbuniversity.ac.in |
| **Similarity** | 10% |
| **Analysis address** | mumtaz.dbuni@analysis.urkund.com |

## Sources included in the report

| | | |
|---|---|---|
| W | URL: http://referenceglobe.com/kpsslp/support/upload_videos/programming%20for%20problem%20solving_1... Fetched: 2/23/2023 8:25:28 AM | 4 |
| SA | **BCSDSC_1-BASIC CONCEPTS OF COMPUTER & C PROGRAMMING_plagarisum check-FINAL.pdf** Document BCSDSC_1-BASIC CONCEPTS OF COMPUTER & C PROGRAMMING_plagarisum check-FINAL.pdf (D139109702) | 1 |
| W | URL: https://www.iare.ac.in/sites/default/files/AERO_PROGRAMMING_FOR_PROBLEM_SOLVING_LECTURE_NOTES.pdf Fetched: 12/9/2022 8:21:24 AM | 2 |
| W | URL: https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming... Fetched: 3/31/2022 4:41:31 AM | 60 |
| W | URL: https://cpp4rstarters.wordpress.com/2012/01/05/4/ Fetched: 12/12/2021 6:03:59 PM | 1 |
| SA | **C book final copy (1).doc** Document C book final copy (1).doc (D31388791) | 5 |
| SA | **C_book_AmrutaJog_ShrutiJathar.pdf** Document C_book_AmrutaJog_ShrutiJathar.pdf (D54111295) | 4 |
| SA | **Block 3 and 4.pdf** Document Block 3 and 4.pdf (D129191893) | 1 |
| SA | **Dr. Baldev SIngh LKC_Programming in C.pdf** Document Dr. Baldev SIngh LKC_Programming in C.pdf (D137918641) | 1 |
| SA | **BCADSC-1-FUNDAMENTALS OF COMPUTERS AND PROGRAMMING IN C-Block-1-4 and 9-16-2-266.pdf** Document BCADSC-1-FUNDAMENTALS OF COMPUTERS AND PROGRAMMING IN C-Block-1-4 and 9-16-2-266.pdf (D138636232) | 6 |
| SA | **Programming in C Block 2.pdf** Document Programming in C Block 2.pdf (D164968573) | 1 |
| SA | **The C Programming Language.pdf** Document The C Programming Language.pdf (D44958811) | 32 |
| W | URL: https://nandhini0205.files.wordpress.com/2018/03/c-programming-2.pdf Fetched: 12/9/2021 6:46:12 PM | 4 |

SA **Sem I_BCA_B21CA02DC.docx**
Document Sem I_BCA_B21CA02DC.docx (D165443993) 1

SA **C_book_FINAL(July 2019).pdf**
Document C_book_FINAL(July 2019).pdf (D54399424) 2

SA **002 CHAPTERS.docx**
Document 002 CHAPTERS.docx (D19089854) 1

## Entire Document

Self-Instructional Material 3 Introduction to Programming NOTES UNIT 1 INTRODUCTION TO PROGRAMMING Structure 1.0 Introduction 1.1 Unit Objectives 1.2 Basic Model of Computation 1.2.1 Definition of a Problem 1.2.2 Designing of Solution to the Problem 1.3 Algorithm 1.3.1 History of Algorithms 1.3.2 Documenting Algorithms 1.3.3 Pseudo Code 1.3.4 Testing Algorithm 1.3.5 Some Simple Rules governing Algorithms 1.3.6 Divide and Conquer 1.4 Flowcharts 1.4.1 Graphical Symbols used in Flowcharts 1.4.2 Basic Control Structures 1.4.3 Flowchart vs Pseudo Code 1.5 Structured Programming Concept 1.5.1 Features of Structured Programming 1.5.2 Advantages of Structured Programming 1.6 Programming Environment 1.7 Software Classification 1.8 Programming Languages 1.8.1 Machine Language 1.8.2 Assembly Language 1.8.3 High-Level Languages 1.8.4 Fourth-Generation Languages (4GLs) 1.8.5 Fifth-Generation Languages (5GLs) 1.8.6 Assemblers 1.8.7 Compilers 1.8.8 Interpreters 1.9 Program Writing and Execution 1.9.1 Source Code 1.9.2 Object Code 1.9.3 Linking and Loading 1.10 Summary 1.11 Key Terms 1.12 Answers to 'Check Your Progress' 1.13 Questions and Exercises 1.14 Further Reading 1.0 INTRODUCTION Programming has become synonymous with computers. Millions of programmers are engaged in developing programs all over the world and billions of lines of program code
4 Self-Instructional Material Introduction to Programming NOTES have been developed and delivered to be used by a wide cross-section of society. Computer programs, also known as software systems, have helped in breaking many barriers. Today, an aircraft may not even take-off without its corresponding software system functioning properly. Satellite and space shuttle technology, weather forecasting, oil exploration, e-commerce, e-governance, Internet, Intranet, e-mail, etc. have been made easy by successful computer programs. Computer software has helped mankind to enhance productivity, efficiency and, above all, quality of life. There are, however, darker sides to programming as well; for example, the case of failure of the Apollo 13 mission due to programming errors, loss of US $ 100 million in one year to an American airlines company due to defect in software for determining excess flight bookings, etc. Such hardships caused by programming errors are too many! Many software development projects go haywire due to poor understanding of customer requirements, programming concepts and poor workmanship. Therefore, a correct understanding of programming methodology is essential for every programmer. A programmer should keep programs short, simple and elegant. Program coding should be carried out in a systematic manner without resorting to short cuts and no programming should be done in a hurry. If there were not enough time, it would be better to postpone program development rather than merely typing in a few program statements. A better quality of program has to be planned and achieved. Therefore, program development is not merely learning the syntax of a programming language or formulating a group of statements, but creating programs that will function accurately forever. 1.1 UNIT
OBJECTIVES
After going through
this unit, you will be able to: • Understand the basic model of
computation • Describe algorithm • Explain the concept of
structured programming •
Define programming environment • Classify software • Comprehend various programming languages • Write simple programs 1.2 BASIC MODEL OF COMPUTATION The easy availability of computers and telecommunication facilities has improved the quality of life dramatically in the twenty-first century. What is a computer? It is a man- made device that is now available off-the-shelf. A computer system has two parts: hardware and software. The former has a physical form, whereas the latter is intangible. Computer hardware can be made to perform a wide variety of tasks by developing appropriate software. Software is nothing but a set of instructions for the computer to carry out specific tasks. The software is also known as computer program. The task of developing software is known as programming and the person carrying out the task of programming is known as a programmer. Programming means developing a sequence of instructions to a computer to provide the solution to a problem. A programmer develops that sequence of instructions or the program. This involves:

Self-Instructional Material 5 Introduction to Programming NOTES • Defining the problem • Designing a solution to the problem • Writing the program • Executing the program in the computer system and testing that it works correctly 1.2.1 Definition of a Problem What is computer programming? It is the process of finding a solution to the real-life problems with the help of computers. For instance, in the case of oil exploration, you write a program to determine whether oil can be found in the given location. Every problem has to be understood correctly before finding a solution to it. Computers cannot do magic; they cannot give a solution if the problem is not defined correctly. The programmer or software professional must first define the problem and then convert it into computer-understandable form. It may be difficult to solve a problem as a whole at one stroke. Dividing a problem into a number of small problems makes it easier to solve complex problems such as oil exploration or weather forecasting or even launching a satellite. However, such a division of problem is possible only when the problem has been defined completely and correctly. The problem has to be defined clearly and unambiguously, instances of which are given below: Example 1: To find out whether a triangle with given sides is a right-angled triangle: Solution: This problem is solved when the conclusion is given based on data. The user gives the input data, i.e., the length of three sides of a triangle and the computer has to find out whether a right-angled triangle can be formed with the given sides. Example 2: Generation of first 10 fibonacci numbers. Solution: Here, virtually there is no input data, but the program has to generate the first 10 fibonacci numbers. The fibonacci numbers are defined in mathematics. Therefore, the output should match the definition of fibonacci numbers. The program itself has to generate the numbers. There is only one series, which is called the Fibonacci series. Otherwise, the problem would not be defined. Example 3: Conversion of upper case English letters to lower case and vice versa. Solution: This problem is also clear. Here, the input range is 26 upper/lower case alphabets and the task is the conversion of the input data into the corresponding lower or upper case letters. Therefore, the input data has to be supplied at the time of execution of the program or before and the computer should give the expected output. The solution to a problem has also to be determined manually for a set of input data in order to test the correctness of the program. Therefore, problem definition has two parts, i.e., input data and output. If these are well defined, then the problem is defined. A computer is best suited for problem solving because of its repetitive ability and correctness of solutions to problems. Human beings may not always be able to give the same output for the same input. The computer will be able to give the same output consistently irrespective of who operates the computer or when and where the computer is operated. Therefore, once a program is written for solving a problem, even an ordinary user can use it and get the results. Though the experts developed the word processor package, it can be used by anybody who knows English or the language in which the package is written. The developers of word processor packages have put in a lot of effort to ensure that even an ordinary user will be able to operate the software and get the output in whatever manner he desires. Even in the case of word processing software,

6 Self-Instructional Material Introduction to Programming NOTES the inputs and outputs are clear to anybody. The input is the process of typing a predefined set of characters and the output is the desired document. 1.2.2 Designing of Solution to the Problem We cannot expect a computer to solve a problem on its own. A methodology for problem solving has to be evolved by the human beings. Then the computer has to be instructed as to how to solve the problem. It will carry out the assigned tasks in a faithful manner. 1.3 ALGORITHM The methodology for problem solving using a computer is known as algorithm. An algorithm gives a step-by-step instruction that could be converted into statements in a programming language understandable by a computer. A simple definition of algorithm is given below: 'An algorithm is a computable set of steps to achieve a desired result'. Thus, an algorithm is a collection of steps. Each step can be converted into a program statement. The set of steps helps in solving the given problem. According to the IEEE standard, an algorithm is 'A finite set of well-defined rules for the solution of a problem in a finite number of steps'. Thus, an algorithm is a set of precise steps. Each step indicates the operation to be performed clearly and unambiguously. The steps are narrated in a precise and simple form. The order of operations is also important. It starts from the top and progresses sequentially unless otherwise specified. A computer program is essentially an algorithm expressed in a programming language. It instructs the computer to perform specific operations. Thus, the steps in an algorithm have corresponding statements in the program. 1.3.1 History of Algorithms A book titled 'Kitab al-jabr w'al-muqabala' (Rules of Restoration and Reduction) was written by an Iranian scientist named Abu Ja'far Mohammed ibn Musa-al- khwarizmi to introduce algebra to the people in the West. The word algebra came from al-jabr in the book title. The word algorithm evolved from English algorisme, which in turn came from Latin algorismus. 1.3.2 Documenting Algorithms The programmer must remember that the computer cannot study the entire problem at one go and give a solution. It is the programmer who makes the computer carry out the operations so as to arrive at a decision. It has to be told to check or calculate each step at a time and finally give the solution. Therefore, a problem requires an algorithmic or step-by-step approach for solution. After a problem is defined, the programmer designs an algorithm giving a step-by-step instruction to solve the problem. Algorithms can be documented in a number of ways as given below: • Natural languages • Pseudo code • Flowcharts • Programming languages

Self-Instructional Material 7 Introduction to Programming NOTES Natural language algorithms tend to be ambiguous and lengthy. Hence, they are not usable for complex algorithms. Documenting programming languages is not necessary since it may not be worth the effort to be put in. Pseudo codes and flowcharts are the structured ways to express algorithms that avoid many of the ambiguities common in natural language statements, while remaining independent of a particular programming language. 1.3.3 Pseudo Code This is currently the most popular tool for documenting algorithms. It is an abbreviated version of the actual computer code and hence, it is known as pseudo code. It looks like a computer code. The IEEE standard defines a pseudo code as 'A combination of programming language constructs and natural language used to express a computer program design'. A sample algorithm in pseudo code to find out whether a triangle is right angled or not is as follows: Algorithm of a right-angled triangle Step 1: Read the length of the three sides Step 2: Store them as side1, side2 and side3 Step 3: Find the square root of the sum of the squares of side2 and side3 Step 4: If it is equal to side1, go to step 10 Step 5: Else, find the square root of (side1 square + side3 square) Step 6: If it is equal to side2 go to step 10 Step 7: Else, find the square root of (side1 square + side2 square) Step 8: If it is equal to side3, go to step 10 Step 9: Else, print 'The sides do not form a right-angled triangle'. END Step 10: Print 'The sides form a right-angled triangle'. END An algorithm should take care of the termination of a program. Termination of a program can be indicated by 'End'. Programs are, in fact, the implementation of algorithms in a programming language. Therefore, every algorithm can be converted into a program. 1.3.4 Testing Algorithm When an algorithm has been developed, it has to be tested with selected inputs. For this purpose, you have to determine what the expected output for each input is. Inputs are nothing but the data, which is to be supplied for the execution of a program while output is nothing but the information or decision conveyed by the computer. Therefore, an algorithm consists of data or inputs, a procedure that uses the data and leads you to a conclusion and lastly, the communication of the conclusion/decision/information. If you look at the example of the right-angled triangle, the sides of the triangle are inputs; checking the square root of the sum of the square of two sides with the third side is the procedure and the conclusion whether the given triangle is a right-angled triangle or not is the output. In the above, if you give input as 3, 4 and 5, the output will be according to step 10. If you give input as 4, 5 and 6, you will get output according to step 9. Therefore, testing is confirming that the actual output is the same as expected. Using a pencil and paper and noting down what happens at each step can test algorithms.

8 Self-Instructional Material Introduction to Programming NOTES 1.3.5 Some Simple Rules governing Algorithms • The steps in the algorithm shall be convertible to computer instructions. Each step in the algorithm may be converted into one or more instructions in the computer language. • The algorithmic steps shall be definite. It shall not be ambiguous. It cannot contain vague statements or inconclusive statements such as dividing zero by zero. • The algorithm steps shall be implementable manually by a human being. • An algorithm may receive zero or more inputs. • The result of an algorithm is production of one or more output. It has to deliver an output in whatever form it may be. • An algorithm shall terminate at some point in time. It shall not contain any endless loop. • An algorithm has to be well structured with the first instruction on the top of the program. • It may contain alternate end points depending on some context, but must definitely end. • An algorithm should be validated by the chosen sample inputs. It should be possible to prove that the algorithm solves the problem correctly with valid and invalid inputs. 1.3.6 Divide and Conquer The most popular method of problem solving is divide and conquer. This means that the problem has to be divided into smaller problems, each of which must be solved to get the complete solution. For instance, if it is required to find the second smallest element in an array, the problem could be divided into two parts—arranging the elements in the array in an ascending order and then getting the second smallest element from the sorted array. This approach is called the divide and conquer strategy. In order to find the sum of the digits in a number, it can be divided into three parts: • Finding modulus of 10, i.e., finding the remainder when 10 divides the number • Adding the remainder to a sum whose initial value is zero • Dividing the number by 10 and then repeating the process till the quotient is zero An example will make the algorithm easy to understand. For instance, if the given number is 73, the first division by 10 will give the remainder 3 and hence, the sum will be: sum = 0 + 3 = 3. Dividing the number will give the new number = 73/10 = 7. When the process is repeated, you will get the sum of digits = 10. Thus, the problem has been divided into 3 subproblems, which can be solved easily. Now, look at some algorithms, which are often required for problem solving using computers. Algorithm for exchanging values of two variables You often come across a need to swap values contained in two variables in many applications such as sorting. If you simply interchange the values, one of the values will be lost. For instance, if you want to swap values contained in var1 and var2 and if you do as follows, what happens?

Self-Instructional Material 9 Introduction to Programming NOTES var1 = var2 var2 = var1 No doubt the contents of var2 will be transferred to var1. However, when you come to the second statement, var2 will get the current value of var1, which is nothing but the original value of var2. Thus, the value contained originally in var1 will be lost. To avoid this, you need to declare another variable of the same type, say temp. Now, you can achieve swapping in three steps as follows: temp = var1 var1 = var2 var2 = temp Here, the original value of var1 is stored in temp. Now var2 is transferred to var1. Thus, var1 contains the original value of var2. Then the contents of temp, which is nothing but the original value in var1, is transferred to var2. Thus, swapping of values contained in two variables needs three steps and declaring another variable of the same type. The complete algorithm to swap two integers is as follows: Algorithm for exchanging (swapping) values • Read var1 and var2 • Declare temp as integer and assign a value of zero to it • temp = var1 • var1 = var2 • var2 = temp • Write var1 and var2 • End Note that the above algorithm receives two inputs (read) and two outputs (write). The algorithm terminates after writing the values, which is indicated by the End statement. Now, try to apply the simple rules discussed earlier to the above algorithm and confirm that this algorithm possesses all the characteristics of an algorithm. No particular notation for scripting the algorithm has been used. As you go along, you will reduce the description part of the steps by indicating the operations to be carried out symbolically. As such, there is no need to use any particular syntax for documentation of the algorithms. Decimal base to binary base conversion The base of a number system is also called radix. While programmers are comfortable with decimal numbers, the computer system uses the binary number system internally. Let us evolve an algorithm for the conversion of a decimal number into binary number. This algorithm deals with whole numbers only. Let us recall how the conversion takes place with an example. Let us convert 19 into binary form. 2 |19 2 |9 – 1 2 |4 – 1

10 Self-Instructional Material Introduction to Programming NOTES 2 |2 – 0 2 |1 – 0 0 – 1 The equivalent of 19 in binary is 10011. Therefore, when you divide by 2, the first remainder is the least significant bit (LSB) and the last remainder is the most significant bit (MSB). For simplicity, assume that you convert them into 8-bit numbers. The following algorithm gives the method of converting a decimal number to an array of bits. Algorithm for decimal to binary conversion STEP 1: Store the decimal number in integer variable num STEP 2: Declare an array b of size 8 to store 8 bits STEP 3: int I = 0 STEP 4: while (I &gt;= 7) {b[I] = num % 2 ; write b [I] ; num = num/2 ; I = I + 1 ; } STEP 5: End You are using a while loop in the above program. When the associated condition is satisfied, in this case (I &gt;= 7), the block of statements following the 'while' will be executed. A block begins with an opening brace and ends with a closing brace. In this case, there are four statements in the block. Now, evaluate what happens in the write operation in each iteration of the while loop. Iteration 1 b [ 0 ] = 19 % 2 = 1 Iteration 2 b [ 1 ] = 9 % 2 = 1 Iteration 3 b [ 2 ] = 4 % 2 = 0 Iteration 4 b [ 3 ] = 2 % 2 = 0 Iteration 5 b [ 4 ] = 1 % 2 = 1 Iteration 6 b [ 5 ] = 0 % 2 = 0 Iteration 7 b [ 6 ] = 0 % 2 = 0 Therefore, 19 10 = 0010011 b [ 0 ] = LSB b [ 7 ] = MSB Thus, the algorithm does the conversion of decimal number into binary number correctly.
Self-Instructional Material 11 Introduction to Programming NOTES 1.4 FLOWCHARTS A flowchart is a schematic representation of an algorithm. A flowchart is used to list the precise steps in an algorithm. It consists of geometrical shapes (boxes) of various types connected together. It indicates the flow of control during program execution. The

| 96% | MATCHING BLOCK 1/126 | W |
|---|---|---|
| flow lines have arrows to indicate the direction of flow of control between the boxes. The | | |

operation carried out at each step is written within the box in simple English. Thus, a flowchart is a graphical illustration of the steps involved in arriving at a computer solution to a problem. The flowchart shows the sequence of steps performed and also the decision as to which step is to be performed next. It is too primitive to the modern programmer. 1.4.1 Graphical Symbols used in Flowcharts Figures 1.1 (a, b) indicate how to depict the start and end operations during program execution. Start End Fig. 1.1 (a) Start Operation Fig. 1.1 (b) End Operation Any computational operation such as assigning value to a variable or adding two numbers can be depicted as shown in Figure 1.2. var1 = var2 * var3 Fig. 1.2 Computational Operation Quite often, in programming, you may have to take a decision based on the occurrence of an event. This can be graphically illustrated as shown in Figure 1.3. Yes No Is (var1= var3)? Fig. 1.3 Representation of Decision Input or output is indicated by a parallelogram as shown in Figure 1.4. Fig. 1.4 Representation of Input or Output
12 Self-Instructional Material Introduction to Programming NOTES 1.4.2 Basic Control Structures In programming, there are three basic control structures as follows: • Simple sequence • Selection pattern • Repetition pattern These will be discussed briefly. Simple sequence: This is the simplest and the most often used control structure. Here, the computer executes one instruction after another in the order given in the program as shown in Figure 1.5. Instruction 1 Instruction 2 Fig. 1.5 Order of Execution of a Simple Sequence Selection pattern: In this case, the computer evaluates a condition. Then, depending on the outcome of the evaluation, the control flows in one of the paths. Once the conditional execution is finished, the control flows rejoin. An example is shown in Figure 1.6. False True A&lt;0 SQUARE A Error Fig. 1.6 Representation of a Selection Pattern Repetition pattern: In this case, on some condition(s) the execution of instructions loops back to a previous instruction as shown in Figure 1.7.
Self-Instructional Material 13 Introduction to Programming NOTES Yes No Is (n=0)? m = n n = m % n GCD = m Fig. 1.7 Representation of a Repetition Pattern Now, let us draw a flow chart for determining whether a triangle whose three sides are given, is right angled or not. If the square root of the sum of the squares of any two sides is equal to the third side, it is a right-angled triangle. You have to check this by taking two sides at a time. The flow chart is given in Figure 1.8. No Yes No No Yes Yes Is (x=side1)? Start Read side1, x = √ (side2) 2 + (side3) 2 y = √ (side1) 2 + (side3) 2 Is (y=side2)? z = √ (side1) 2 + (side2) 2 Is (z=side3)? Write 'Not Right angled' END Write 'Right- angled Triangle" Fig. 1.8 Flowchart for Checking Right-angled Triangle

14 Self-Instructional Material Introduction to Programming NOTES 1.4.3 Flowchart vs Pseudo Code Flowchart: It is easy to understand and explain. Since it occupies a lot of space, it is not suitable for larger programs. Sometimes, the designers are forced to omit steps in order to fit it in a page. Hence, professional programmers do not prefer flowcharts Pseudo code: It is easy to develop and maintain. A pseudo code can be developed using C language constructs by a C programmer. Hence, it reduces the time taken for the conversion of algorithm to program code later. A pseudo code occupies less space and takes less time for documenting. However, if it is not properly indented and aligned while writing, it may be difficult to understand the program logic. However, modern and professional programmers use pseudo code to document the design of a program. Nevertheless, it is reiterated that program development should follow the following four steps: • Defining the problem and documenting in natural language (English) • Designing a solution algorithmically and documenting the algorithm • Writing the program • Testing the program 1.5 STRUCTURED PROGRAMMING CONCEPT Two mathematicians Corrado Bohm and Guiseppe Jacopini proved that any computer program could be written only with three program structures as follows: • Sequences • Decisions • Loops This discovery is considered to be a precursor to the evolution of a methodology for modern programming known as structured programming. Edsger Dijkstra even advocated that the goto statement should be abolished from all high-level languages because of its ill effects on programs. The programs that include the goto statement can cause innumerable problems, particularly during the maintenance of the software. The program, which uses the goto statement, is called spaghetti code, i.e., a code that has no simple direct logical structure. Structured programming is the name given to good programming practices. It is a preferred methodology for programming in the procedure-oriented languages. Structured programming consists of guidelines for designing programs. The structured programming concepts were evolved to improve the quality of programs. 1.5.1 Features of Structured Programming It is a philosophy and a concept that facilitates designing of programs that are easily understandable, modifiable and do not cause surprises. Some of the features of structured programming are given below: • Flow of control in the program should be as simple as possible. • The program should be constructed using independent modules or functions or subroutines. Check Your Progress 1. What is computer programming? 2. Define algorithm. 3. In how many ways can an algorithm be documented? 4. What are the differences between a flowchart and a pseudo code? 5. What are the three basic control structures?

Self-Instructional Material 15 Introduction to Programming NOTES • It uses only three types of logical structures as follows: o Sequences—statements that are executed one after another o Decision—one of the two blocks of a program code is executed based on the outcome of testing a particular condition. An example is the if...else structure o Loops or iteration—one or more statements are executed repeatedly as long as a particular condition or a combination of conditions remains true. The program is constructed frequently designed using the top-down design methodology. The top-down design methodology involves designing the overall program structure first, followed by designing individual functions. The opposite of the top-down design methodology is the bottom-up methodology, wherein individual functions are designed first. In either case, each function is coded as a separate module. This modularization has many advantages. One of them is that the modules can be reused in other programs. The modules are tested individually and later integrated with other modules. The use of iteration constructs such as for, while, etc. are encouraged in structured programming. The modules should be designed in such a manner that they have one entry point and one exit point. Modular programming automatically ensures structured programming. Structured programming also means that you develop an understandable and maintainable program. Some of the popular codes of good programming practices that fit in the definition of structured programming are as follows: • Write only one statement per line • Coin meaningful names for constants, variables and functions • Use capitals for the names of constants • Divide programs into functions • Each function performs one task • Each function must have at least one comment statement • Skip a line between functions • Skip a line after the declaration statement in the main function as well as each function • Bring clarity by skipping lines wherever required • Put only one brace on each line • Align all opening braces and closing braces • Indent as much as possible to bring out logical structure of the program • Some of the program statements that could be indented are body of function, body of loop, body of the if...else statement • Indent each case in the switch statement • Indent an item within a struct of definition or declaration Many high-level languages such as ADA, Pascal, C, FORTRAN, etc. support structured programming.

16 Self-Instructional Material Introduction to Programming NOTES Features to be avoided Use of the following is discouraged in a structured program: • goto statements • break or continue in the middle of loops • Multiple exit points in a function • Multiple entry points in a function 1.5.2 Advantages of Structured Programming The primary purpose of structured programming is to create the right programs right first time and every time. It enables design of programs that are correct, understandable, testable and modifiable. The structured programming results in additional advantages as given below: Parallel designing: Since the top-down structured design is advocated during the high- level design of the program, a number of modules have to be identified with clear specifications. Therefore, a number of programmers can work in parallel designing one module. Furthermore, the programmer will understand the role of each module in the overall program structure, since he has access to the big picture from the top-level design document very early in the project. It has also been found that structured programming reduces the time to develop programs, because of work parallelism as explained above as well as clarity in the complete structure of the program. Furthermore, structured programming facilitates divide and conquer of the problem, thereby helping in focussing on one problem at a time. All these lead to quick turnaround time for the development of programs. Reusability: Reusability is facilitated by the modular design of the program. Each module is a good candidate for reusability in different programming projects. Ease of debugging: Since each function is specialized and performs only one task, the functionality can be checked individually. Small codes will be easy to debug rather than large programs. Since there will be clarity of the specifications of each function, wrong understanding will be avoided while developing and testing the program. Enhanced understandability: Structured programming advocates adequate comment statements and documentation. It also advocates coining of meaningful names for the functions as well as variables and constants. All these facilitate ease of understanding of the complete program. Ease of maintenance: The time taken to locate the module that contains error will be much shorter in modular structured programs. Since the functions are cohesive and there is lesser coupling between functions, the side effects of maintenance will also be minimal. Thus, structured programming has become very popular in procedure-oriented programming. 1.6 PROGRAMMING ENVIRONMENT Computer program or software is nothing but a set of instructions to a computer to carry out specified tasks. The instructions are written in a language understandable by the
Self-Instructional Material 17 Introduction to Programming NOTES computer. Programs can be developed and executed only in a computer system. Programming environment consists of the following: • Computer hardware, which has a physical appearance. It consists of central processing unit (CPU), keyboard, main memory, secondary storage devices such as floppy disc drive, hard (Winchester) disc drive, compact disc drive, video monitor, printer, modem, sound card, input/output port, etc. • Operating system, which is also the software required for the operation of the computer system. It is known as system software. Popular operating systems include Linux, UNIX and Windows family of operating systems. • Integrated (program) development environment (IDE). Examples include GNU gcc compiler, Borland C++ compiler, Turbo C++ integrated development environment, Microsoft Visual Studio, etc. At this stage, it is important to understand what a computer programming language is and what the various types of computer languages are. 1.7 SOFTWARE CLASSIFICATION Software can be classified into two types—system software and application software. System software is used for operating the computer system and its associated programs. They also provide a platform for the development of programs (applications). This includes the operating system, utilities, compilers, assemblers and interpreters. Application software is developed for fulfilling the specific needs of a user. Examples include software packages such as accounting packages, banking software, games packages and many programs developed for specific purposes such as attendance monitoring, rail reservation, fingerprint analysis, etc. Both system and application software are developed in programming languages. For instance, the UNIX operating system is written in 'C' language. 1.8 PROGRAMMING LANGUAGES Programming languages are, in turn, classified into machine language, assembly language, high-level language, fourth-generation languages (4GLs) and fifth-generation languages (5GLs). 1.8.1 Machine Language Since a CPU is a hardware device, it can only recognize voltages. A voltage, which is relatively high, may be recognized as binary 1 and a voltage that is relatively low as binary 0. Therefore, instructions to a computer in the machine language will consist of 1s and 0s. The binary digits (1 and 0) will, in turn, be converted into the corresponding voltages. Thus, the processing unit of a computer can directly recognize the machine language. The machine language is unique to hardware. Any computer hardware can understand only its machine language. A computer is always made to perform user- intended tasks by using the machine language. This is not transparent to the users of the modern times. However, when the computers were invented, computer inventors began operating computers with the machine language, which consists of zeros and ones. Assume that the instructions are made up by the combination of 5 bits, 1s or 0s. Each

18 Self-Instructional Material Introduction to Programming NOTES one of the 32 different states, which can be arrived at by the combination of 1s and 0s, can represent 32 different instructions in the machine language. For instance, 00000 could be for 'add'; 00001 for 'subtract'; 000010 for multiplication and so on. Each computer has its own machine code for various operations. In this case, the hardware is designed to perform 32 different operations depending on the input, i.e., the instruction. Programming involves writing a meaningful set of instructions in the machine code and is therefore, machine-dependent. It is very difficult and cumbersome for an ordinary programmer to use machine language since the interaction is directly with the hardware. However, programs written in any high-level language have to be finally converted into machine language for executing tasks in a computer. Machine language is called a first- generation language. 1.8.2 Assembly Language Assembly language is known as a second-generation language. Each processor has an assembly language consisting of symbolic names for instructions. Each symbolic name has a corresponding machine code consisting of 1s and 0s. For instance, ADD represents addition; MVI represents move immediate the number; SUB for subtraction. These mnemonics or symbolic names are supplied along with each processor or computer. Therefore, you need not know the machine language in order to develop the program. Programming can be carried out in the assembly language, which is more easily understandable. An assembly language is one step better than the machine language in so far as program friendliness is concerned, but in actual practice, in order to execute a program, assembly language has to be converted into the actual machine code. A computer can understand only machine language. The assembler translates the program code formulated in the assembly language into machine language. Programs written in the assembly language will not be as efficient as programs directly written in machine language. Efficiency is nothing but the size of the code. The size of the code determines the time taken for the execution of a program and therefore, a large code takes more time than a code of smaller size. However, it is relatively easier to program in assembly language as compared to machine language. 1.8.3 High-Level Languages High-level languages (HLL) are somewhat similar to English. The popular high-level languages are FORTRAN, COBOL, BASIC, C, C++, Java, etc. It is very easy even for a non-programmer to understand them. High-level languages use normal English words such as do, return and write, which are also called reserved words. Although 'C' language is classified as HLL, it is on a higher level as compared to the assembly language, but on a lower level than other HLLs. The lower the level of the language, the easier the interaction with the hardware. Programs written in a high-level language have to be converted into machine language. No doubt, they will be inefficient. It is much easier to program than the assembly language. High-level languages are known as third-generation languages. They allow symbolic naming of operations like 'continue', 'return', etc. They can be translated into several different machine languages by the respective compilers to suit the specific hardware (CPU). Thus, HLLs are not unique to any processor unlike the assembly or machine languages. Conversion of a code in an HLL to machine code is carried out by a compiler. Many different systems can be programmed in the same manner in an HLL. The relationship between the three generations of languages discussed so far is shown in Figure 1.9.

Self-Instructional Material 19 Introduction to Programming NOTES Fig. C Compiler for A AL A Assembler A CPU A C Compiler for B AL B Assembler B CPU B HLL C Fig. 1.9 Relationship between the Three Generations of Languages With reference to Figure 1.9, CPU A and CPU B receive machine codes after the respective compilers convert them from one HLL, C. While the program is coded in the same HLL, there are two compilers, each corresponding to the specific processor, namely CPU A and CPU B. From Figure 1.9, it will be evident that each CPU has a corresponding assembler and assembly language (AL). For instance, CPU A has a corresponding assembler A to convert the program written in the assembly language of CPU A. Thus, while an HLL is common to all computers, the assembly language and the corresponding assembler is CPU-specific. Hence, programs developed in an HLL can be executed in any machine, which has the corresponding compiler. This is not true for assembly language programming. 1.8.4 Fourth-Generation Languages (4GLs) These are designed to make high computing power available to non-programmer professionals. Examples of 4GL are: • Database Management System (DBMS) • Graphics generator • Statistical analysis functions, etc. The popular DBMS packages such as Oracle, SQL, Access, etc. fall under 4GLs. 1.8.5 Fifth-Generation Languages (5GLs) These are computer languages based on the concepts of expert systems, knowledge- based systems and natural language processing. The expert system development languages such as LISP, Prolog, etc. and Visual programming languages fall under 5GLs. 1.8.6 Assemblers Assemblers are computer programs that translate programs expressed in the assembly language into the machine code. Each computer, therefore, has its own assembler. Programs can be written in the assembly language and converted into the machine code before execution using assemblers.

20 Self-Instructional Material Introduction to Programming NOTES 1.8.7 Compilers Compilers are also computer programs. They translate programs written in high-level languages (HLLs) into the machine code. Assemblers and compilers increase the productivity of programmers and therefore, programmers prefer assemblers and compilers for developing programs. 1.8.8 Interpreters Interpreters, like compilers, are used with HLLs. Interpreters convert each line of a program code into machine language and check the correctness before the next line is typed. Compilers check the full program. Compilers check for all the errors, issue error messages/warnings about any errors found, create machine codes on successful compilation, i.e., when there are no errors in the program. In the case of interpreters, each statement is checked immediately on keying and if any error occurs, development is terminated with error messages then and there. The BASIC language provides interpreters, while other languages provide compilers. 1.9 PROGRAM WRITING AND EXECUTION 1.9.1 Source Code Program statements written in a high-level language for solving a given problem are called source code. The source code is entered in a text editor. Before attempting to enter the source code, you should formulate the algorithm and document it in the pseudo code. The pseudo code may be converted into a code in a high-level language. Then the same has to be typed in a text editor available in the system. 1.9.2 Object Code Now you are ready to compile the source code to get the machine code that will be understood by the CPU in the computer hardware where you want to execute the program. You have to call the compiler and submit your source code for compilation. After the compilation process, there are two possible outcomes as given below: • Errors in the source code • No errors If there are no errors, you get an object code file. If there are errors, the compiler will give error messages. Therefore, the programmer should carefully examine the program and correct all the errors and recompile. When there are no errors, the compiler will automatically generate the object code. The object code is in the machine language. However, it is not yet ready for execution. If your program contains more than one source code file, all the source code files are to be compiled separately to get the corresponding object code. 1.9.3 Linking and Loading During this process, you combine all the object files. Another important code is that corresponding to the particular operating system of the computer system. This is essential for executing the program in the given hardware and the operating system. In order to get an executable file for the source file, the linker will combine all these object codes. Check Your Progress 6. Explain the divide and conquer strategy. 7. What are the features of structured programming? 8. Describe various programming languages.

Self-Instructional Material 21 Introduction to Programming NOTES The executable file will be in the machine code. It can now be loaded into the system for execution by the loader. Then it can be executed and results obtained. The steps involved in program execution are summarized as follows: • Type the program in a text editor • Save and give a name to the program • Compile the program • Look for errors and correct them • Link the object codes to produce executable code • Load the executable code • Execute/run the program and analyse the result The methodology for the preparation of the source code, compilation, linking and execution may vary from system to system. Learn the correct operations of the IDE/ system being used for the various steps given above. 1.10 SUMMARY In this unit, you were introduced to the concept of programming. The task of developing software is known as programming. Programming means developing a sequence of instructions to a computer to provide a solution to a problem. A programmer develops a sequence of instructions or a program. Computer programming is the process of finding a solution to the real-life problems with the help of computers. The methodology for problem solving using a computer is known as algorithm. An algorithm gives a step-by- step instruction that could be converted into statements in a programming language understandable by a computer. Algorithms can be documented in a number of ways such as natural languages, pseudo code, flow charts and programming languages. A flow chart is a schematic representation of an algorithm. The most popular tool for documenting algorithms is called pseudo code. It is an abbreviated version of the actual computer code and hence, it is known as pseudo code. Programming languages are classified into machine language, assembly language, high-level languages, fourth- generation languages (4GLs) and fifth-generation languages (5GLs). 1.11 KEY TERMS • Programming: The task of developing software is known as programming. Programming means developing a sequence of instructions to a computer to provide a solution to a problem. • Algorithm: The methodology for problem solving using a computer is known as algorithm. An algorithm gives a step-by-step instruction that could be converted into statements in a programming language understandable by a computer. • Flowchart: A flowchart is a schematic representation of an algorithm. • Pseudo code: The most popular tool for documenting algorithms is called pseudo code. It is an abbreviated version of the actual computer code and hence, it is known as pseudo code.

22 Self-Instructional Material Introduction to Programming NOTES • Structured programming: Structured programming is the name given to good programming practices. It is a preferred methodology for programming in the procedure-oriented languages. Structured programming consists of guidelines for designing programs. 1.12 ANSWERS TO 'CHECK YOUR PROGRESS' 1. The task of developing software is known as programming. Programming means developing a sequence of instructions to a computer to provide a solution to a problem. 2. The methodology for problem solving using a computer is known as algorithm. An algorithm gives a step-by-step instruction that could be converted into statements in a programming language understandable by a computer. 3. Algorithms can be documented in a number of ways such as natural languages, pseudo code, flowcharts and programming languages. 4. A flowchart is a schematic representation of an algorithm, whereas pseudo code is the most popular tool for documenting algorithms. It is an abbreviated version of the actual computer code and hence, it is known as pseudo code. A flowchart is used to list the precise steps in an algorithm. It consists of geometrical shapes (boxes) of various types connected together. It indicates the flow of control during program execution. A pseudo code is 'a combination of programming language constructs and natural language used to express a computer program design'. 5. In programming, the three basic control structures are simple sequence, selection pattern and repetition pattern. 6. The most popular method of problem solving is known as divide and conquer strategy. This means that the problem has to be divided into smaller problems, each of which must be solved to get the complete solution. 7. Some of the features of structured programming are as follows: • Flow of control in the program should be as simple as possible. • The program should be constructed using independent modules or functions or subroutines. • Structured programming uses only three types of logical structures. These are as follows: n Sequences—statements that are executed one after another. n Decision—one of the two blocks of a program code is executed based on the outcome of testing a particular condition. An example is the if...else structure. n Loops or iteration—one or more statements are executed repeatedly as long as a particular condition or a combination of conditions remains true. 8.

| 83% | MATCHING BLOCK 2/126 | SA | BCSDSC_1-BASIC CONCEPTS OF COMPUTER & C PROGRA ... (D139109702) |
|---|---|---|---|

Programming languages are classified into machine language, assembly language, high-level language,

fourth-generation languages (4GLs) and fifth-generation languages (5GLs).

Self-Instructional Material 23 Introduction to Programming NOTES 1.13 QUESTIONS AND EXERCISES Short-Answer Questions 1. Is pseudo code better than flowcharting? Explain. 2. How is swap different from exchange? 3. Every loop should have a terminating condition. Why? 4. What is an algorithm? Long-Answer Questions 1. Compare flowchart and pseudo code. 2. Describe the rules governing algorithms using flow chart as well as pseudo code. 3. Write algorithms for solving the following problems: (a) Roots of a quadratic equation (b) Finding the greatest of 3 numbers 4. Explain the importance of loops and rules for framing them. 5. Write short notes on: (a) Top-level design (b) Problem definition (c) Divide and conquer (d) Five Generations of languages 6. Solve the following problems using algorithms given in this chapter. Write what happens in each step. (a) Swap (45, 75) (b) Binary equivalent of decimal number 1024 (c) GCD (21, 84) 1.14

FURTHER READING Gottfried, Byron S. Programming with C, 2nd edition. Schaum's Outline Series. New York: McGraw-Hill, 1996. Forouzan, Behrouz A. and Richard F. Gilberg. Computer Science: A Structured Programming Approach Using C, 2nd edition. California: Thomson, Brooks Cole, 2000. Dey, Pradip and Manas Ghosh. Computer Fundamentals and Programming in C. New Delhi: Oxford Higher Education, 2006. Bronson, Gary J. A First Book of ANSI C, 3rd edition. California: Thomson, Brooks Cole, 2000. Kanetkar, Yashwant. Understanding Pointers in C.

New Delhi: BPB Publication, 2001. Kanetkar, Yashwant. Let us C. New Delhi: BPB Publication, 1999.

Kernighan, Brian W. and Dennis Ritchie. C Programming Language, 2nd edition. New Jersey: Prentice Hall, 1988.

24 Self-Instructional Material Introduction to Programming NOTES Foster, W. D. and L. S. Foster. C by Discovery. Boston: Addison-Wesley, 2005. Kanetkar, Yashwant. Working with C. New Delhi: BPB Publication, 2003. Horton, Ivor. Instant C Programming. New Jersey: Wrox Press (John Willey & Sons), 1995. Lawlor, Steven C. The Art of Programming Computer Science with 'C'. New Jersey: West Publishing Company, 1996.

Self-Instructional Material 25 Introduction to C Language NOTES UNIT 2 INTRODUCTION TO C LANGUAGE Structure 2.0 Introduction 2.1 Unit Objectives 2.2 History of 'C' Language 2.2.1 Developing a C Program 2.2.2 Source Code 2.2.3 Object Code 2.2.4 Linking and Loading 2.3 Program Execution 2.3.1 Executing a C Program in the UNIX System 2.3.2 Entering Program 2.3.3 Compilation 2.3.4 Execution 2.4 Sample C Program 2.4.1 Variation in the Main Function 2.5 Tokens 2.5.1 The C Character Set 2.5.2 Identifiers 2.5.3 Keywords 2.5.4 Data Types 2.6 Variables 2.6.1 Size of Variables 2.7

Constants 2.7.1 Integer Constants 2.7.2 Character Constants 2.7.3 Floating Point or Real Numbers 2.7.4 Enumeration Constant 2.7.5 String Constants 2.7.6 Symbolic Constants 2.8

Type Modifiers 2.9 Escape Sequences 2.10 Arrays 2.10.1 Array Declaration 2.11 Expressions and Statements 2.12 Summary 2.13 Key Terms 2.14 Answers to 'Check Your Progress' 2.15 Questions and Exercises 2.16 Further Reading 2.0 INTRODUCTION In the previous unit, you were introduced to programming. In this unit, you will get in- depth understanding of the C language. Dennis Ritchie developed the C language in 1972 under the influence of BCPL and B. The language is, however, rich in data types unlike the other two languages. It is universal. It can be developed and executed in any platform that has a development environment for C. It can run under UNIX, LINUX and even in parallel computers. In this unit, you will also gain knowledge of tokens. There are six classes of tokens in the C programming language, namely keyword, identifier, constant, string literal, operator and punctuators. The unit also discusses arrays, which is

26 Self-Instructional Material Introduction to C Language NOTES another form of data type. There can be
arrays of integers, arrays of characters and arrays of floating-point numbers.

An array means a collection of a number of integers or floats or items of the same data type. An array contains data of the same type.
2.1 UNIT
OBJECTIVES After going through this unit, you will be able to: •
Elucidate the
history of the C language •
Develop a C program • Explain different types of tokens • Describe variables and constants and their various types • Define arrays and declare them • Understand expressions, statements and symbolic constants 2.2 HISTORY OF 'C' LANGUAGE Martin Richards developed a language called BCPL in the mid-1960s. It had many good features. In the year 1970, Ken Thompson, also working at AT&T Bell Labs USA, developed a language called B, which was influenced by BCPL. He developed B as a compact language for system programming. Subsequently, in 1972, Dennis Ritchie, also working at Bell Labs, designed and implemented C language on the UNIX operating system on a Digital Equipment Corporation (DEC) PDP-11 computer. The C language was, therefore, developed under the influence of BCPL and B. It is, however, rich in data types unlike the other two languages. The B language can be considered as the C language without types. C added data types such as char and float to B. By 1973, the C language was ready. It further added new features between 1973 and 1980 and was used to write the UNIX kernel for the PDP-11 computer. The languages BCPL, B and C are procedure-oriented languages similar to FORTRAN. However, they are more compact having few keywords. Simple compilers translate them. They use programs in the standard library for input/output and other interactions with the operating system. During the 1980s, the C language compilers became available for most computer architectures and operating systems. It became a programming tool in PC, which increased its popularity. Ritchie along with Kernighan published a book 'The C Programming Language' in 1978. The Kernighan and Ritchie (K&R) description of the language became a kind of industry standard, popularly called K&R C. Since then, as many C compilers came into existence and there were minor compatibility problems due to variations in implementation. The American National Standards Institute (ANSI) formed a committee (X3JII) for bringing out a standard for the C language. This committee brought out a standardized definition of the C language in 1989. The international standard on the C language, namely ISO/IEC 9899 was released in 1990 by adopting the ANSI standard on the C language. This standard is called C90. Subsequently, it was revised in 1999. The revised standard is called C99. Thus, there are three forms of C, namely K&R C, C90 and C99. Even thereafter, minor amendments to the standard are taking place. You will learn the C language that conforms to the ISO/IEC 9899: 1990 standard, hereinafter called the Standard.
Self-Instructional Material 27 Introduction to C Language NOTES Note: ISO—International Organization for Standardization IEC—International Electro-technical Commission 2.2.1 Developing a C Program A number of integrated development environment (IDE) for developing and executing C programs are available in the market. Freeware compilers for C and a host of other languages are available on the Internet from an organization called The Free Software Foundation. This is called gcc C of GNU. It can be downloaded from the following URLs: http://www.gnu.ai.mit.edu/software/gec to work with UNIX operating system
http://www.delorie.com/djgpp/ to work under DOS/ Windows environment A typical methodology for the development of a C program using GNU C in the Windows environment in the DOS prompt is given in Figure 2.1. START Design algorithm Enter source code in text editor Save file Compile Edit source code Errors Linker Object code of the relevant C library functions eg. &gt;stdio.h&lt; Other object files Errors Get executable code Execute run or END Source Code Yes No Operating system- related code Get object code Check result Yes No Fig. 2.1 Steps involved in developing a C Program
28 Self-Instructional Material Introduction to C Language NOTES The various steps involved in program development as given in Figure 2.1 are briefly discussed below. 2.2.2 Source Code The program statements written in a high-level language are referred to as source code. The source code is entered in a text editor. Before attempting to enter the source code, you should formulate the algorithm and document it in pseudo code. The pseudo code may be converted into the C language code and typed in the text editor available in the system. After entering the code, it may be saved in a file with file name .c extension. You can save the file as Example 2.1.c 2.2.3 Object Code Now you are ready to compile the source code. You have to call the compiler and submit your source code for compilation. If you are using a GNU compiler, you type it in the DOS Command for compilation as follows: gcc–2.1.c If you are using Turbo C++, you select compile and click. After the compilation process is over, there are two possible outcomes: • There are errors in the source code. • There are no errors. If there are no errors, you get an object code file with the same filename with .o extension automatically. In case of the above-mentioned example, you will get Ex 2.1.c. If there are errors, the compiler will give error messages. The error messages will give the line numbers in the program along with the type of error. The programmer should note the line numbers and the error messages and then edit the source file. The statements in the line numbers should be checked for errors. Therefore, the programmer should carefully examine the program and correct all the errors and recompile. When there are no errors, the compiler will automatically generate the object code and save it with a .o extension with the file name same as that of the corresponding source code. The object code is in the machine language. However, it is not yet ready for execution. If your program contains more than one source code file, all the source files are to be compiled separately to get the corresponding object code. Assume that you have another source file called px.c. You will get px.o when the compilation is successful. 2.2.4 Linking and Loading During this process, you combine all the object files. In this case, Ex 2.1.c and px.o. Additionally, you add the object code corresponding to the header files included in the source code or program under execution. For instance, you will combine the object code corresponding to the printf() function in the &gt;stdio.h&lt; file. Another important code is that corresponding to the particular operating system of the computer system. This is essential for executing the program in the given hardware and the operating system. The linker to get an executable file for the source file will combine all these. The executable file will be in the machine code. It will be generated automatically on successful linking and loaded for execution. You get Ex 2.1.c.exe in this case after successful linking.

Self-Instructional Material 29 Introduction to C Language NOTES 2.3 PROGRAM EXECUTION In the DOS Prompt, you can execute the program by typing Ex 2.1.c.exe. If you are using Turbo C++, you can select RUN and click. The result will be displayed by the computer system on the monitor. The programmer may check the specific methodology for all the above steps in the development environment used by him. However, it will not be difficult to write a program and start executing the program. The programs may also be executed under other operating systems such as UNIX, DOS and in any other computer hardware such as Sun Workstations and IBM computer systems ranging from PCs to mainframes, etc. 2.3.1 Executing a C Program in the UNIX System The C language is universal. The examples given in the book were developed and executed in a PC running the Windows operating system. However, it can be developed and executed in any platform that has a development environment for C. It can run under UNIX, LINUX and even in parallel computers. UNIX was the system in which Dennis Ritchie executed the first C program. The following are the steps to be carried out for program development and execution in a system with a typical UNIX operating system: 2.3.2 Entering Program The C program can be entered in one of the UNIX editors such as ed or vi. After entering the program, you have to save it with a lower case .c extension. For instance, you can save it as Example 2.1.c 2.3.3 Compilation The compiler in the UNIX system is called cc. You can compile the above program by typing the following: cc–Ex 2.1.C If there are errors, the program will not compile. The errors will be listed. The errors are to be corrected. On clean compilation, the system generates an executable file called a.out. 2.3.4 Execution You execute the above-mentioned file a.out and the result will appear on the screen. Whenever a file is compiled as mentioned above, and if there are no errors, the system will produce an executable file with the name a.out. Therefore, the old executable files will be erased. If you want to retain all the executable codes for future use, you have to rename it. From the above discussions, it will be clear that the object file is not available. On clean compilation, the object code produced is converted into an executable code and placed in a.out. Hence, the object file is not available. Suppose, you want to retain the executable code in Ex 2.1.c, you can compile the program as follows: cc–2.1.C This will result in having a copy of the executable file in Ex 2.1.c
30 Self-Instructional Material Introduction to C Language NOTES Thus, you have to understand the applicable commands for editing, compiling, linking and executing the program in a computer system. Once it is learnt, it will hold well forever in the system. 2.4 SAMPLE C PROGRAM A program in the C language is given below, which displays a text. Take a close look at the program: /* Example 2.1*/ /* program for displaying a text */ #include &gt;stdio.h&lt; int main() { printf ("Om Vinayaga") ; } Create a new file in the text editor. Then type in the program exactly as given above. Save this program as Example 2.1. Next, compile the program. The compiler after compilation will give a message. Look at the message box. If the C program was compiled in a C++ compiler, there may be warnings, but you can safely ignore them. However, the errors, if any, should not be ignored. The compiler will give the errors and line numbers. Sometimes, even an experienced programmer will find it difficult to understand the error messages. If you encounter the same difficulty in understanding the error messages, you need not worry. Open the program file again and check whether you have typed it exactly as in the book including the semicolons, quotation marks and brackets and the program is a working program. Keep checking and compiling till the compiler says 'success', meaning there is no error in the program after compilation. Now, you have to link the object code to get the executable code. When there are no errors, even after linking, you have to execute or run the program. Use the right command for 'run'. On execution,
you will get the result as follows: Result of the program
Om Vinayaga You have succeeded in establishing a communication link with the computer. You can now talk to it regularly by learning new commands and using better communication methods. You have now become familiar with the methodology for learning a programming language. The steps involved are summarized as follows: • Type the program in a text editor. • Save and give a name to the program. • Compile the program. • Look for errors and correct them. • Link the object code to produce an executable code. • Execute/Run the program and analyse the result. The methodology for the preparation of the source code, compilation, linking and execution may vary from system to system. Learn the correct operations of the IDE/system being used for the various steps mentioned above.
Self-Instructional Material 31 Introduction to C Language NOTES 2.4.1 Variation in the Main Function Most compilers have implemented K&R C. Some have implemented C90. Not many have implemented the new features introduced in C99. Hence, it is important that the reader checks what standard his compiler supports. In the Example 2.1, you did not explicitly return anything. On successful execution, the program returns zero. Return of any other integer means that the program execution was not successful. Hence, you can add a statement 'return 0' at the end of the program. The main function did not receive any value, which was indicated by the empty parentheses. You can indicate nothing by the keyword void. The modified program including the above two features is as follows: /* Example 2.2*/ /* program for displaying a text */ #include &gt;stdio.h&lt; int main(void) { printf ("Om Vinayaga") ; return 0; } Result of the program Om Vinayaga Thus, both the above programs have carried out the task successfully. However, in the rest of the book, the style given in the previous program will be used. The reader may adopt the style that works in his compiler. 2.5 TOKENS The Standard defines six classes of tokens in the C programming language as follows: • Keyword • Identifier • Constant • String literal • Operator • Punctuators Tokens are similar to the atomic elements or building blocks of a program. A C program is constructed using tokens. There are certain other building blocks of a program that do not form part of any of the above. They are as follows: • Blanks • Horizontal tabs • Vertical tabs • New line characters • Form feed • Comments

32 Self-Instructional Material Introduction to C Language NOTES 2.5.1 The C Character Set The C language supports and implements the American Standard Code for Information Interchange (ASCII) for representing characters. The ASCII uses 7 bits for representing each character or digit. The characters are coded from 0000000 (decimal 0) to 1111111 (decimal 127). Therefore, the ASCII consists of a code for 128 characters in all. The ASCII values (decimal equivalent of the 7 bits) of some alphabets and digits are given in Table 2.1. Table 2.1 ASCII Values of Selected Alphabets ASCII Value Character or Digit 48 0 49 1 57 9 65 A 66 B 67 C 89 Y 90 Z 97 a 98 b 121 y 122 z The digits and alphabets are organized sequentially and hence, it is easy to get the ASCII value; for instance, the ASCII value of D is 68, E is 69, 8 is 56, x is 120 and so on. The ASCII table is given in Annexure 1. 2.5.2 Identifiers Any name is an identifier. Just as the name of a person, street or city helps in the identification of a person or a street or a city, the identifier in the C language assigns names to files, functions, constants, variables, etc. An identifier in the C language is defined as a sequence of alphanumeric characters, i.e., alphabets or digits. The first character of an identifier has to be an alphabet. In the C language, lowercase alphabets and uppercase alphabets are considered to be different. For instance, VAR and var represent different names in the C language. VALID IDENTIFIERS C1 PROC1 P34 VAR_1 EX1 a bc Ual1 Aa INVALID IDENTIFIERS 1PROGA 4.3 A-B
Self-Instructional Material 33 Introduction to C Language NOTES Any function name is also an identifier. For instance, 'printf' is the name of the function available with the C language system. The function helps in printing. Therefore, identifiers can be constructed with alphabets (A...Z), (a...z), (0...9). In addition, underscore can also be used in identifiers. Unless otherwise specified, small letters are usually used for identifiers. 2.5.3 Keywords These are also known as reserved words in C. In the first program, 'int' is a reserved word or keyword. They have a specific meaning to the compiler. They should be used for giving specific instructions to the computer. These words cannot be used for any other purpose such as naming a variable. C is a very concise language containing only 32 reserved words and this is one of its strengths. Common statements, such as print, input, etc., are implemented through library functions in C, giving relief to programmers and reducing the size of code as compared to other programming languages. This makes the task of programming simple. Now take a look at the keywords given in Table 2.2. You will use most of them in the book. Their meaning will become clear as you read the book. Table 2.2

**89%** **MATCHING BLOCK 3/126** **W**

C Keywords auto break case char const continue default do double else enum extern float for goto if int long register return short signed sizeof static struct switch typedef union unsigned void volatile while 2.5.4

Data Types Data is used in a program to get information. In a program used to find out the greater of two numbers, the numbers are data and the output, which says which number is greater, is information. C is a versatile language and handles many different types of data in an elegant manner. Bit stands for binary digit, i.e., 0 or 1. Each byte is a collection of 8 bits, i.e., 8 consecutive bits of '0' or '1'. Data is handled in a computer generally in terms of bytes and therefore, will be in the form of multiples of 8 bits. Each ASCII character is represented by one byte. Fundamental data types An item that holds data is also called an object. An object has a name or identifier associated with it. Each object can hold a specific type of data. There are five basic data types in C, as follows: • Character • Integer • Real numbers • Void (
comprising an empty set of values) • Enum (which will be introduced later)
34 Self-Instructional Material Introduction to C Language NOTES
You have to understand how a computer works. Assume that two numbers a and b are to be multiplied. First
of all,
the two numbers have to be stored in the memory. Then the required calculation has to be performed. The result has also to be stored in
the
memory.
Each number is of a specific data type; for instance, all three of them can be declared to be integers. Each data type occupies a specific size in the memory. What does one mean by size? It is the amount of storage space required; each bit needs one storage space. One byte needs eight storage spaces. If a number is of type integer declared as int, it is stored in 2 bytes. The number depending on its type gets stored in different forms. If a number is of float type, it takes 4 bytes to store it. All characters can be represented according to the ASCII table and hence, 1 byte, i.e., 8 bits are good enough to store a character, which is represented as char. These sizes may vary from computer to computer. The header files &gt;limits.h&lt; and &gt;float.h&lt; contain information about the sizes of the data types. Real numbers can be expressed with single precision or double precision. Double precision means that real numbers can be expressed more precisely. Double precision also means more digits in mantissa. The type 'float' means single precision and 'double' means a double precision real number. Table 2.3 indicates the size of various
data
types. Table 2.3 Size of Data Types Data Type Size char 1 byte int 2 bytes float 4 bytes double 8 bytes Maximum and minimum magnitudes The maximum and minimum values of data types are not limitless. For example, &gt;limits.h&lt; specifies the minimum and maximum magnitudes for integers and characters. Since char is stored in a byte, it is as good as a short integer.
char can
be stored as an unsigned character, which means all the 8 bits can be used to store it. The maximum value of an 8-bit number is 255 when all bits are 1. If it is a signed char, the first bit will be reserved for storing the sign. The sign bit will be 0 for a positive number and 1 for a negative number. The integer values of signed and unsigned chars are as follows: CHAR-BIT 8 bits in a
char

SCHAR MAX + 127 maximum value of signed char SCHAR MIN – 127 minimum value of signed char UCHAR MAX 255 maximum value of
unsigned char
Now, let us look at the maximum and minimum magnitudes for the integer data type. They are:
INT MAX + 32767 maximum value of int INT MIN – 32767 minimum value of int
Integer means signed integer. The data type integer occupies 2 bytes or 16 bits. The most significant bit is reserved for sign. It will be '0' for a positive number and '1' for a negative number. Therefore, you can easily calculate how the limits for the integer data types have been arrived at.
Self-Instructional Material 35 Introduction to C Language NOTES
The standard has also provided for another type of integer called short int with the same maximum and minimum values. 2.6 VARIABLES The names of variables and constants are identifiers. The names are made up of alphabets, digits and underscore, but the first character of an identifier must be an alphabet. C allows up to 31 characters for the identifier (names) and therefore, the naming of the variables should be carried out in an easily understandable manner. For example, in the program for the calculation of, Simple interest I = pnr/100, You can declare them with actual names, p = principal, r = rate_of_interest, n = number_of_ years Naturally, programmers
may not
like typing long names for fear of making mistakes elsewhere in the program apart from being reluctant to increase their typing workload. Meaningful names can, however, be assigned to data types even with few letters. For example, p = princ; r = intrate; n = years Some compilers may allow names with upto 31 (thirty-one) characters, but may consider the first eight characters for all purposes. Hence, a programmer could coin shorter yet meaningful names, instead of using single alphabets for variable names. One should, however, be careful not to use the reserved words, such as 32 keywords, for variable names as they have a specific meaning to the compiler. If they are used as variable names, then the compiler will get confused. Be careful not to use the reserved words as identifiers. A program to find out the square of integer 5 is given below. /*Example 2.3*/ /*program to find square of 5*/ #include &gt;stdio.h&lt; int main() { printf("square of %d= %d", 5, 5*5); } Result of the program square of 5= 25 You have now achieved the objective of finding the square of 5. Later on, you may want to find out the square of another number, say 7, for example. You would have to write the same program again replacing 5 by 7 and then compile and run it. This would waste a lot of time. To save time, you can, therefore, write a general-purpose program as follows: /*Example 2.4*/ /*program to find square of any given number*/ #include &gt;stdio.h&lt;
Check Your Progress 1. Who developed the C language and when? 2. What are tokens? How many classes of tokens are there? 3. What is an identifier? 4. What do you mean by keywords?
36 Self-Instructional Material Introduction to C Language NOTES
int main() { int num; printf("Enter the integer whose square is to be found\n"); scanf("%d", &num);
printf("square of %d= %d", num, num*num); } Here you define num as an integer variable. When '&' precedes num, it indicates the memory address of num. At the first printf, the message appears as it is and the cursor goes to the next line because of the new line character \n at the end of the message, but before the closing quotation mark. The next statement is new to you. It is used to receive an integer typed on the console. You can type in any integer number, and the number typed will be stored in the memory at the memory location named 'num'. The purpose of the statement is, therefore, to get the integer (because of the appearance of %d within quotes) and it is stored at the memory address 'num'. The next statement prints the number typed and its square. When you execute the program, the following message appears on the monitor: Enter the integer whose square is to be found. Since you want to find out the square of 25, type: 25 Promptly the reply will be as shown: square of 25 = 625 The next time you may want to find out the square of another number, say 121. Then simply run the program and when prompted, type 121 to get the answer. Here, the number whose square has to be found out has been declared as a variable. The variable has a name and is stored at the location pointed to by the variable name. Therefore, the execution of the program for finding out the square of any number is easy with the above modification
as in Example 2.4.
Variables and constants are fundamental data types. A variable can be assigned only one value at a time, but can change value during program execution. A constant, as the name indicates, cannot be assigned a different value during program execution. For example, PI, if declared as a constant, cannot have its value varied in a given program. If PI has been declared as a constant = 3.14, it cannot be reassigned any other value in the program. Programs may declare a number of constants. Variables are similarly useful for any programming language. If PI has been declared as a variable, then it can be changed in the program to any value. This is one difference between a variable and a constant. Whether an identifier is constant or variable depends on how it is declared. Both variables and constants belong to one of the data types like int, float, etc. The convention in 'C' is to indicate the names of constants by the upper case letters. PI SIGMA
Self-Instructional Material 37 Introduction to C Language NOTES
Variable names are, on the other hand, indicated by the lower case letters: int a float xy 2.6.1 Size of Variables The C programmer should understand how much memory storage each variable type occupies in the IDE used by him. The following example will help you to find the size of each variable type. The result will be in terms of bytes occupied by the variable type. A new keyword sizeof is used to find out the size. The syntax for using the keyword is as follows: sizeof (&gt;data type&lt;) or sizeof (&gt;expression&lt;) Consider the following example: /*Example 2.5*/ /*program to find out the sizes of various types of integers*/ #include&gt;stdio.h&lt; int
main() { printf("size of char =%d\n", sizeof(
char));

printf("size of short=%d\n", sizeof(
short)); printf("size of int =%d\n", sizeof(int)); printf("size of unsigned int=%d\n", sizeof(unsigned)); printf("size of long int=%d\n", sizeof(long)); printf("size of
unsigned long
int=%d\n", sizeof(unsigned long));
printf("size of float =%d\n", sizeof(float)); printf("size of double=%
d\n", sizeof(
double)); printf("size of long double%d\n", sizeof(long double)); } Result of the program size of char = 1 size of short = 2 size of int = 2 size of unsigned int = 2 size of long int = 4 size of unsigned long int = 4 size of float = 4 size of double = 8 size of long double = 10 Therefore, it is obvious that a long double occupies 10 bytes and stores long floating- point numbers with double precision. Note that the size of short int will be either equal to or less than the size of an integer variable.
38
Self-Instructional Material Introduction to C Language NOTES
Variables, which require more than 1 byte for storage, will be stored consecutively in the memory. 2.7
CONSTANTS The following are the types of constants: • Integer constant • Character constant • Float constant • Enumeration constant • String constant • Symbolic constant All these types
will now be explained: 2.7.1
Integer Constants The following are the types of integers: int unsigned int long unsigned long Variations in Integer Types You can use the sign bit also for holding the value. In such cases, the variable will be called unsigned int. The maximum value of an unsigned int will be equal to 65535 because
you
are using the Most Significant Bit (MSB) also for storing the value. The minimum value will obviously be 0. A long integer is represented as long
int or simply long.
The maximum and minimum values of long are as follows: LONG MAX + 2147483647 LONG MIN – 2147483647 Unless otherwise specified, integers or long integers will be signed, i.e., the first bit will be reserved for the sign. The long int obviously uses 4 bytes or 32 bits. The magnitudes of long can also be doubled by using an unsigned long integer denoted as unsigned long. However, integers are not suitable for very low values and very large values. This can be overcome by floating point or real numbers.
An integer constant may be suffixed by the letter u or U to specify that it is
an unsigned integer. Similarly, if the integer is suffixed with l or L, it signifies a long integer. If you specify unsigned long integer you suffix the constant with ul or UL.
Self-Instructional Material 39 Introduction to C Language NOTES

**100%** **MATCHING BLOCK 4/126** **W**

The following are the examples of valid and invalid integers: Valid integers +345 /* integer */ 345 /* integer */ –345 /* integer */ 729u /* unsigned integer */ 729U /* unsigned integer */ –112345L /* Long integer */ 112345UL /* Unsigned Long integer */ +112345l /* Long integer */ 112345l /* Long integer - if no sign precedes, it is a positive number */ Invalid integers 345.0 /* decimal point not allowed */ 112, 345L /* no comma allowed */ 112 345UL /* = blank not allowed */ 112890345L /* exceeds the maximum */ +112 345UL /* unsigned cannot have + */ (345l /* ( not allowed */ –345s /* illegal characters */

You
have so far considered only decimal numbers. C, however, entertains other
types
of numbers as well. The octal numbers will be preceded by 0 (zero). The following are the examples of valid and invalid octal numbers: Valid octal number 0346 0547 0120 Invalid octal number 0394 /* 8 or 9 are not allowed in an octal number */ 0 x 345 /* prefix has to be zero only */ The C language also supports hexadecimal numbers. Here, since the base is 16, we use alphabets also in the numbers as given in Table 2.4. Table 2.4 a or A for 10 b or B for 11 c or C for 12 d or D for 13 e or E for 14 f or F for 15 Additionally, hexadecimal numbers will be preceded by 0X or ox, i.e., zero followed by x.
40
Self-Instructional Material Introduction to C Language NOTES The following are the
examples of valid and invalid hexadecimal numbers: Valid hexadecimal numbers 0x345 0xA132 0x100 0x20B
Invalid hexadecimal numbers 0x, 123 /* no comma */ 0x /* cannot be empty */ 0A00 /* x is missing */ 2.7.2
Character Constants A character constant is a single character enclosed in single quotes as in 'x'.
Characters can be alphabets, digits or special symbols. The following are
the
examples of valid and invalid character constants: Valid character constants 'A' 'Z' 'C' 'c' Invalid character constants '\n' '\t' '\u' '\b' AA 'AA' "AA" '1a' A character constant represents its integer value as defined in the character set of the machine. Therefore, you can add 2 characters. For example, the ASCII values of digit 1 = 49 and C = 67. When you add these values you get code 116 whose equivalent character is t. Now, verify this with the following example: /* Example 2.6 demonstrates that chars can be treated like integers*/ #include &gt;stdio.h&lt; int main() { const char ALPHA1='1'; char alpha2='C'; char alpha3; alpha3=ALPHA1+alpha2;
Self-Instructional Material 41 Introduction to C Language NOTES

putchar(alpha3); }
Result of the program t Therefore, characters can be treated like integers as well, although they are declared as character variables and constants. Since characters are of type int, you could add them. Characters can also be defined as integers as in
the following
example: /* Example 2.7 Demonstrates that a char can also be declared as int*/ #include &gt;stdio.h&lt; int main() { int x; x='1'+'C'; printf("x as integer=%d\n", x);/*x printed as integer*/ printf("x as character=%c\n", x);/*x printed as character*/ } Result of the program x as integer=116 x as character=t 2.7.3 Floating Point or Real Numbers The difference between floating point and integer numbers are as follows: •
Integers are whole numbers without decimal points but a float has always a decimal point. Even if the number is a whole number, it is written with a decimal point. For instance, 42 is an integer, while 42.0 is floating-point number. • Floating-point numbers occupy more space for storage as we have already seen. A real number in the simple form consists of a whole number followed by the decimal point and also one or more decimal numbers following the decimal point, which makes the fractional part. This form of representation is known as
the

**99%** **MATCHING BLOCK 5/126** W

fractional form. It must have a decimal point. It could be either positive or negative. As usual, the default sign is positive. No commas or blanks or special characters are allowed in between. The following are the examples of valid and invalid float types: Valid floats 144.00 226.012 Invalid floats +144 /* no decimal point */ 1,44.0 /* comma not allowed */ 42 Self-Instructional Material

Introduction to C Language NOTES
Scientific notation: Floating-point numbers can also be expressed in scientific notation. For example, 3.0 E 2 is a floating-point number. The value of the number will be equal to 3.0 × 10 2 = 300.0 Instead of the upper case E, the lower case e can be used as in 0.453 e + 05, which will be equal to 0.453 × 10 5 = 45300 There are two parts in the scientific notation of a real number, which are as follows: • Mantissa (before E) • Exponent (after E) In the scientific form, the following rules are to be observed: • The mantissa part can be positive or negative. • The exponent must have at least one digit, which can be a positive or negative integer. Remember the exponent cannot be a floating-point number. type float is a single precision number occupying a storage space of 4 bytes. type double represents floating-point numbers of double precision and hence occupies 8 bytes. If you look at the file &gt;float.h&lt; you will find the maximum and minimum floating-point numbers as shown: FLT – MAX 1E + 37 maximum floating point number FLT – MIN 1E – 37 minimum floating point number Floating-point constants The constants are suffixed as shown: F or f – float no suffix – double L or l – long double If an integer is suffixed with L or l, then it is a long integer. If a float is suffixed with L or l, then it is a long double floating-point number. Examples Valid floating-point constants 1.0 e 5 123.0 f /* float */ 11123.05 /* double */ 23467.34 e 5 l /* long double */ Invalid real constants 245.0 /* invalid float, but valid double */ 456 /* It is an integer */ 1.0 e 5.0 /* exponent cannot be a real number */ When they are declared as variables, they can be declared as follows: float a = 3.12; float a, b, c; float val1;
Self-Instructional Material 43 Introduction to C Language NOTES
float val2; long double val3; The values of constants cannot be altered in programs. They can be defined as follows: const int PRINC = 1000; const float INT_RATE = 0.12 f; The values of PRINC and INT_RATE cannot be changed in the program even by introducing another assignment statement when they are declared as constants using const.
The following
example verifies this statement: /*Example 2.8 Demonstrates that constants cannot be changed even with assignment statements. To verify, include statements 7, 8 & 9 in the program by removing the comment specifiers at the beginning of the program statement 7 and the end of statement 9*/ #include&gt;stdio.h&lt; main() { const int PRINC =1000; const float INTST=0.12f; printf("PRINCIPAL=%d INTEREST=%f\n", PRINC, INTST); /*PRINC =2000; INTST=0.24f; printf("PRINCIPAL=%d INTEREST=%f\n", PRINC, INTST);*/ } Key in the example and execute the program. After successful compilation, you will get the result as given below. Result of the program PRINCIPAL = 1000; INTEREST = 0.1200 Now include the second part of the program by removing /* and */ at statements 7 and 9, respectively. Earlier this was treated as a comment. Now this part will get included in the program. Now compile it. You will get a compilation error. This is due to your attempt to redefine the constants PRINC and INTST, which is not allowed. Incidentally, the technique of including or excluding a program segment at will using /* and */ is a convenient method for program development. 2.7.4 Enumeration Constant The keyword for this data type is enum. You can define guardian as follows: enum guardian { father, husband, guardian };
44
Self-Instructional Material Introduction to C Language NOTES
Note that there are no semicolons, but only a comma after the members except the last member. There is not even a comma after the last member. There is no conflict between both guardians. The top one is enum and the bottom one is a member of enum guardian. See the similarity between structure, union and enum. The enum variables can be declared as follows: enum guardian emp1, emp2, emp3; This is similar to structures and unions. The first part is the declaration of the data type. The second part is the declaration of the variable. The initial values can be assigned in a simpler manner as given below: emp1 = husband; emp2 = guardian; emp3 = father;
You

have to assign only those declared as part of the enum declaration. Assigning constants not declared will cause error. The compiler treats the enumerators given within the braces as constants. The first enumerator father will be treated as 0, the husband as 1 and the guardian as 2. Therefore, it is strictly as per the natural order starting from 0. The enumerated data type is never used alone. It is used in conjunction with other data types. You can write a program using enum and struct.
It is shown as follows: /*Example 2.9 enum within structure*/ #include &gt;stdio.h&lt; main() {
enum guardian { father, husband, relative }; struct employee { char *name; float basic; char *birthdate; enum guardian guard; }emp[2]; int i; emp[0].name="RAM"; emp[0].basic= 20000.00; emp[0].birthdate= "19/11/1948"; emp[0].guard= father; emp[1].name="SITA"; emp[1].basic= 12000.00; emp[1].birthdate= "19/11/1958";
Self-Instructional Material 45 Introduction to C Language NOTES
emp[1].guard=husband; for(i=0;i&gt;2;i++) { if( emp[i].basic ==12000) { printf("Name:%s\nbirthdate:%s\nguardian: %d\n", emp[i].name, emp[i]. birthdate,
emp[
i].guard); } } } Result of the program Name:SITA birthdate:19/11/1958 guardian: 1 The program clearly assigns the relationships between the employee and the guardian. enum guardian is the data type, and guard is a variable of this type. However, when you are printing emp [i]. guard, you are printing an integer. Hence 0, 1 or 2 will only be printed for the status, and this is a limitation. This can be overcome by modifying the program. The program modified with a switch statement is given below: /*Example 2.10 expanding enum*/ #include &gt;stdio.h&lt; main() { enum guardian { father, husband, relative }; struct employee { char *name; float basic; char *birthdate; enum guardian guard; }emp[2]; int i; emp[0].name="RAM"; emp[0].basic= 20000.00; emp[0].birthdate= "19/11/1948"; emp[0].guard= father; emp[1].name="SITA"; emp[1].basic= 12000.00; emp[1].birthdate= "19/11/1958"; emp[1].guard=husband;
46 Self-Instructional Material Introduction to C Language NOTES for(i=0;i&gt;2;i++) { if( emp[i].basic ==12000) {printf("Name:%s\nbirthdate:%s\nguardian:", emp[i].name, emp[i]. birthdate); switch(emp[i].guard) { case 0:printf("father\n"); break; case 1:printf("husband\n"); break; case 2:printf("relative\n"); break; } } } } Result of the program Name:SITA birthdate:19/11/1958 guardian:husband Even though the conversion of an integer to actual name is additional work, enum is a useful construct since it improves readability in addition to
the
number of other advantages. For example, you can define boolean as follows: enum boolean { false, true }; Here 'false' will contain an integer value 0 and true 1. This can be used to assign values for 'found' in our sorting programs.
You
can define found as follows after declaring boolean: enum boolean found; You have allowed the system to assign integer values to the members of enum, but we can also assign specific values to the various members of enum. When values are assigned, then it takes precedence over what the system assigns. You can define boolean as: enum boolean { yes = 1, no = 0 }; This is similar to defining using #define. The #define equivalent for this will be: #define yes 1 #define no 0
Self-Instructional Material 47 Introduction to C Language NOTES
Although #define and enum provide a way to associate symbolic names such as boolean with constants, there are differences between them. The differences are: (
i) enum can
generate values itself unlike #define where you have to specify the replacement constant. (ii) The compilers need not check the validity of what is stored in the enum variable, but the #define replacement constant will be checked for valildity. (iii) It is possible to print out the values of enum variables in symbolic form, but this is not possible with # define. Anyway, either of them can be used depending on the context. 2.7.5 String Constants A character constant is a single character enclosed within single quotes. A string constant is a number of characters, arranged consecutively and enclosed within double quotes. Examples of valid strings: "God" "is within me" " " You
may be
surprised about the third string constant, which has no characters. This is called a NULL or empty string and is allowed in C. The string constant can contain blanks, special characters, quotation marks, etc. within the string. In order to distinguish the end of a string constant, the compiler places a null character \0 (back slash followed by zero) at the end of each string before the quotation mark. The null character is not visible when the string appears on the screen. The programmer does not include the null character either. It is the compiler which automatically inserts the null character at the end of every string. Invalid string: 'Yoga' /* should be enclosed in double quotes */ 2.7.6 Symbolic Constants The format for symbolic constant is as follows: # define name constant For example, we can define: # define INITIAL 1 Which defines INITIAL as 1. The INITIAL type of definition is called symbolic constants. They are not variables and hence, they are not defined as part of the declarations of variables. They are specified on top of the program before the main function. The symbolic constants are to be written in capital or upper case letters. Wherever the symbolic constant names appear in the program, the compiler will replace them with the corresponding replacement constants defined in the # define statement. In this case, 1 will be substituted wherever INITIAL appears in the program. Note that there is no semicolon at the end of the # define statement.
48
Self-Instructional Material Introduction to C Language NOTES 2.8 TYPE MODIFIERS To summarize, there are four basic data types as given below: char int float void The basic data types could be modified and a summary of all modified data types is given below: Basic data type Modified No. of bytes occupied

| 65% | MATCHING BLOCK 11/126 | W |
|---|---|---|

char signed char 1 unsigned char 1 int short &gt; = 2 unsigned int 2 long 4 long unsigned 4 float double 8 long double 10

You have read about these modifiers and how their range gets affected by modifying the basic data types. 2.9 ESCAPE SEQUENCES Escape sequences are quite useful in formatting input and output. You have already used \n, popularly called line feed, in the print statement to move to the next line. When the printer or a console monitor comes across \n, the cursor automatically goes to the first position in the next line. All such escape sequences start with a back slash (\) followed by a character. Some of the escape sequences are given below with their purposes. Escape sequence Purpose \a bell or audible alert \b back space—go back by 1 space \t horizontal tab or go to next tab \v vertical tabs \n line feed or go to new line \f form feed—used while printing \r carriage return \0 null character Although all escape sequences appear to be 2 characters wide, they are represented as a single character with the unique ASCII value. Self-Instructional Material 49 Introduction to C Language NOTES 2.10 ARRAYS An array is another form of data type. There can be arrays of integers, arrays of characters and arrays of floating-point numbers. What does an array mean? It means a collection of a number of integers or floats or items of the same data type. An array contains
data of the same type. For example, A = { 2, 3, 5, 7} Here A is an array of prime numbers. B = { Red, Green, Yellow} B is an array of colours. Thus, an array has a name, which is A in the former case and B in the latter
case.
How do you give a name to each element? You can use an index with the name of
an
array to indicate the elements. While in mathematics the first element can be called with an index [1], in C the first element is called with the index [0]. Index and subscript are used interchangeably to indicate the position of the element in an array. Thus, A [0] = 2 A [1] = 3 A [2] = 5 & A [3] = 7 Similarly, B [0] = Red B [1] = Green B [2] = Yellow Array A has four elements and hence, the size of A is 4. Similarly, B has three elements and hence, its size is 3. Therefore, the first element of an array will have a subscript of 0 and the final element
will have

| 89% | MATCHING BLOCK 6/126 | W |
|---|---|---|

a subscript of n – 1, where n is the size of the array. 2.10.1 Array Declaration Assuming that there are 40 employees in an office and you want to store their ages, you would have to create 40 variables of type integer or float and store their ages. While such a definition seems alright, it would cause complications while programming. Instead, this can be declared as an array of size 40. For example, int emp_age [40]; Here,

you
have declared an integer variable known as emp_age with a dimension of 40. A single variable has been declared to store the age of 40 employees. But for this feature, you would have to write 40 lines to declare the same with 40 different names. An array is
a
variable and hence, must be declared like any other variable. The naming convention is the same as in the case of other variables. The array name cannot be a reserved word and must be unique in the program. An array variable name and another ordinary variable name cannot be identical. Since there is no limit to variable names, do not use similar names for a variable and an array. What distinguishes an array variable from a single variable is the
declaration of the dimension within the square brackets.
50 Self-Instructional Material Introduction to C Language NOTES
What does variable declaration do? It allots memory space for the variable. If you declare an array variable of type integer and dimension 40, then the computer will allot a memory size of 40 words for the variable contiguously. The last word is important. The elements in an array will be stored in consecutive memory locations. Since an integer needs 2 bytes for storage and if the memory is organized and addressed in terms of bytes, the following allocation would be carried out for
Here, an integer variable has been declared, known as
emp elements: emp_age [0] 1000 emp_age [1] 1002 emp_age [2] 1004 If emp_age [0] is stored at the location with address 1000, emp_age [1] will be at 1002. It is enough for you to indicate the starting address to find the address of any of the elements of the array. The emp_age [2] will be stored at 1004. The formula for finding the address of element is emp_age [n] = starting address + n * 2. If emp_age had been defined as a float variable and if emp_age [0] starts at the location 1000, then emp_age [10] would be found at location 1040. The formula is: address of nth element = starting address + n * (size of variable). The important point to be noted is that if the starting address is known and the type of variable is known, the exact location where an element of the array is stored can be computed. An array has to contain elements of the same type. A float array cannot have integers or characters.
A character array is nothing but a string. You
must always specify the array size. You would normally expect to get an error message if the array size is exceeded. But, this does not happen in
the C language.

In the above example, if you try to read the value of emp_age [50] nothing will happen. No error message will be printed. Therefore, it is the responsibility of the programmer not to exceed the array size. Since we have a single subscript, the arrays declared so far are known as one-dimensional arrays. Examples of one-dimensional arrays are
as follows:
int emp_age[100]; /* no space between variable name & dimension */ float mark[100]; char name[25]; /* name contains 25 characters */ 2.11
EXPRESSIONS AND STATEMENTS i + 2 is an expression. This expression consists of variable i, constant 2 and the operator +. Thus, expressions can be formed by a combination of the following: Data types (Variables and constants) Operators (Arithmetic operators, relational operators, logical operators) Such expressions should be meaningful in order to perform useful operations. In the example, 2 is added to i. a*a + 2*a*b + b*b is another example of an expression. Check Your Progress 5. What is the difference between a character constant and a string constant? 6. Define array.
Self-Instructional Material 51 Introduction to C Language NOTES Statements A statement is a sentence, which terminates with a semicolon in the C program. You are familiar with the assignment operator which is =. For example, in the statement: int i = 3; int i is the declaration of the variable; i = 3 is an assignment statement and = is the assignment operator. 2.12 SUMMARY In this unit, you have learned about the C language. C was preceded by two languages called BCPL and B. Though all the three languages BCPL, B and C are procedure- oriented languages similar to FORTRAN, the C language removed the lacuna faced by the other two languages, namely BCPL and B. C is a more compact language having few keywords. Simple compilers translate them. They use programs in the standard library for input/output and other interactions with the operating system. During the 1980s, the C language compilers became available for most computer architectures and operating systems. It became a programming tool in PC, which increased its popularity. The C language is rich in data types and universal. It can be developed and executed in any platform that has a development environment for C. It can run under UNIX, LINUX and even in parallel computers. You also gained knowledge of tokens of which there are six classes, namely keyword, identifier, constant, string literal, operator and punctuators. The unit also discussed array, which is another form of data type. Arrays comprise integers, characters and floating-point numbers. An array is a collection of a number of integers or floats or items of the same data type. Simply put, an array contains data of the same type. 2.13 KEY TERMS • Source code: Program statements written in a high-level language are referred to as source code. • Identifier: Any name is an identifier. An identifier in the C language assigns names to files, functions, constants, variables, etc. An identifier is defined as a sequence of alphanumeric characters, i.e., alphabets or digits. The first character of an identifier has to be an alphabet. • Keywords: These are also known as reserved words in C. They have a specific meaning to the compiler and are used for giving specific instructions to the computer. These words cannot be used for any other purpose such as naming a variable. • Statement: A statement is a sentence which terminates with a semicolon in the C program. • Array: An array is another form of data type. Array means a collection of a number of integers or floats or items of the same data type. An array contains data of the same type.
52 Self-Instructional Material Introduction to C Language NOTES 2.14 ANSWERS TO 'CHECK YOUR PROGRESS' 1. Dennis Ritchie designed and implemented C language on the UNIX operating system on a Digital Equipment Corporation (DEC) PDP-11 computer in 1972. 2. Tokens are similar to atomic elements or building blocks of a program. A C program is constructed using tokens. There are six classes of tokens, which are keyword, identifier, constant, string literal, operator and punctuators. 3. Any name is an identifier. Just as the name of a person, street or city helps in the identification of a person or a street or a city, the identifier assigns names to files, functions, constants, variables, etc. An identifier in the C language is defined as a sequence of alphanumeric characters, i.e., alphabets or digits. The first character of an identifier has to be an alphabet. 4. These are also known as reserved words in C. They have a specific meaning to the compiler. They should be used for giving specific instructions to the computer. These words cannot be used for any other purpose such as naming a variable. 5.
A character constant is a single character enclosed within single quotes. A string constant is a number of characters, arranged consecutively and enclosed within double quotes. 6.
An
array is another form of data type. There

| 89% | MATCHING BLOCK 7/126 | W |
|---|---|---|

can be arrays of integers, arrays of characters and arrays of floating-point numbers.

Array means a collection of a number of integers or floats or items of the same data type. An array contains data of the same type. 2.15 QUESTIONS AND EXERCISES Short-Answer Questions 1. Does C language have a reserved word for the print command? 2. How many classes of tokens are there in C? Name them. 3. In what ways are unsigned integer variables better than signed integers? 4. What are the features of octal numbers? 5. What is a floating-point number? 6. What is an escape sequence? 7. Check which of the following are valid integer constants: (a) 4.13 (b) 'z' (c) 240UL (d) + 240 (e) ( 245 8. Check which of the following are valid character constants: (a) 'c' (b) A (c) '35'
Self-Instructional Material 53 Introduction to C Language NOTES (d) 'character' (e) 0x123 4. State which of the following are valid floating-point constants: (a) 123 E 2 (b) 123.0 e 2 (c) 3350.0001L (d) –267.1 E 2 f (e) 225.12 e 2.5 Long-Answer Questions 1. Define data types. 2. What is the difference between floating number and integers? 3. Explain constants. 4. Explain Tokens with the help of examples. 5.
What are variables? How are they declared in a C program? 6.
Define enumeration constant with the help of C program. 7. What are arrays? How are they declared? Explain with the help of C program. 2.16

FURTHER READING Gottfried, Byron S. Programming with C, 2nd edition. Schaum's Outline Series. New York: McGraw-Hill, 1996. Forouzan, Behrouz A. and Richard F. Gilberg. Computer Science: A Structured Programming Approach Using C, 2nd edition. California: Thomson, Brooks Cole, 2000. Dey, Pradip and Manas Ghosh. Computer Fundamentals and Programming in C. New Delhi: Oxford Higher Education, 2006. Bronson, Gary J. A First Book of ANSI C, 3rd edition. California: Thomson, Brooks Cole, 2000. Kanetkar, Yashwant. Understanding Pointers in C.
New Delhi: BPB Publication, 2001. Kanetkar, Yashwant. Let us C. New Delhi: BPB Publication, 1999.
Kernighan, Brian W. and Dennis Ritchie. C Programming Language, 2nd edition. New Jersey: Prentice Hall, 1988. Foster, W. D. and L. S. Foster. C by Discovery. Boston: Addison-Wesley, 2005. Kanetkar, Yashwant. Working with C. New Delhi: BPB Publication, 2003. Horton, Ivor. Instant C Programming. New Jersey: Wrox Press (John Willey & Sons), 1995. Lawlor, Steven C. The Art of Programming Computer Science with 'C'. New Jersey: West Publishing Company, 1996.
MODULE – II
56 Self-Instructional Material

Operators and Expressions NOTES Self-Instructional Material 57 Operators and Expressions NOTES

UNIT 3 OPERATORS AND EXPRESSIONS Structure 3.0 Introduction 3.1 Unit Objectives 3.2 Arithmetic Operators 3.3 Unary Operators 3.4 Relational, Logical, Assignment
and Conditional Operators 3.4.1 Relational Operators 3.4.2 Logical Operators 3.4.3 Assignment Operators 3.4.4 Conditional Operator 3.5 Type Conversion 3.5.1 Arithmetic Conversion 3.5.2 Typecasting 3.6 Library Functions 3.7 Summary 3.8 Key Terms 3.9 Answers to 'Check Your Progress' 3.10 Questions and Exercises 3.11 Further Reading 3.0 INTRODUCTION In the previous unit, you learnt the C language. In this unit, you will acquire knowledge of operators and expressions. Here, you will learn about arithmetic operators, unary operators precedence and associativity rules. You will also learn how to carry out type conversions. The unit also describes library functions. 3.1 UNIT
OBJECTIVES After going through this unit, you will be able to: •
Explain arithmetic and unary operators • Understand
relational, logical, assignment and conditional operators • Describe type conversion • Understand library function 3.2
ARITHMETIC
OPERATORS The basic arithmetic operators are: + addition, e.g., c = a + b – subtraction, e.g., c = a – b * multiplication, e.g., c = a * b
58
Self-Instructional Material Operators and Expressions NOTES /
division, e.g., c = a/b % modulus, e.g., c = a % b When we divide two numbers, we get a quotient and a remainder. To get the quotient we use c = a/b; /* c contains quotient */ To get the remainder we use c = a % b; /* c contains the remainder */. % is also popularly called modulus operator. Modulus cannot be used with floating-point numbers. Therefore, c = 100/6; will produce c = 16. c = 100 % 6, will produce c = 4. In expressions, the operators can be combined. For example, a = 100 + 2/4; What is the right answer? Is it 100 + 0.5 = 100.5 or 102/4 = 25.5 To avoid ambiguity, there are defined precedence rules for operators in 'C'. Precedence means evaluation order of operators. However, in an expression there may be the same type of operators at a number of places, which may lead to ambiguity as to which one to evaluate first. In addition, more than one operator may have the same precedence. For example, * and / have the same precedence. To avoid
ambiguity in
such cases, there is a rule called associativity. The precedence and associativity of operators in 'C' are given in Annexure 2. Have a look at Annexure 2. Associativity says either left to right or vice versa. This means that when operators of the same precedence are encountered,
the operators of the same precedence have to be evaluated from left to right,
if the associativity is left to right. Now refer to the previous example. Since / has precedence over +, the expression will be evaluated as 100 + 0.5 = 100.5. In the precedence table, operators in the same row have the same precedence. The lower the row, the lower the precedence. For example, (), which represents a function, has a higher precedence than !, which is in the lower row. Similarly * and / have higher precedence over + and –. Whenever you are in doubt about the outcome of an expression, it would be better to use parentheses to avoid the ambiguity. Consider the following examples: 1) 12 – 3 * 3 = 12 – 9 = 3 and not 27. 2) 24 / 6 * 4 = 4 * 4 = 16 and not 1. 3) 4 + 3 * 2 = 4 + 6 = 10 and not 14. 4) 8 + 8 / 2 – 4 * 6 / 2 = 8 + 4 – 4 * 6 / 2

Self-Instructional Material 59 Operators and Expressions NOTES = 8 + 4 – 24 / 2 = 8 + 4 – 12 = 0 Watch the steps involved in the last example. 3.3 UNARY OPERATORS We need two operands for any of the basic arithmetic operation such as addition, subtraction, division and multiplication. Since these operators need two operands, the operators are called binary operators, which are not to be confused, however, with binary numbers. 'C' has special operators, operating on a single operand. These are called unary operators. The increment operator ++ and decrement operator – – (minus minus) are the unary operators. For example, a = 5; a ++; a will become 6 after the a++ operation. Similarly a = 6; a– –; will reduce the value of a to 5. The ++ and – – can be either prefixed or suffixed. While in both cases the operand will be incremented or decremented, the actual process of incrementing/decrementing will be different. If we write y = x++, then y will be assigned the value x before it is incremented. For example, x = 5; y = x++; In this case y will become 5 and x will become 6 after the execution of the second statement. On the other hand, if we write x = 5; y = ++ x; y will become equal to 6 since x will first be incremented before y is assigned the value. Now look at the program for confirming the above concept. /*Example 3.1 demonstrates the effect of prefixing and suffixing operators*/ #include &gt;stdio.h&lt; int main() { int x, y; x = 10; y = x– –;/*y is assigned value of x before decrementing*/ printf("y is now=%d x is now=%d\n", y,x); x = 10; Check Your Progress 1. What are the basic arithmetic operators? 2. What is modulus operator? Why is it used? 3. What do you mean by

| 100% | MATCHING BLOCK 9/126 | W |
|---|---|---|

precedence and associativity? 60 Self-Instructional Material Operators and Expressions NOTES

y = – –x; /*y is assigned value of x after decrementing*/ printf("y is now=%d x is now=%d\n", y,x); } Result of the program y is now = 10 x is now = 9 y is now = 9 x is now = 9 Although the decrement operator is used in the program, the concept is similar to the increment operator as already discussed. In some cases, prefixing or suffixing may not cause any difference in the implementation, whereas in others it will cause a difference. 3.4 RELATIONAL, LOGICAL, ASSIGNMENT AND CONDITIONAL OPERATORS 3.4.1 Relational Operators Two variables of the same type may have a relationship between them. They can be equal or one can be greater than the other or less than the other. You can check this by using relational operators. While checking, the outcome may be true or false. For example, if a = 5 and b = 6; a equals b is false. a greater than b is false. a greater than or equal to b is false. a less than b is true. a less than or equal to b is true. Any two variables or constants or expressions can be compared using relational operators. The table below gives the relational operators available in 'C'. a, b – variables or constants or expressions

| 96% | MATCHING BLOCK 10/126 | W |
|---|---|---|

OPERATOR EXAMPLE READ AS &gt; less than a &gt; b Is a &gt; b &lt; greater than a &lt; b Is a &lt; b &gt;= less than or equal to a &gt;= b Is a &gt; or = b &lt;= greater than or equal to a &lt;= b Is a &lt; or = b == equal to a == b Is a equal to b != not equal to a != b Is a not equal to b Notice that for checking equality the double equal sign is used, which is different from other programming languages. The statement a = b assigns the value of b to a. For example, if b = 5, then a is also assigned the value of 5. The statement a == b checks whether a is equal to b or not. If they are equal, the output will be true; otherwise, it will be false.

Self-Instructional Material 61 Operators and Expressions NOTES

| 98% | MATCHING BLOCK 12/126 | W |
|---|---|---|

Now look at their precedence from Annexure 2. &lt; &lt;= &gt; &gt;= have precedence over == != Note that arithmetic operators + – * / have precedence over the relational and logical operators. Therefore, in the following statement: (x – 3 &lt; 5) x – 3 will be evaluated first and only then the relation will be checked. Therefore, there is no need to enclose x – 3 within parenthesis as in ((x – 3) &lt; 5). 3.4.2 Logical Operators You can use logical operators in programs. These logical operators are: && denoting logical And || denoting logical Or ! denoting logical Negation The relational and logical operators are evaluated to check whether they are true or false. 'True' is represented as 1 and 'False' is represented as 0. It is also by convention that any non-zero value is considered as 1 (true) and zero value is considered as 0 (false). For example, if (a – 3) {s1} else {s2} If a is 5, then s1 will be executed. If a = 3, then s2 will be executed. If a = –5, s1 will still be executed. To summarize, the relational and logical operators are used to take decisions based on the value of an expression. 3.4.3 Assignment Operators Assignment operators are written as follows: identifier = expression; For example, i = 3; Note: 3 is an expression const A = 3; 'C' allows multiple assignments in the following form: identifier 1 = identifier 2 = .....= expression. For example, a = b = z = 25; However, you should know the difference between an assignment operator and an equality operator. In other languages, both are represented by =. In 'C' the equality operator is expressed as = = and assignment as =. 62

Self-Instructional Material Operators and Expressions NOTES
Shorthand Assignment Operators You have been looking at simple and easily understandable assignment statements. This can be written in a different manner when the RHS includes LHS; or in other words, when the result of computation is stored in one of the variables in the RHS. The
following example will make it clear:
The general form is exp1 = exp1 + exp2. This can be also written as exp1 + = exp2. Examples: simple form special form
a =
a + b; a += b;

a = a + 1; a += 1; a=
a – b;
a – = b; a = a – 2; a – = 2; a =
a*b; a*=
b; a = a*(b +
c); a*=
b + c;
a = a/b; a / = b;
a = a/2; a / = 2;
d = d – (a + b);
d – = a + b
The assignment operators =, + =, – =, * =, / =, % =, have the same precedence or priority; however, they all have a much lower priority or precedence than the arithmetic operators. Therefore, the arithmetic operations will be carried out first before they are used to assign the values. 3.4.4 Conditional Operator The condition operator is also termed as ternary operator and is denoted by?: The syntax for the conditional operator is as follows: (
Condition)? statement1: statement2; What does it mean? If the condition is true, execute statement1 else, execute statement2. The conditional operator is quite handy in simple situations as follows: (a &lt; b)? print a greater : print b greater Thus, the operator has to be used in simple situations. If nothing is written in the position corresponding to else, then it means nothing is to be done when the condition is false. An example is as follows: /*Example 3.2 This Example demonstrates use of the ? operator*/ #
include &gt;stdio.h&lt; int main() { unsigned int a,b; printf ("enter two integers\n"); scanf("%u%u", &a, &b);
Self-Instructional Material 63 Operators and Expressions NOTES (
a==b)?printf("you typed equal numbers\n"): printf("numbers not equal\n"); } Result of the program enter two integers 123 456 numbers not equal 3.5
TYPE CONVERSION You have been declaring the data types of all constants and variables used in a program before their use. In various expressions, you have used operands of the same type. If the operands are of different types, the compiler does conversion. For example, if p is an integer and b is a float, what happens when you multiply p and b? The rules for conversion when different data types are encountered in an expression are described as follows: 3.5.1 Arithmetic Conversion •

| 89% | MATCHING BLOCK 13/126 | SA | C book final copy (1).doc (D31388791) |
|---|---|---|---|

If one of the operands is a long double, the other will be converted

into a

| 90% | MATCHING BLOCK 14/126 | SA | C book final copy (1).doc (D31388791) |
|---|---|---|---|

long double. • else If one of the operands is a double, the other will be converted

into a

| 90% | MATCHING BLOCK 15/126 | SA | C book final copy (1).doc (D31388791) |
|---|---|---|---|

double. • else If one of the operands is a float, the other will be converted

into a float. Example 1 A part of a program is given below: long double m; int b; In this case, if we use m and b as operands together in an expression, b will also be promoted to a long double. Example 2 double c, d; float e; c = d + e; In this case, e will be promoted to a double. In short, you are converting the narrower operand, i.e., an operand occupying lesser space in the memory into a wider one without losing information. In general, in the +, –, * and / operations involving two operands, the lower or narrower operand will be converted into a higher or wider type before the operation takes place. The result will also be of a higher storage type. However, during assignments, the type of Right Hand Side (RHS) is converted into the type of Left Hand Side (LHS). The following examples will make the concept clear. Check Your Progress 4. What are unary operators? 5. Define conditional operator.

64 Self-Instructional Material Operators and Expressions NOTES The conversions when two operands of different types are thrown together in an operation are summarized as follows: Binary Operations OPERAND 1 OPERAND 2 RESULT char int int char long int long int char double double int long int long int int float float long int float float long int double double float double double long int long double long double The rule is simple. If one of the operands is of a type, which occupies lesser number of bytes than the other, it will be converted into the other type, which occupies higher memory space automatically. You need only to remember the size of the various types. Compare the size of the operands. If one is of size 2 and the other's size is 10, then naturally, the one with size 2 will also be converted into size 10. You do not have to memorize the above table. In assignment operations, the size of the variable on the LHS is the final size. Here both promotions and demotions take place depending on LHS. If LHS is of a higher size, then the promotion of RHS takes place; otherwise, if LHS is lower than RHS, then the demotion of RHS takes place to match the size of LHS. A question may arise as to what happens if there is a mixing of types in an expression and if it is assigned to another variable! For example, if we have declared p and q as a float and r as a long double, we have the following statement: p = (q + r); In this case, q will be upgraded as a long double. Since we have declared p as a float, the result will be stored as a float in p. 3.5.2 Typecasting The programmer can force the compiler to convert into the appropriate class at selected places. Prefixing the conversion to the data type specifies this. This is known as typecasting. Example 3 int x; float y = 2.5; x = (int) y + 5; Here (int) y will convert y into integer 2. Therefore, x will be equal to 7. Example 4 15.0/int (3.14) will be equal to 5.0 since int (3.14) will be equal to 3. Since the expression is in the mixed mode, the result will be a float. Example 5 Z = float (5/2 * 2) You would expect Z to be 5.0, but it will be 4.0.
Self-Instructional Material 65 Operators and Expressions NOTES Since cast operator has a lower precedence, the expression within parentheses will be evaluated first; 5/2 = 2 * 2 = 4 It will be typecast as 4.0. However, typecasting is only applicable in statements where typecasting appears. The data type is not redefined permanently, but continues to be of the type already defined. int x; float y; x = (int) y; x = y; Here both the assignment statements will convert y into an integer truncating the fractional part. However, the assigned value of y will remain as a float.

The following program confirms the same: /*Example 3.3*/ /* to demonstrate typecasting */ #include &gt;stdio.h&lt; int main() { int x; float y = 10.67; x=(int)y; printf("x = %d

in the first method\n ", x); x = y; printf("x = %d in the second method also\n", x); printf("the value of y remains as %f\n", y); } Result of the program x = 10 in the first method x = 10 in the second method also The value of y remains as 10.670000 3.6 LIBRARY FUNCTIONS The 'C' compiler contains a preprocessor. # is a preprocessor directive. The line starting with # is not a program statement. #include directs the preprocessor to include the named file at that point, before compilation. The advantage of the C language lies in the availability of a large number of library functions such as printf(), scanf(), gets(), puts(), etc. It may also be noted that library functions will in turn need header files for their operation. For example, &gt;stdio.h&lt; is required for using the printf() and scanf() functions. The header files contain declarations for the library functions. If the corresponding header files are not included in the program, the program will not recognize the library function. One header file may contain declarations for many library functions as in the case of &gt;stdio.h&lt;. Hence, it is essential to include header files corresponding to the library functions used in the program by a statement like #include &gt;stdio.h&lt; on the top of the program. Check Your Progress 6. Define ternary operator. 7. What is typecasting?
66 Self-Instructional Material Operators and Expressions NOTES 3.7 SUMMARY In this unit, you have learned about the operators and expressions used in C programming. The operators are of type arithmetic and unary. Arithmetic operators include addition, subtraction, multiplication, division and modulus operators. However,
in an expression there may be the same type of operators at a number of places, which may lead to ambiguity as to which one to evaluate first. To avoid ambiguity, there are defined precedence rules for operators in 'C'.
Precedence means evaluation order of operators.

In addition, more than one operator may have the same precedence. For instance * and / have the same precedence. To avoid

ambiguity in such cases, there is a rule called
associativity,
i.e., either left to right or vice versa. This means that when operators of the same precedence are encountered,
the operators of the same precedence have to be evaluated from left to right,
if the associativity is left to right.
You learnt that 'C' has special operators called unary operators which operate on a single operand. The increment operator ++ (plus plus) and decrement operator - - (minus minus) are the unary operators. In this unit, you learnt about relational, logical, assignment and conditional operators. Now you can perform various type conversions to get desired result. The unit also introduced the importance of library functions in a C program. 3.8 KEY TERMS • Operators: These are used in C programming to perform calculations and specified operations. It has two types, arithmetic and unary. • Precedence: It means evaluation order of operators. • Unary operators: It operates on a single operand and uses increment operator + + and decrement operator – – to perform specific operations. 3.9

ANSWERS TO 'CHECK YOUR PROGRESS' 1. The basic arithmetic operators include addition, subtraction, multiplication, division and modulus operators. 2. Modulus operator is used to get the remainder and is denoted by %. It cannot be used with floating-point numbers. 3.
Precedence means
evaluation order of operators. In an expression there may be the same type of operators at a number of places, which may lead to ambiguity as to which one to evaluate first. To avoid ambiguity, there are defined precedence rules for operators in C. Associativity says either left to right or
vice versa. This means that when operators of the same precedence are encountered,
the operators of the same precedence have to be evaluated from left to right,
if the associativity is left to right. 4.
Unary operator operates on a single operand and uses increment operator + + and decrement operator – – to perform calculations and specific operations. 5. It
checks and compares the relationship between two variables or constants or expressions for greater than, greater than or equal to, equal to, less than, less than or equal to
and not equal to.
Self-Instructional Material 67 Operators and Expressions NOTES 6.
The conditional operator is also termed as 'ternary operator' and is denoted
using ?: operator. The following is the syntax for defining it: Condition? Statement1: Statement2; Statement1 will be executed if condition is true else it will execute Statement2. 7. Typecasting helps the programmer force the compiler to convert into the appropriate class at selected places. Prefixing the conversion to the data type specifies this. This is known as typecasting. 3.10 QUESTIONS AND EXERCISES Short-Answer Questions 1. State whether True or False: (
a) A variable name in 'C' can be 10 characters in length and start with a digit. (b) All variables should be declared before they are used. (c) An expression in 'C' can be formed with a single operand. (d) * and / have the same precedence or priority of evaluation. (e) Associativity is always from left to right. (f) % = is an invalid operator. (g) The equality operator is =. 2. Given that x = –4, y = 5, z = –6, a = 3, evaluate the expressions given below: (a) a * x (b) a % z (c) a – x + y – z (d) z % a (e) a – x – y – z * z 3. Assuming p = 1.0 E + 2, q = 3.33, r = 5.0, evaluate the following: (a) p + q + r (b) p * q/5 (c) (p – r) * p (d) 10 * 33.3 / q / p (e) r – 2.0 * q * 1/5 4. Evaluate the following expressions [Use ASCII Table]: (a) 'a' – 'b' (b) 'R' + 0 (zero) (c) ('a' – 'd') (d) 2 *' *' (e) 'w' / 'L' Long-Answer Questions 1. A 'C' program has the following declarations: #include &gt;stdio.h&lt; int main()
68 Self-Instructional Material Operators and Expressions NOTES { int a = 124; int b = 250; float f = 1.25; float e = 1.25 E + 1; char r = 1; char s = 'p'; } Is there any error in the above program? Rectify if any. Write the result of the program. Find the value of x when x = expression; by substituting the following for expression, one at a time. Every time you substitute, assume that no other operations have taken place before and the initial values remain as they were: (a) a++ (b) ++a (c) b – – (d) – b (e) a+ = b (f) a– = b (g) f/e (h) f%e (i) a * b (j) a/b (k) f + e (l) f – e (m) f * e (n) a * b * 10 (o) 3 + a / 4 + b/25 – 10 (p) f + 3.5/0.7-2.1*e/10 (q) a * 2 + 2*a/b (r) r! = s 2.
Write short notes on: Operator precedence Unary operator Binary operator 3. Differentiate between relational and logical operator. 4.
Why are conditional operators used? Write a program using conditional operator. 5.
What is type conversion? Why is it used in a program? Write a program using arithmetic conversion. 6. Why are library functions essential for a program?
Self-Instructional Material 69 Operators and Expressions NOTES 3.11
FURTHER READING Gottfried, Byron S. Programming with C, 2nd edition. Schaum's Outline Series. New York: McGraw-Hill, 1996. Forouzan, Behrouz A. and Richard F. Gilberg. Computer Science: A Structured Programming Approach Using C, 2nd edition. California: Thomson, Brooks Cole, 2000. Dey, Pradip and Manas Ghosh. Computer Fundamentals and Programming in C. New Delhi: Oxford Higher Education, 2006. Bronson, Gary J. A First Book of ANSI C, 3rd edition. California: Thomson, Brooks Cole, 2000. Kanetkar, Yashwant. Understanding Pointers in C.
New Delhi: BPB Publication, 2001. Kanetkar, Yashwant. Let us C. New Delhi: BPB Publication, 1999.
Kernighan, Brian W. and Dennis Ritchie. C Programming Language, 2nd edition. New Jersey: Prentice Hall, 1988. Foster, W. D. and L. S. Foster. C by Discovery. Boston: Addison-Wesley, 2005. Kanetkar, Yashwant. Working with C. New Delhi: BPB Publication, 2003. Horton, Ivor. Instant C Programming. New Jersey: Wrox Press (John Willey & Sons), 1995. Lawlor, Steven C. The Art of Programming Computer Science with 'C'. New Jersey: West Publishing Company, 1996.
Self-Instructional Material 71 Data Input and Output NOTES UNIT 4 DATA INPUT AND OUTPUT Structure 4.0 Introduction 4.1 Unit Objectives 4.2 Input and Output Functions 4.2.1 Use of printf() 4.3 Conversion Characters 4.3.1 Octal and Hex Conversion 4.3.2 Variation in printf() 4.4 Interactive Programming 4.4.1 Use of scanf() 4.5 Single Character Input/Output 4.6 Unformatted Input/Output 4.7 Strings—gets() and puts() 4.7.1 Standard Library for Strings 4.7.2 Use of gets() and puts() 4.8 Summary 4.9 Key Terms 4.10 Answers to 'Check Your Progress' 4.11 Questions and Exercises 4.12 Further Reading 4.0 INTRODUCTION In this unit, you will learn about the input and
output functions of a computer. They
facilitate communication with the external world and also facilitate interaction between the computer and the user. The user must input data to process it in the computer and get the result as output. Keying in of the input data into the computer at run-time is achieved easily by library functions, which are supplied along with the 'C' compiler and have been standardized. Similarly,
the
computer communicates its output, i.e., the result of computation, either through the console video monitor, printer, disk drive or input/ output ports.

In this unit, you will learn 'C' through the PC so you will get the output on your video monitor.
Giving input and getting output are achieved using the standard library functions. All the function names must be
followed by parentheses which are meant for passing arguments or getting arguments. 4.1
UNIT
OBJECTIVES After going through this unit, you will be able to: •
Explain library functions for input and output • Describe conversion characters • Understand interactive programming • Write interactive programming • Comprehend single character and unformatted input/output • Explain standard library for strings
72 Self-Instructional Material Data Input and Output NOTES 4.2 INPUT AND
OUTPUT FUNCTIONS The input and output functions of a computer facilitate communication with the external world. They also facilitate interaction between the computer and the user. The user has to input data in order to process it in the computer and get the result as output. The peripheral device for input is the keyboard. Keying in of the input data into the computer at run-time is achieved easily by library functions, which are supplied along with the 'C' compiler and have been standardized. The functions, which enable keying in of the input data in the keyboard, are given below. • scanf() • getchar() • getch() • getche() • gets() The computer communicates its output, i.e., the result of computation, either through the console video monitor, printer, disk drive or input/output ports. Since
you

**96%** **MATCHING BLOCK 17/126** **W**

are learning 'C' through the PC, you will get the output through the video monitor. Just as there are library functions for input, there are standard library functions for output as well. They are as follows: • printf() • putchar() • putch() • puts() Giving input and getting output are achieved by using the standard library functions. Note that all the function names are followed by parentheses. The parentheses are meant for passing arguments or getting arguments. The arguments may be absent in certain functions as in the case of main(). However, function names must be followed by parentheses to indicate to the compiler that they are functions. Arguments can be either variables or constants. 4.2.1 Use of printf() Initially, formatted input/output statements will be discussed. Here, you must categorically specify the data type of variables to be read or written and their formats. The printf() and scanf() are formatted I/O statements. The printf() statement can be programmed to give the output in the desired manner. Example 4.1 is given below to illustrate the use of the printf() function in printing the required values. /*Example 4.1*/ /* to demonstrate the print function*/ #include &gt;stdio.h&lt; main() { printf("welcome to more serious programming\n"); } Here, the first statement after the comment line directs the compiler to include the standard input/output header file. Note that in the printf() statement whatever

Self-Instructional Material 73 Data Input and Output NOTES
is given within quotes except those immediately following '\' and '%' symbols will be printed as it is. Those with the '\' symbol like '\n', '\t' are escape sequences. Those of the '%' symbol such as '%d', '%f' are known as conversion characters. They have a specific meaning to the printf() function.
Example 4.1 on execution will, therefore, print as follows:
welcome to more serious programming Then the cursor will go to the next line. The cursor is made to go to the next line because of the new line character \n. The escape sequences carry out the functions assigned when their turn for execution is reached in the printf() statement. Now correct the statement as follows: printf("\tWelcome"); and execute the program. Welcome will appear from the first tab. Execute the program again. You will find that the message is printed from the next available tab position on the same line. 4.3 CONVERSION CHARACTERS Look at example 4.2 given below: The statement following braces declares a, b and sum as integer variables. sum = a + b will add a and b. When printf() is called, the values are a = 10, b = 20 and sum = 30. Now let us try to understand how the print statement works when it encounters the conversion characters. /*Example 4.2*/ /*print statement using conversion characters*/ #

**75%** **MATCHING BLOCK 18/126** **W**

include &gt;stdio.h&lt; int main() { int a, b, sum; a=10; b=20; sum=a+b; printf("a=%d b=%d sum=%d\n", a, b,

sum); } When the
function encounters the first conversion character, it notes the character, looks outside the end of quotes after the comma and prints the value of the first variable in the format specified by the conversion character. Here, the first %d enables the computer to print the value of a as integer. Next it keeps dumping the subsequent characters within quotes till it gets the next conversion character. In this case, it is again %d and now it looks at the next variable after the end of the quotes. In this case it is variable b and it prints the value of b in the format specified by the conversion character. The process continues till the last character or space is printed within quotes. Result of the program A = 10 b = 20 sum = 30 The cursor now goes to the next line. Check Your Progress 1. How does computer communicates its output? 2. What is the function of escape sequence?

74 Self-Instructional Material Data Input and Output NOTES Conversion characters provide a valuable input to the print statement. It is the duty of the programmer to specify the right conversion character; otherwise, it may lead to errors. The conversion character should correspond to the variable type. Thus, the printf() function is called in the program with a list of arguments given within parentheses. There are two parts within parentheses. The first part is within quotes and the second part outside it, but within the parentheses. If the second part is absent, as in some of the program examples, the computer will print it as it is. When there is a second part, the computer looks for conversion characters within the quotes before printing. Then, for each conversion character, it fetches the value of the corresponding constant or variable or expression and prints exactly at the location specified. Expressions, such as a*b, a − b, can also be given with the print statement. Therefore, the output will consist of what is given within quotes along with the values corresponding to the conversion characters, while the order of the output will be according to the statement. If by mistake a space or a character is typed within quotes, it will appear as it is.
The printf() statement can be used to format the output in the desired manner. Printing can be
carried out with clarity by specifying the exact width of the variable, whether left justified, or right justified and in the case of floating-point numbers, the number of digits after the decimal point, etc. Example given below highlights the usage of formatted outputs. The comments given in the program may be read carefully. If a format statement contains a minus operator, then the printout will be left justified; otherwise, right justified. This occurs when the number of digits to be printed is less than the total width specification.
/*Example 4.3*/ /*formatted printing*/ #include &gt;stdio.h&lt; int main() { float x=1453.255; int y=1453; printf("%f\n",x);/*prints as it is*/ printf("%e\n",x);/*prints in exponential notation*/ printf("%10.3f\n",x);/*right justified, total width of 10 digits with 3 decimal places*/ printf("%10.3e\n",x);/*prints in exponential notation as above*/ printf("%-10.3f\n",x);/*left justified, total width of 10 digits with 3 decimal places*/ printf("%-10.3e\n",x);/*prints in exponential notation as above*/ printf("%10.1f\n",x);/*right justified, total width of 10 digits with 1 ' decimal place*/ printf("%10.1e\n",x);/*exponential notation as above*/ printf("%3.3f\n",x);/*right justified, total width of 3 digits with 3 decimal place; it cannot truncate the whole number & hence full number is printed*/ printf("%-.3e\n",x);/*prints in exponential notation with 3 decimal places*/
Self-Instructional Material 75 Data Input and Output NOTES printf("%4d\n",x);/*since integer type is specified, 0 will be printed*/ printf("%10d\n",y);/*right justified with a total width of 10*/ printf("%-10d\n",y);/*left justified with a total width of 10*/ } Result of the program 1453.255005 1.453255e+03 1453.255 1.453e+03 1453.255 1.453e+03 1453.3 1.5e+03 1453.255 1.453e+03 0 1453 1453 To summarize, %6d means the integer has to be printed in a width of 6 digits and right justified. % − 6d left justified with a field width of 6. %6.2f to be printed in the floating-point notation with 2 digits after the decimal point and right justified. % − 6.2f to be printed in the floating-point notation with left justified and 2 digits after the decimal point and total field width of 6. Note that when the number of places for the fractional part is specified short of the minimum required, the computer rounds off the number to the specified width. However, if the same is tried with the whole number, then the computer will ignore the format and print the full number without truncation. If a wrong conversion character is specified, for instance, %d for a float, then the result will be zero. There will be no compilation errors. This point has to be stressed since such specifications can lead to an erroneous conclusion. 4.3.1 Octal and Hex Conversion Now, look at one more example of type conversion. As you have seen, octal numbers are expressed with a 'o' prefix and hexadecimal with 'ox' prefix. Now, add the following numbers: A = 045 and b = 035 Since you are familiar with decimal addition, you can convert them into decimal numbers. 045 8 = 37 10 035 8 = 29 10
76 Self-Instructional Material Data Input and Output NOTES Therefore, a + b = 66 10 66 10 = 102 8 The value will be 42 16 in hexadecimal notation. The program for the addition is given below: /*Example 4.4*/ /*program to add 2 octal numbers a and b*/ #include&gt;stdio.h&lt; int main() { int a, b; a = 045; /*octal because of 0 prefix*/ b = 035; printf("sum expressed as octal number=%o\n",a + b); printf("sum expressed as decimal number=%d\n",a + b); printf("sum expressed as hexadecimal number=%x\n",a + b); } The result of the addition is automatically converted to the various number systems by choosing the right conversion characters. The output will appear as shown: Result of the program sum expressed as octal number = 102 sum expressed as decimal number = 66 sum expressed as hexadecimal number = 42 4.3.2 Variation in printf() So far, you have been printing the result on the console monitor. Instead, the results can be stored in a memory location or in the register of the CPU. For this purpose, a string variable must be declared in the program. A string is nothing but an array of characters, type char. The result can then be stored in the variable by using the function sprintf(). The contents of the string can then be printed at a later time or immediately. Example 4.4 has been modified and given below for demonstrating the use of sprintf(). /*Example 4.5*/ /*program to demonstrate sprintf*/ #include &gt;stdio.h&lt; int main() { char buff[50]; int a; /*multiplicant*/ int b; /*multiplier*/ int c; /*product*/ a=5; /*a is assigned value 5*/ b=4; /*b is assigned value 4*/ c=a*b; sprintf(buff, "product of a&b = %d\n", c);
Self-Instructional Material 77 Data Input and Output NOTES printf("%s", buff); } Look at the program. Strings are always declared as a character array of given size. Since 50 characters are sufficient to hold the result in the program, a character array called buff is created of size 50 by the declaration on top. The next modification is in the print statement. Instead of printf, sprintf has been used in this program. The print arguments are preceded by the name of the string variable, i.e., buff. There is no other change. This will enable storing the result in the string variable. The contents of the variable can be printed at a later time as given in the next statement by using printf(). Thus the results could be stored in another string variable using sprintf() and printed later on. The output of the program is given below. Result of the program product of a&b = 20 4.4 INTERACTIVE PROGRAMMING 4.4.1 Use of scanf() So far, you have been providing data as part of the program itself. You will now get the variables from the keyboard itself at run-time. This is similar to the 'input' statement of BASIC. 'C' does not have a reserved word like 'input' for the same and you can, therefore, use the scanf() function with its declaration in &gt;stdio.h&lt;. Now you can write a program to find the average of two numbers, declaring the numbers and average as floating-point numbers. Begin by keying in Example 4.6. /*Example 4.6*/ /*
program to find the average
of two given numbers*/ #include &gt;stdio.h&lt;
int main() { float

num1, num2; printf("Enter two numbers whose average is to be found\n"); scanf("%
f%f", &num1, &num2); printf("average of %f and %f = %f\n", num1, num2, (num1 + num2)/2.0); } You have declared num1 and num2 as floats. In the next statement, you give a message to enter two numbers whose average is to be found. The function scanf() scans the numbers typed from the keyboard during program execution. Look now at the arguments within the scanf() function. %f has been given twice within quotes. This means that you are going to scan two floating-point numbers. The numbers will be stored in the addresses corresponding to &num1 and &num2, respectively. Check Your Progress 3. What does conversion characters do? 4. How are strings declared?
78 Self-Instructional Material Data Input and Output NOTES &num1 simply means that num1 is a variable and &num1 is the address of the variable in the memory. You must prefix variables with & to denote the addresses of integer, character and floating-point variables. This is not required in the case of strings. You are familiar with the next statement. It will print the average in the following steps: Step 1 : Print "average of" Step 2 : Fetch the value of the floating-point variable appearing first, namely num1 Step 3 : Print value of num1 Step 4 : Print "and" Step 5 : Fetch the value of num2 Step 6 : Print value of num2 followed by the = sign Step 7 : Calculate the value of (num1 + num2)/2.0, which is the average and print this value. All variables will be printed in the floating-point notation. The interaction with the system (when the program is executed) will be as follows: Enter two numbers whose average is to be found. 10.0 20.0 average of 10.000000 and 20.000000 = 15.000000 When the scanf() function asks for the values, key them in and press the Enter or Return key. If only one value was expected from the user, after a Return the program will proceed to the next operation; otherwise, wait for the next entry. Once all variables have been received, after a Return it will go on to execute the next operation. When more than one value is to be given each value can be followed by one of the following: • Tab key • Space bar However, once all values have been given, hitting the Return key should indicate the end of the input data. Note that scanf() by itself does not give any message. Whenever scanf() is encountered, the cursor will blink on the screen. Since we may not know what to enter, we build a message as given in the first printf() statement in the program. Thus scanf() is a very useful function for getting input. In the scanf() function, the formats of the variables to be scanned are to be given sequentially in the order in which the values for variables are to be received (typed by the user). The formats of the variables together are to be enclosed within quotes. There will be no comma between the format specifiers. After the ending of the quote, the addresses of the variables are to be indicated in the case of integers, characters and floating-point numbers. The format specifiers are nothing but the conversion characters. In this example, the values can be input in any form, i.e., integer form, float form or exponential form, but the result will be printed only in the float form. Even if you give Self-Instructional Material 79 Data Input and Output NOTES the value as 100 followed by the enter or tab or space key and then 200, the result will appear as, average of 100.00000 and 200.000000 = 150.000000 The result will be the same even if you enter 1.0e2 and 2.0e2. 4.5
SINGLE CHARACTER INPUT/OUTPUT Characters can be scanned through the scanf() function and printed through the printf() function as given in Example 4.7. /*Example 4.7*/ /*input and output of character through scanf() and printf()*/ # include&gt;stdio.h&lt; int main() { char alpha; printf("Enter a character\n");
scanf("%c", &
alpha); printf("\n The character typed by you is:- %c\n", alpha); } The program is simple to understand. A variable alpha is declared of type char. The message directs the user to enter a character. The scanf() function uses the conversion character %c since you have to scan a character type variable. The (ampersand) &
alpha indicates
the address where the scanned character is to be stored. There are two points to be noted here. We have to specify the format as %c and after entering the character, we have to hit the Return key. The interaction with the computer while executing the program was captured and given below: Enter a character S The character typed by you is:- S 4.6
UNFORMATTED INPUT/OUTPUT There is another way of getting a character by using unformatted input/output. Key in the example shown. /*Example 4.8*/ /*input and output of character through getchar() and putchar()*/ # include&gt;stdio.h&lt; int main() { char alpha; printf("Enter a character\n");
80
Self-Instructional Material Data Input and Output NOTES alpha = getchar(); putchar(alpha); } getchar() is a library function that returns the character read from the standard input device. There is no conversion character associated with it. After entering the character the Return key has to be hit as in the case of scanf(). You have to assign the getchar() function to a character variable. Similarly, putchar(alpha) writes the value of alpha onto the screen. There is no format or conversion character needed for using both getchar()and putchar(). The use of arguments is clear from the putchar(alpha) statement. You are invoking putchar() with an argument alpha, i.e., the putchar() function is executed with the argument alpha. This will become clear when functions are discussed later in the book. The input and output of program execution follows as, Enter a character w w You have avoided the complicated conversion character specification for reading or writing characters but we still have to hit the Return key. Can we avoid hitting the Return key? This is possible by using the getch() or getche() functions. Either of the functions can be used to get one character from the standard input device, i.e., the keyboard, without the need to hit the Return key. The getch() function gets the character typed without displaying it on the monitor. The function getche() gets the character by echoing it on the screen, i.e., displaying the character typed; otherwise, there is no difference in their operation. There is no need to press the Enter key in both these cases. However, the declarations for these library functions are defined in &gt;conio.h&lt; and hence, conio.h has to be included in the program whenever getch(), getche()and putch() are used. The counterpart for getch() is putch(). Example 4.9 illustrates the use of these functions for getting and printing single characters. /*Example 4.9*/ /*input and output of a character through getch() and putch()*/ #include&gt;stdio.h&lt; #include&gt;conio.h&lt;/*conio.h required*/ int main() { char alpha; printf("Enter a character\n"); alpha = getch(); putch(alpha); } Result of the program Enter a character J

Self-Instructional Material 81 Data Input and Output NOTES Note that in this case, as soon as you type j, the letter j appears. It is due to the execution of the last statement and not what was typed. However, the character typed does not show up and hence, you see only one 'j' 4.7
STRINGS—gets() AND puts() A string is an array of characters. The functions gets() and puts() are appropriate when strings are to be received from the screen or sent to the screen without errors. 4.7.1 Standard Library for Strings There are a number of library functions for string manipulation as given below: strlen (CS)—returns the length of string CS.
char * strcpy (s, ct)—copy string ct to string s, including
NULL and return s. char *
strcat (s, ct)—concatenate string ct to end of string
s;
return s. int strcmp (cs, ct)—compare string cs to string ct; return &gt; 0 if cs &gt; ct, 0 if cs = = ct or &lt; 0 if cs &lt; ct
char * strchr (cs, c)—returns the pointer to the
first occurrence of c in cs or NULL if not present.
There are some more string functions. If these are to be used, &gt;string.h&lt; should be included before the main function. 4.7.2 Use of gets() and puts() You can use scanf() to receive strings from the screen. The program using scanf() for reading a name is
given below:
char name [25]; scanf (" %s ", name); Strings can be declared as an array of characters as shown above. In the scanf() function, when you get the array of characters as a string, it is enough to indicate the name of the array without a subscript. When you get a string, there is no need for writing '&' in the scanf() function. You can indicate the name of the string variable itself. Strings may contain blanks in between. If you use a scanf() function to get the string with a space in between such as 'Rama Krishnan', Krishnan will not be taken note of since space is an indicator of the end of entry of the input. But gets() will take all that is entered till the enter key is pressed. Therefore, after entering the full name, the enter() key can be pressed. Thus, using gets() is a better way for strings. You can get a string in a much simpler
way using gets(). The syntax for gets is, gets(name); Similarly, puts() can be used for printing a variable or a constant as follows: puts (name); puts ("Enter the word");
Check Your Progress 5. Define scanf() function. 6. Differentiate between scanf() and printf(). 7. Define getchar() function.
82 Self-Instructional Material Data Input and Output NOTES
However, there is a limitation. printf() can be used to print more than one variable and scanf() to get more than one variable at a time in a single statement. However, puts() can output only one variable and gets() can input only one variable in one statement. In order to input or output more than one variable, separate statements have to be written for each variable. As you know that gets() and puts() are unformatted I/O functions, there are no format specifications associated with them. Take another interesting example: If a word is a palindrome, you will get the same word when you read the characters from the right to the left as well. Examples are : nun malayalam These words when read from either side give the same name. You will write a program to check whether a word is a palindrome or not. This program uses a library function called strlen(). The function strlen(str) returns the size or length of the given string. Now
take a
look at the program. /*Example 4.10*/ /* to check whether a string is palindrome*/ #include &gt;stdio.h&lt; #include &gt;string.h&lt; #define FALSE 0 int main() { int flag = 1; int right, left, n; char w[50]; /* maximum width of string 50*/ puts("Enter string to be checked for palindrome"); gets(w); n=strlen(w)-1; for ((left = 0, right = n); left &gt;= n/2; ++left, − −right) { if (w[left]!=w[right]) { Flag = FALSE; break; } } if (flag) { puts(w); puts("is a palindrome"); } else printf("%s is NOT a palindrome"); }
Self-Instructional Material 83 Data Input and Output NOTES
Result of the program Enter string to be checked for palindrome palap palap is a palindrome If strlen() or gets() or puts() are used in a program, you have to include &gt;string.h&lt; before the main(). Step 1: Now analyse the functioning of the above example: You are defining a symbolic constant FALSE as 0. You initialize flag as 1. You define a string w as an array of 50 characters. gets(w); returns the word typed and stores it from location &w [0] Assume that you typed 'nun' and analyse what happens in the program. strlen (w) will return the length of the word typed. In this case strlen (w) = 3. You subtract this by 1 to get the subscript of the rightmost character. The subscript of the leftmost character is obviously 0. You are initializing the for loop with the following: left = 0 right = n = 2 flag = 1 You check whether left &gt; = n/2 and Since it is so, you check whether w [0]! = w [2]. The condition is false since w[0] = w[2] = 'n'. Therefore, the groups of statements following if are skipped : flag remains 1. Step 2: Now left is incremented to 1 and right is decremented to 1. Again w[1] ! = w[1] is false, flag remains 1. Therefore, control returns to the for statement. Now, left = 2 right = 0 Since left is greater than n/2, control comes out of the for loop. Now the statement if (flag) will be executed. It will check whether flag is true. In this case, flag is still true. Therefore, the computer prints that the word is a palindrome. If the word is not a palindrome, then what happens? Now assume that you typed 'book' and see what happens in the program. To start with, left = 0 right = 3 w [left] ! = w[right], Therefore, the statements within the { } will be executed, flag will be set to false and then the break statement will be executed. The statement break causes immediate exit from the loop. Now
flag is false. Therefore, the else statement is executed to say that the word is NOT a palindrome.
Check Your Progress 8.
When are functions gets() and puts() used? 9. When do we use the printf(), scanf(), puts() and gets() functions?
84 Self-Instructional Material Data Input and Output NOTES 4.8 SUMMARY In this unit, you have learned about
the input and output functions of a computer. These functions facilitate communication with the external world
and help the user to interact with the computer in the desired manner.

The user has to input data in order to process it in the computer and get the result as output. The peripheral device for input is the keyboard. Keying in of the input data into the computer at run-time is achieved easily by library functions, which are supplied along with the 'C' compiler and have been standardized. The functions, which enable keying in of the input data in the keyboard are scanf(), getchar(), getch(), getche()
and gets(). You learnt that the computer also
communicates its output, i.e., the result of computation, either through the console video monitor, printer, disk drive or input/output ports.
The standard library functions for output are printf(), putchar(), putch() and
puts(). Giving input and getting output are achieved using the standard library functions
and all the
function names are followed by parentheses to indicate to the compiler that they are functions.
The parentheses are meant for passing arguments or getting arguments.
You also learnt that
the arguments may be absent in certain functions as in the case of main().
Arguments can be either variables or constants. 4.9 KEY TERMS • Conversion characters: It provides a valuable input to the print statement and corresponds to the variable type. The printf() function is called in the program with a list of arguments given within parentheses. If a wrong conversion character is specified, then the result will be zero. • String: It is an array of characters of given size and is defined using type char to perform string input/output operations. • scanf(): It is a very useful function for getting input. The formats of the variables to be scanned are to be given sequentially in the order in which the values for variables are to be received (typed by the user). The formats of the variables must be enclosed within quotes and there will be no comma between the format specifiers. • getchar(): It is a library function that returns the character read from the standard input device. There is no conversion character associated with it. 4.10 ANSWERS TO 'CHECK YOUR PROGRESS' 1.
The computer communicates its output, i.e., the result of computation, either through the console video monitor, printer, disk drive or input/output ports. 2.
The escape sequences carry out the functions assigned when their turn for execution is reached in the printf() statement. 3.
Conversion characters provide a valuable input to the print statement. It is the duty of the programmer to specify the right conversion character; otherwise, it may lead to errors. The conversion character should correspond to the variable type. Thus, the printf() function is called in the program with a list of arguments given within parentheses. 4. Strings are always declared as a character array of given size.
Self-Instructional Material 85 Data Input and Output NOTES 5. scanf() is a very useful function for getting input. In the scanf() function, the formats of the variables to be scanned are to be given sequentially in the order in which the values for variables are to be received (typed by the user). The formats of the variables together are to be enclosed within quotes. There will be no comma between the format specifiers. 6.
Characters can be scanned through the scanf() function and printed through the printf() function. 7.
getchar() is a library function that returns the character read from the standard input device. There is no conversion character associated with it. After entering the character the Return key has to be hit. We have to assign the getchar() function to a character variable. 8. The
functions gets() and puts() are appropriate when strings are to be received from the screen or sent to the screen without errors. 9.
printf() can be used to print more than one variable and scanf() to get more than one variable at a time in a single statement. However, puts() can output only one variable and gets() can input only one variable in one statement. 4.11
QUESTIONS AND EXERCISES Short-Answer Questions 1. State whether true or false: (
a) scanf() is an unformatted input function. (b) \t causes the cursor to jump to the next vertical tab while printing. (c) When the specified total field width is less than the required width, the format specification for the number is not obeyed for the whole number. (d) %3.1f means the total width = 4 (e) The scanf() statement must specify the address of the variable. 2. Discuss the output of the following programs. Try to find the output on paper and confirm this with execution in a computer. (a) #include &gt;stdio.h&lt; int main() { putchar('

**43%** **MATCHING BLOCK 19/126** W

c') ; } (b) #include &gt;stdio.h&lt; int main() { unsigned a,b,c; a=10; b=5; c= (a*a)+(b*b)+(2*a*b); printf("Square of %d + %d is equal to %d ",a,b,c); } 86

Self-Instructional Material Data Input and Output NOTES Long-Answer Questions 1. Explain the input and output functions used in C programming. 2. What are conversion characters? Why are they used? 3. How will you convert an octal to a hex? 4. Why is interactive programming useful? Write an interactive program to add any three decimal numbers. 5. How is single character input/output function performed? 6. What do you mean by unformatted input/output. 7. Explain the importance of gets() and puts() functions. Write a program using these functions to accept the string from the user and then display it on the screen. 8. Write programs for the following: (a) To find out whether a given unsigned integer less than 11 is prime or not. (b) To find out whether a number is divisible by 2, 3, 4, 5 and 6. (c) To get 4 floats at run-time and print their average. 4.12
FURTHER READING Gottfried, Byron S. Programming with C, 2nd edition. Schaum's Outline Series. New York: McGraw-Hill, 1996.

Forouzan, Behrouz A. and Richard F. Gilberg. Computer Science: A Structured Programming Approach Using C, 2nd edition. California: Thomson, Brooks Cole, 2000. Dey, Pradip and Manas Ghosh. Computer Fundamentals and Programming in C. New Delhi: Oxford Higher Education, 2006. Bronson, Gary J. A First Book of ANSI C, 3rd edition. California: Thomson, Brooks Cole, 2000. Kanetkar, Yashwant. Understanding Pointers in C.
New Delhi: BPB Publication, 2001. Kanetkar, Yashwant. Let us C. New Delhi: BPB Publication, 1999.
Kernighan, Brian W. and Dennis Ritchie. C Programming Language, 2nd edition. New Jersey: Prentice Hall, 1988. Foster, W. D. and L. S. Foster. C by Discovery. Boston: Addison-Wesley, 2005. Kanetkar, Yashwant. Working with C. New Delhi: BPB Publication, 2003.
Horton, Ivor. Instant C Programming. New Jersey: Wrox Press (John Willey & Sons), 1995. Lawlor, Steven C. The Art of Programming Computer Science with 'C'. New Jersey: West Publishing Company, 1996.
Self-Instructional Material 87 Control Statements NOTES UNIT 5 CONTROL STATEMENTS
Structure 5.0 Introduction 5.1 Unit Objectives 5.2 Branching 5.2.1
If Statement 5.2.2 If...else Statement 5.2.3 Nesting of the if...else Statements 5.2.4 Logical Operators and Branching 5.2.5 Conditional Operator and if...else 5.3
Loops and Control Constructs 5.3.1 Iteration using if 5.3.2 For Statement 5.3.3 Symbolic Constants and Looping 5.3.4 Other Forms of the for Loop 5.3.5 The while Loop 5.3.6 Do...while 5.4 Linear Search 5.5 Switch Statement 5.6 Break, Continue, Return 5.7 Significance of the Comma 5.8 Summary 5.9 Key Terms 5.10
Answers to 'Check Your Progress' 5.11 Questions and Exercises 5.12 Further Reading 5.0 INTRODUCTION In this unit, you will learn about the control statements used in C programs. These are used for solving complex problems. Depending on the occurrence of a particular situation if and else keywords are used for branching to different segments of the program. 'C' is ideal for handling branching because the syntax is clear and unambiguous.
You will learn that relational operators are used in conjunction with branching constructs.
If the condition is true, then a single statement or group of statements following if will be executed. If more than one statement is to be executed, then the statements are grouped within braces. To perform the same operation a number of times, loop or iteration is used. You will learn about
symbolic constants which are written in capital or upper case letters. Further, in this unit, you will learn about the search process which is used for finding out whether a given number or string is present in an array of data. This problem is encountered in many real-life situations. Searching for a name in a telephone directory or a voter's list are all quite common problems. Here, the entire array is compared from the beginning till the end. If the item to be searched matches with an item in the array of data, then the search is stopped; otherwise, the search is continued till the end of the array. This method of searching is called linear search.
In this unit, you will also learn about while, do ....while, switch, break, continue and return statements.
You can use commas to declare more than one data type.
88 Self-Instructional Material Control Statements NOTES 5.1 UNIT
OBJECTIVES After going through this unit, you will be able to: • Understand the basic concept of
branching • Use if and if...else branching statements in your C programs •
Explain logical and conditional operators •
Understand the importance of loop and control constructs in a C program • Use if, for, while and do...while in a program •
Do linear search using C program • Define switch, break, continue and goto statements • Explain the significance of the comma 5.2 BRANCHING
Real-life application programs do not merely consist of simple multiplication or addition. They call for solving complex problems. Depending on the occurrence of a particular situation, we may follow different paths; the
if and else keywords
are quite handy in branching to different segments of the program. 'C' is ideal for handling branching because the syntax is clear and unambiguous. You will now read about the branching constructs. Relational operators are used in conjunction with branching constructs. Hence,
you
look at them first. 5.2.1
If Statement The syntax of the if statement is as follows: if (condition) {
statements} If the condition is true, then a single statement or group of statements following the if will be executed. If more than one statement is to be executed, then the statements are grouped within braces. If it is a single statement, then curly braces are not required. If the condition turns out to be false, then the next statement after those belonging to the if will be executed. Example 5.1 will make the concept clear. Input two integers from the keyboard. If they are equal, then the program will print, 'you typed equal numbers'; otherwise, it will print nothing. /*Example 5.1
This
program demonstrates
the use of if*/ #
include &gt;stdio.h&lt; main() { unsigned int a,b; printf ("
enter two integers\n"); scanf("%u%u", &a, &b);
Self-Instructional Material 89 Control Statements NOTES
if (a==b) { printf("you typed equal numbers\n"); } } Each opening curly brace has to have a matching closing curly brace. In Example 5.1 the first closing curly brace corresponds to the if statement and the second one to the main function. Execute the program by first keying in two equal valued unsigned integers. Result of the program enter two integers 56 56 you typed equal numbers After you are satisfied, you can try the program with unequal numbers. You will not get any message. 5.2.2 If...else Statement

You

| 92% | MATCHING BLOCK 20/126 | W |
|---|---|---|

did not get any message when the numbers were unequal and this can be avoided by using the else statement. The usage of if .. else is shown below. if (condition true) { statements s1 } else { statements s2 } statements s3; The statement else is always associated with an if. If the condition is true, then statements s1 will be executed. After executing them, the program will skip the else block and control goes to statement s3 that follows the else block. If the condition is false, then the statements in the else block, i.e., s2 will be executed followed by statement s3. Statements s1 will not be executed at all when the condition becomes false. The usage of braces clearly brings out which statements belong to the if block and which to the else block. Example 5.2 brings out the usage of if... else. /*Example 5.2 This program demonstrates use of if.. else*/ #include &gt;stdio.h&lt; main() { 90 Self-Instructional Material Control Statements NOTES unsigned int a,b; printf ("enter two integers\n"); scanf("%u%u", &a, &b); if (a==b) { printf("you typed equal numbers\n"); } else { printf("numbers not equal\n"); } } The output of the program when unequal numbers were keyed in is as follows. Result of the program enter two integers 17 13 numbers not equal 5.2.3 Nesting of the if...else Statements You witnessed the usage of a single if statement in Example 5.1. You saw if followed by else in Example 5.2. There is no restriction to the number of if, which can be used in a program. This applies to else as well, but else can only follow an if statement. You can have the following in a program: { if (condition1) { if (condition2) {statements−s1} else if (condition3) {statements−s2,} } else {statements−s4} statements−s5 } This is called a nested if and else statement. As the level of nesting increases, it will be difficult to analyse and logical mistakes will be made more easily. In the above example, when condition1 is false, statements−s4 will be executed. If condition1 is true and condition2 is also true, then statements− s1 will be executed.

Self-Instructional Material 91 Control Statements NOTES
If condition1 is true and condition2 and condition3 are false, statements−s5 will be executed directly. To execute statements−s2 Condition1 has to be true; Condition2 has to be false, And condition3 has to be true. Try to analyse this yourself. There are better methods to solve the above problem, which will be discussed later. For example, three unequal integers are keyed in and are called x, y and z. Write a program to find the greatest of the three numbers. Before
writing a
program, we must write the algorithm. We should not straight away get down to programming. ALGORITHM 1 Algorithm for finding the greatest of 3 integers. Step 1: Print a message to enter 3 integers. Step 2: Get three numbers and store them at &x, &y and &z. Step 3: Check if x &lt; y Step 4: If false, go to step 9 Step 5: If true Step 6: Check if x &lt; z Step 7: If true, write x is the greatest; End Step 8: If false, write z is the greatest; End Step 9: Check if y&lt;z Step 10: If true, write y is the greatest; End Step 11: If not, write z is the greatest. End Now code these steps into a 'C' program, which is shown in Example 5.3. /*Example 5.3
This program demonstrates
the use of the nested if.. else*/ #
include &gt;stdio.h&lt;
main() { int x,y,z; printf ("enter three unequal integers\n"); scanf("%d%d%d", &x, &y, &z);
if(x&lt;y) {
if(x&lt;z) { printf("x
is greatest\n"); }
92
Self-Instructional Material Control Statements NOTES else {
printf("z is greatest\n"); } } else { if(y&lt;z) { printf("y is greatest"); } else { printf("z is greatest"); } } } Test the correctness of the program by giving a different set of values for x, y and z. Result of the program enter three unequal integers 908 231 907 x is greatest Look at the example. It uses multiple nesting of if .. else. Take care to see that every opening brace has a corresponding closing brace. It is better to indent the braces as shown in the example so that no mistake is committed. Take care to see that else matches with the corresponding if and each opening brace {matches with a corresponding closing brace}; if either an opening {or closing} is extra, then an error will result. 5.2.4 Logical Operators and Branching In the above examples you have been checking one condition at a time. It would be nice if you could check more than one condition at a time. 'C' provides three logical operators for combining more than one condition. These are as follows: Logical and represented as && Logical or represented as || Negation or not represented as ! (exclamation).
Take a look at
some examples of usage of the logical operators. In Example 5.3 it was concluded that, if x &lt; y and if x &lt; z, then x is the greatest. You will represent the same as, if ((x &lt; y) && (x &lt; y)) printf ("x is the
greatest");
Self-Instructional Material 93 Control Statements NOTES

You will see that the program has become much more elegant. The syntax for && is, if ((condition1) && (condition2)) { statements−s1 } Statements−s1 will be executed only if both the conditions are true. The syntax for 'or' is as follows: if ((condition 1 ) ||(condition 2)) { statements−s2 } In this case, even if one of the conditions is true, the statements−s2 will be executed. At least one condition has to be true for the execution of s2. However, if both are false, s2 will not be executed. The NOT operator with symbol ! can be used along with any other relational or logical operator or as a stand-alone operator. It simply negates the operator following it. The syntax of '!' is as follows: if ! (condition) statement s3; s3 will be executed only when the condition is not true or the condition is false. Now rewrite Algorithm 1 by using the logical operators. The revised Algorithm 2 is shown here: ALGORITHM 2 Step 3: If (x &lt; y) and (x &lt; z), x is the greatest. Step 4: Else if (x&gt;y) and (y&lt;z), y is the greatest. Step 5: Else print z is the greatest. The complete program is given in Example 5.4. /*Example 5.4 This Example demonstrates the use of logical operators*/ #
include &gt;stdio.h&lt;
main() { int x,y,z; printf ("enter three unequal integers\n"); scanf("%d%d%d", &x, &y, &z);
if ((x&lt;y) && (
x&lt;z)) printf("x
is greatest\n"); else { if((x&gt;y) && (y&lt;z)) printf("y is greatest\n"); else
94 Self-Instructional Material Control Statements NOTES printf("z is greatest\n"); } } Result of the program enter three unequal integers 12 23 78 z is greatest Now write a program to convert a lower case letter typed into an upper case letter. For this purpose you may have to refer to the ASCII table in Annexure 1. It is obvious that if
you
subtract 32 from the ASCII value of a lower case alphabet,
you
will get the ASCII value of the corresponding upper case letter. Now write an algorithm for the conversion of lower case to an upper case letter. It is given in Algorithm 3.
ALGORITHM 3 Step 1: Send a message for getting a character Step 2: Get a character Step 3: Check whether the character typed is &lt;=a and &gt;=z (This is essential since you can only convert a lower case alphabet into upper
case.)
Step 4: If so, subtract 32 from the value of the character; if not, go to step 6 Step 5: Output the character with the revised ASCII value; END Step 6: Print 'an invalid character' END The algorithm is implemented in Example 5.5. /*Example 5.5 Conversion of lower case letter to upper case*/ #include &gt;stdio.h&lt; main() { char alpha; printf ("enter lower case alphabet\n"); alpha=getchar(); if (( alpha &lt;='a')&& (alpha&gt;='z')) { alpha= (alpha-32); putchar (alpha); } else printf("invalid entry; retry"); } Now you can test the program by giving both the valid and invalid inputs; valid inputs are the lower case letters and invalid inputs are all other characters.
Self-Instructional Material 95 Control Statements NOTES
Result of the program The result for the invalid input is as follows: enter lower case alphabet 8 invalid entry; retry The result when tried with a valid input is given below: enter lower case alphabet n N The programs should be executed, i.e., tested with both the valid and invalid inputs. 5.2.5 Conditional Operator and if...else The syntax for the conditional operator is as shown here: (Condition)? statement1: statement2; What does it mean? If the condition is true, execute statement1; else, execute statement2. Here nesting is not possible. The if...
else statement
is more readable than the conditional(?) operator. However, the conditional operator is quite handy in simple situations as follows: (a &lt; b) ? print a greater : print b greater; Thus, the operator has to be used in simple situations. If nothing is written in the position corresponding to else, then it means nothing is to be done when the condition is false. Example 5.2 is rewritten using the ? operator in Example 5.6. /*Example 5.6 This Example demonstrates use of the ? operator*/ #
include &gt;stdio.h&lt; main() { unsigned
int a,b; printf ("enter two integers\n"); scanf("%u%u", &a, &b); (a==b)?printf("
you typed equal numbers\n"): printf("numbers not equal\n"); } Result of the program enter two integers 123 456 numbers not equal Check Your Progress 1. Define if statement. 2. Explain the logical operators provided by 'C'. 3. What happens when you
subtract 32 from the ASCII value of a lower case alphabet?
96
Self-Instructional Material Control Statements NOTES 5.3 LOOPS AND CONTROL CONSTRUCTS Quite often, you
have to perform the same operation a number of times. You may also have to repeat the same operation with one or more of the values changed, which is known as loop or iteration. It is definitely possible to write a program for such situations with the concepts you have learned so far. However, there are special constructs in any programming language for carrying out repeated calculations. 5.3.1 Iteration using If Before you look at loop constructs, let us consider an example to see the need for repetitive calculations. Assume that
you
want to find the sum of the first 10 natural numbers 1 to 10. This can be achieved through successive addition, i.e., first
you
initialize the sum to 0 and then add 1 to the sum. Next you add 2 to the sum, then 3, and so on till you add 10 to the sum. Thus, by repeated addition 10 times,
you
have found the sum of first 10 natural numbers. The algorithm below summarizes what
you
have done: Step 1: Sum = 0 Step 2: I = 1 Step 3: If I &gt; = 10 perform the following operations: sum = sum +

I; I = I + 1;
Step 4: Print the sum Now analyse the algorithm: At the beginning, steps 1 and 2 are entered with sum = 0 and I = 1 since I &gt; = 10 Sum will be equal to sum + I, i.e., sum = 0 + 1 = 1 I = I + 1, i.e., I = 2 Now the program goes to step 3. with I = 2 and sum = 1 Since I &gt; = 10 sum = sum + I sum was 1 and I is 2 sum = 1 + 2 = 3 Next I will be incremented to 3 Third iteration: Step 3 is approached with I = 3 and sum = 3 since I &gt; = 10 sum = sum + I = (1 + 2) + 3 I = 4 Ninth iteration: Step 3 is approached with I = 9
Self-Instructional Material 97 Control Statements NOTES since I &gt; = 10 sum = (1 + 2 + 3 +...........+ 8) + 9 I is incremented to 10 Tenth iteration: Step 3 is approached with I = 10 since I &gt; = 10 sum = sum + I = (1 + 2 + 3 +........+ 8 + 9) + 10 Now I is incremented to 11 since I &gt;= 10 is not true, the program does not execute the statements following the if and jumps to step 4. In step 4 the sum is printed. This algorithm is implemented in program 5.7. /*Example 5.7 demonstrates use of if for iteration*/ #include &gt;stdio.h&lt; main() { int sum=0, i=1; /*declaration and initialization combined*/ step3: /*label- loop starts here*/ if (i &gt;=10) { Sum = sum + i; i = i + 1; goto step3; } printf("sum of first 10 natural numbers=%d\n", sum); } Result of the program sum of first 10 natural numbers = 55 The program
has implemented the algorithm in toto.
The program uses if and goto keywords. According to the algorithm, the program has to go to step3. Step3 in this program is called a label, which is followed by a colon. The rules for coining a label name are the same as for an identifier. The label can be placed anywhere in the same function where the goto statement is found. Usage of goto is considered to be bad programming practice since it leads to errors when changes are made in the program and also affects readability. It is always possible to write a program without using goto.
The program can be rewritten without goto by using a for statement. 5.3.2 For Statement
The
for statement is meant for the easy implementation of iterations unlike if. The syntax of for is
given
ahead:
98 Self-Instructional Material Control Statements NOTES
for (exp1; exp2; exp3) {statements;} Note the keyword, the parentheses and semicolons. There is no semicolon after exp3, exp1, exp2 and exp3 are expressions. The usage of the for loop is given below: exp1– Contains the initial value of an index or a variable. exp3– Contains the alteration to the index after each iteration of the body of the for statement.
The body of the statement is either a single statement or a group of statements enclosed within braces.
If a single statement has to be executed, then braces are not required. exp2– Condition that must be satisfied if the body of statements is to be executed. An example of a for loop is given below: for (i = 0; i &gt; 5; i++) printf("%d", i); The loop will start with an initial value of i = 0. Since i &gt; 5, the body of the for loop will be executed and it will print 0. Now the exp3 will be executed and i will be incremented to 1. Since i is less than 5, body of the loop will again be executed to print 1. This will continue till 4 is printed.
i will
now be incremented to 5 and since i is not less than 5, the for loop will be terminated. This is how for is used to carry out repetitive operations. Now write a program for finding the sum of the first 10 natural numbers using the for statement. The program is given in Example 5.8 /*Example 5.8 demonstrates the use of the for statement to find the sum of the first 10 natural numbers*/ #include &gt;stdio.h&lt; main() { int sum=0, i; /*declaration and initialization combined*/ for (i=1; i&gt;=10; i++) /*loop starts here*/ { Sum = sum + i; } printf("sum of the first 10 natural numbers=%d\n", sum); } Result of the program sum of first 10 natural numbers = 55 Note the difference between Example 5.7 and Example 5.8. You have eliminated the label Step 3 and the goto statement. The initialization of i = 1 is carried out as part of the for statement. The incrementing of i is also carried out as part of the for statement. The program has, therefore, been simplified.
Self-Instructional Material 99 Control Statements NOTES How does the program work? Step 1: i = 1 i is checked with i &gt;= 10 Since i is less than 10, the for loop is executed. sum = sum + i = 0 + 1 = 1 Step 2: i is incremented to 2 2 is &gt;=10 Therefore, the for loop is executed. sum = sum + i = (1) + 2 Step 9: i is incremented to 9 9 is &gt; = 10 Therefore, the for loop is executed. sum = sum + i = (1 + 2 +........+ 8) + 9 Step 10: i is incremented to 10 10 is &gt; = 10 sum = sum + I = (1 + 2 +..........+ 9) + 10 Step 11: i is incremented to 11. 11 is not &gt; = 10. Therefore, the for loop is now terminated. The printf() function is now executed automatically. Now summarize the operation of the for loop. When a program encounters a for loop, it first checks the condition through the expression in the middle. If the condition is satisfied, it executes the group of statements. After executing the statements in the body of the loop, the program transfers the execution to the for statement and the third expression is executed, which is usually incrementing or decrementing. Then the condition is checked. If the condition is not satisfied, the group of statements will not be executed and the program will skip to the next statement after the statements pertaining to the for statement. By chance if the initial value was typed as 11 instead of 1 in the program, the condition will turn out to be false and the group of statements will not be executed at all. Three Components of for
The three components of a for statement are as follows: exp1 and exp3 are assignments or function calls.
Function calls will be discussed at a later stage; exp2 is a relational expression. The three expressions may not always be present. However, even if an expression is not present, the associated semicolon should be present.
For example, for (; exp2 ;) {s1}
100
Self-Instructional Material Control Statements NOTES

Here, the initial value is not specified and the incrementing does not take place after every iteration. Presumably, the initial value is assigned elsewhere and incrementing or a similar operation takes place as part of the group of statements following the for. However, since exp2 is present, the loop will terminate. However, if all three expressions are omitted as follows: for ( ; ; ) the loop will never terminate because the conditional statement is absent. If exp2 is not present, it is assumed that the condition is true always. Such a statement should not be used. Instead of incrementing, you can use i + = 2 as exp3 when i will be incremented by 2 every time. Now try to print the list of even numbers up to 50. The program is
as shown: /*
Example 5.9 variation in for statement - to print even numbers*/ #include &gt;stdio.h&lt; main() { int i=2; for (; i&gt;10; i+=2) /*loop starts here*/ { printf("%i is an even number\n",i); } } Here you initialize i = 2 before the for loop itself. However, the corresponding semicolon is present at the right place. Result of the program 2 is an even number 4 is an even number 6 is an even number 8 is an even number 5.3.3
Symbolic Constants and Looping So far you have been giving the initial value, the increment and final value as part of the programs. Assuming you want to change one or more of them later on, how are you to go about it? You would then have to painfully rewrite the program. C provides a method by which this can be done with the least changes by using the # define statement.
The format of this statement is as follows: # define name constant For example, you can define, # define INITIAL 1 Which defines INITIAL as 1. The INITIAL types of definitions are called symbolic constants. They are not variables and hence, they are not defined as part of the declarations of variables. They are specified on top of the program before the main(). The symbolic constants are to be written in capital or upper case letters. Wherever the symbolic constant names appear in a program, the compiler will replace
Self-Instructional Material 101 Control Statements NOTES
them with the corresponding replacement constants defined in the # define statement. In this case, 1 will be substituted wherever INITIAL appears in the program. Note that there is no semicolon at the end of the # define statement.
You can now write a program to print out the numbers between a given range, say 100 to 150, which are divisible by 3, i.e., when divided by 3 the modulus = 0. Such numbers are 'evenly divisible by 3'. /*Example 5.10 demonstrates the use of symbolic constants-program to find numbers between 100 and 150 evenly divisible by 3*/ #include &gt;stdio.h&lt; #define LOW 100 #define UPPER 125 #define STEP 1 main() { int num; for (num=LOW; num&gt;UPPER; num+=STEP) /*loop starts here*/ { if(num%3 == 0) printf("%i is evenly divisible by 3\n", num); } } Result of the program 102 is evenly divisible by 3 105 is evenly divisible by 3 108 is evenly divisible by 3 111 is evenly divisible by 3 114 is evenly divisible by 3 117 is evenly divisible by 3 120 is evenly divisible by 3 123 is evenly divisible by 3 You can use this technique in future programs. Assume that you want to find out all the numbers evenly divisible by 3 between 1 and 1000. You have to define LOWER as 1 and UPPER as 1000. Assuming that later on you want to find out the numbers evenly divisible by 7, you would have to again rewrite the program. This can be avoided by defining the DIVISOR as 7 and substituting within the condition (number % DIVISOR == 0). In this way, you can write a program to find out the numbers evenly divisible by any number in any range. 5.3.4 Other forms of the
for Loop The for loops can be nested as follows: for (i = 1; i &gt;= 10; i++) {
102 Self-Instructional Material Control Statements NOTES for (j = 1; j &gt;= 5; j++) { for (k = 1; k &gt;= 2; k++) { s1 } } } The statement s1 will be executed as follows: First time i = 1, j = 1, k = 1 Second time i = 1, j = 1, k = 2 Third time i = 1, j = 2, k = 1 i = 1, j = 2, k = 2 i = 1, j = 3, k = 1 i = 1, j = 3, k = 2 Lastly i = 10, j = 5, k = 2 s1 will be executed 2*5*10 = 100 times.
Any level
of nesting is acceptable; However, the higher the level of nesting is, the more easy it will be to commit mistakes and more difficult to understand. Now, look at some more for loops. For ( x = −5; − −x &lt;= −10; ) { } Here, the decrement and conditional statements are combined in exp2. Since decrement is a prefix, the decrement of x is carried out first. The condition is then checked in order to decide whether to continue or not. Then the loop is executed. Therefore, the first iteration will be carried out with x = −6 and the last with x = −10. Another variation of the statement is as follows: for (y = 100; y ++&gt;= 200;) { s2 } Here too exp2 and exp3 are combined. This is a postfix notation. The following sequence is carried out: condition check, increment and execute the loop. Therefore, the statement s2 following the for loop will be executed the first time with y = 101 and finally with y = 201 as well. The for loop is a popular iteration construct not only in 'C' but also in other languages. Here, the initial value, the step and the final value are clear and unambiguous and simple to write. There are other loop statements also. In the next section, you will study the while loop.
Self-Instructional Material 103 Control Statements NOTES 5.3.5

| 92% | MATCHING BLOCK 21/126 | W |
|---|---|---|

The while Loop The while loop is a subset of the for loop. The syntax for the while loop is given below: while (expression) {statements;} This means that the statement(s), which is a single statement or multiple statements, will be executed while the expression is true. When it becomes false, the execution will stop. The while is similar to the for loop without exp1 and exp3. The for loop can be simulated or replaced with the while loop as given below. exp1; while (exp2) { statements exp3; } The programmer can use while or for

at
his discretion. The for
loop is preferred when the initialization and incrementation are simpler. Let us look at
an example.

Let us write a program for the generation of any multiplication table. The program is given below: /*Example 5.11 use of while - You can generate multiplication tables of your choice using this program. caution: Don't exceed maximum limits of integer */ #include &gt;stdio.h&lt; main() { int a,b,product; a=1; b=0; product=0; printf("Enter which table you want"); scanf("%d",&b); while (a &gt;=10) { product = a*b; printf("%2d X %d= %3d\n",a,b,product); a++; } } When the program asks you to enter a table and you type 12, you will get the 12th table
as given below.
104 Self-Instructional Material Control Statements NOTES
Result of the program Enter which table you want 12 1
X 12= 12 2 X 12= 24 3 X 12= 36 4 X 12= 48 5 X 12= 60 6 X 12= 72 7 X 12= 84 8 X 12= 96 9 X 12= 108 10 X 12= 120
Note here that the condition is (a &gt; =10); incrementing is done within the loop in a++. Variable a is initialized as 1 before entering the while loop. If you want to print the table up to 16 × 12 = 192 then simply change the condition to while (a &gt; =16). Simple, isn't it? 5.3.6
Do . . . while This is a modification of the while statement. In the while statement, before the group of statements following the while are executed, the condition associated with the while is checked. If the condition is true or fulfilled, then the associated statements are executed. If not, the program skips the statements associated with the while loop. This is depicted in Figure 5.1. Fig. 5.1 while Loop After execution of the statements, the program will check again whether the condition is true and then continue to execute or skip the statements depending on the condition. The statements may not be executed even once if the condition was false at the entry point. Self-Instructional Material 105 Control Statements NOTES However, the do...while works differently as shown in Figure 5.2.
Fig. 5.2 do...while Loop Here the statements following do will
be executed once before the condition is checked. If it is true, then the statements will be executed again. If not, the program will skip the statements and proceed further. Thus, whatever be the condition, the statements following do will be executed once before checking the condition. This is the essential difference between do...while and while. The while loop tests the condition on top of the loop; but do...while tests at the bottom after executing the statements. The while loop executes the statements after checking the condition; but do...while executes the statements before testing the condition. The syntax of do...while is as follows: do { statements } while (expression); The statement do...while is not used as frequently as the while loop. Example 5.12 converted upper case alphabets to lower case. Incase you want to convert more alphabets you will have to execute the program again and again. This can be avoided by the while loop. The program will continue to run as long as you want to convert more and more alphabets. The program has been rewritten with while for converting upper case to lower case. You can convert as long as you want. When you want to stop, enter 1. The program, which uses while for converting an upper case character to a lower case is given below: /*Example 5.12 Conversion of upper case to lower case alphabet*/ #include &gt;stdio.h&lt; #include&gt;conio.h&lt; main() {
int alpha=0; while (alpha!='1') {
106
Self-Instructional Material Control Statements NOTES
printf ("\nenter upper case alphabet- enter 1 to quit\n"); alpha=getche(); if ( alpha &lt;='A'&& alpha&gt;='Z') { alpha= (alpha+32); putch(alpha); } else { if(alpha!='1') printf("\ninvalid entry; retry"); else printf("End of session"); } } } Result of the program enter upper case alphabet- enter 1 to quit Pp enter upper case alphabet- enter 1 to quit p invalid entry; retry enter upper case alphabet- enter 1 to quit Qq enter upper case alphabet- enter 1 to quit 1End of session
The above
program works till 1 is pressed. It continues to convert upper case to lower case till 1 is pressed. The program has been designed in such a manner that it will perform at least one iteration of the statements following while. This can be rewritten using do...while. The rewritten program is as follows: /*Example 5.13 Conversion of upper case to a lower case alphabet*/ #include &gt;stdio.h&lt; #include&gt;conio.h&lt; main() { int alpha=0; do { printf ("\nenter upper case alphabet- enter 1 to quit\n"); alpha=getche(); if ( alpha &lt;='A'&& alpha&gt;='Z')
Self-Instructional Material 107 Control Statements NOTES {

alpha=(alpha+32); putch(alpha); } else { if(alpha=='1') printf("End of Session"); else printf("\ninvalid entry; retry"); } }while(alpha!='1'); } Result of the program enter upper case alphabet- enter 1 to quit Gg enter upper case alphabet- enter 1 to quit o invalid entry; retry enter upper case alphabet- enter 1 to quit Dd enter upper case alphabet- enter 1 to quit 1End of Session How does it differ? Here too, the program will attempt to convert one character before it can be terminated. Assuming that the first character was 1, the program will still attempt to convert it and print the message "End of Session" before it quits. Suppose the first character is a valid one and a number of characters are converted in succession; when you want to terminate the program, 1 has to be pressed and even then the program will not stop immediately. It will stop only after the statements are executed. Since the problem is the same, a detailed look at both the examples will bring out the similarity in operation between both the constructs. However, there are occasions when it is quite suitable as given in the next section. 5.4

LINEAR SEARCH Search is the process of finding out whether a given number or string is present in an array of data. This problem is encountered in many real-life situations. Searching for a name in a telephone directory or a voter's list are all quite common problems. Searching and sorting algorithms and programs will be discussed in a number of units. Here, the entire array is compared from the beginning till the end. If the item to be searched matches with an item in the array of data, then the search is stopped; otherwise, the search is continued till the end of the array. This method of searching is called linear search. The algorithm for linear search is as follows:
108 Self-Instructional Material Control Statements NOTES ALGORITHM 1 Declaration array ia of size n, i = 0 item to be searched = x found = false do if ( ia [i] == x ) then found = true else i = i + 1 while ( i &gt;= n " 1 && not found ) The program, which implements the algorithm for linear search, is as follows: /*Example 5.14 - Linear search*/ #include &gt;stdio.h&lt; main() { int i=0, ia[]={100, 200, 300, 400, 500}; int x=0, found=0; printf("\nEnter the element to be searched\n"); scanf("%d",&x); do { if (ia[i]==x) found=1; else i++; } while (i&gt;=4 && found==0); /*precedence of && is lower*/ if(found==1) printf("%d found", x); else printf("%d NOT FOUND", x); } Result of the program Enter the element to be searched 150 150 NOT FOUND The statement do...while is quite suitable for this program. The loop will be executed once without checking the condition, which is quite alright in this case. By chance if the number to be checked is the first item in the array, then found will become 1 and hence, the program will come out of the loop. If the number is not found when all the items are compared, then too the program will come out of the loop. The loop will be terminated when found == 1 or i becomes greater than n−1. Check Your Progress 4.
Why is goto not used in a program? What can be used instead of goto? 5.
Write the syntax for for statement. 6. What are symbolic constants? How are they defined? 7. Define while loop. 8. Differentiate between while and do...while loops. Write the syntax for the do...while loop.
Self-Instructional Material 109 Control Statements NOTES 5.5
SWITCH STATEMENT Switch statements allow clear and easy implementation of multiway decision-making. Assuming that a number is received from the keyboard and depending on the value, we want to carry out some operations, the switch statement can be used effectively in this situation. In simpler situations if...else could be used, and in complex situations, switch can be used. For example, if you get numbers starting from 1 to 4 and print their values in words, you can use the if...else statement as given in
the program 5.15 below: /*
Example 5.15 converts the digits 1-4 in words using if*/ #
include &gt;stdio.h&lt; #include &gt;conio.h&lt; main() { int a; char ch ='c'; while (ch=='c') { printf("\nEnter a digit 1 to 4\n"); scanf("%d",&a); if(a== 1) printf("One\n"); else if (a== 2) printf("Two\n"); else if(a== 3) printf("
Three\n"); else if(a==4) printf("Four\n"); else printf("Illegal character\n"); printf("enter 'c'if you want to continue\n"); printf("or any other character to end\n"); ch=getche(); if (ch!='c')
printf("End of Session"); } } Result of the program Enter a digit 1
to 4 3
110
Self-Instructional Material Control Statements NOTES
Three enter 'c' if you want to continue or any other character to end c Enter a digit 1 to 4 1 One enter 'c' if you want to continue or any other character to end c Enter a digit 1 to 4 2 Two enter 'c' if you want to continue or any other character to end nEnd of Session You have struggled hard to do this exercise. Assuming that you

want to get a one digit number up to 9 and print its value in words, it would become a more complex task. The switch statement comes in handy in such situations. The syntax of the switch statement is as follows: switch (expression) { case constant or expression : statements case constant or expression : statements .. default : statements } When the switch keyword is encountered, the associated expression is evaluated. The program now looks for the case, which matches with the value of the expression. Execution then starts from the statement corresponding to the case which matches. Each case has to be accompanied by integer expressions, which must be unique, as otherwise the program will not know where to start. For example, if the first two cases are as follows: case 10 : s1; case 10 : s2; In this case, the program would not know whether to execute s1 or s2 when the expression of switch evaluates to 10. Therefore, the constant expressions following the case keyword should all be unique. There may be occasions when none of the constant expressions matches the switch expression in which case the default statements will be executed. Thus, switch allows branching of the program execution to an appropriate place. A program to print the values of the digits in words is given below: /*Example 5.16 converts the digits 0-9 in words*/ #include &gt;stdio.h&lt; #include &gt;conio.h&lt; Self-Instructional Material 111 Control Statements NOTES main() { int a; char ch ='c'; while (ch=='c') { printf("\nEnter a digit 0 t0 9\n"); scanf("%d",&a); switch(a) { case 0:printf("Zero\n"); break; case 1:printf("One\n"); break; case 2:printf("Two\n"); break; case 3:printf("Three\n"); break; case 4:printf("Four\n"); break; case 5:printf("Five\n"); break; case 6:printf("Six\n"); break; case 7:printf("Seven\n"); break; case 8:printf("Eight\n"); break; case 9:printf("Nine\n"); break; default:printf("Illegal character\n"); } printf("enter 'c' if you want to continue\n"); printf("or any other character to end\n"); ch=getche(); if (ch!='c') printf("End of Session"); } } Result of the program Enter a digit 0 t0 9 6 Six enter 'c' if you want to continue 112 Self-Instructional Material Control Statements NOTES or any other character to end c Enter a digit 0 t0 9 7 Seven enter 'c' if you want to continue or any other character to end nEnd of Session

We have
used

the do...while concept in this example also. The program first enters the do loop. After the execution of the loop, the program asks you to enter c if you want to continue or any other character to end the program. If you enter c, the program will continue. Thus, if you want to exit you can press any other letter, say 'n'. If you press 'n' the program stops instantly. This is the right way of using the do...while loop. Now the switch is analysed. The program asks for entry of a digit 0 to 9. The entered digit is stored in variable a. If a = 5, then the program goes to case 5. It is followed by printing the value Five and then break. If you press 8, then case 8 matches and Eight will be printed. What is a break statement ? Assume that all the break statements are removed from the program. Then if you press 1, 'One' will be printed and all the statements following that will be executed. This means that Two, Three, ... till Nine will be printed. If you enter 7, it will print all the numbers starting from Seven, which in not desirable. The break statement has, therefore, been introduced. After printing the value of the number, the break statement takes the program to the end of the switch statement. The end is just the closing brace corresponding to switch after the default printf() statement. Therefore, the combination of switch and break does the trick. Assuming that a character other than 0 to 9 is entered, none of the cases match. Therefore, default is executed. The program will print 'illegal character'. The default is optional and even in its absence, when no match is found, the program will come out of the switch statement without any action. The case statements need not be in any specified order. The default can be on top before case 0 and the case can occur in any order. The program is made to execute, as long as you want, by the do statement. If do is absent, then the program will be executed only once. The do...while is necessary to make it an iterative program. Every switch statement, therefore, contains a condition in the form of an expression. The expression could also be a single variable as in this case. The expression will be evaluated at the time of program execution and must be an integer. Then depending on the value it goes to a case label,
which is like a label in a goto statement. The label should match with the value of the expression. Unless the program is made to exit by statements such as break after executing the group of statements corresponding to a particular case, the program will execute all the statements in the program from then on. This should be noted. Therefore, the programmer has to specify where to end. In the case of if...else, where to begin and where to end is clear as also in the case of switch. The program starts at the beginning of the case that meets the condition, but ends at the bottom of the switch
Self-Instructional Material 113 Control Statements NOTES unless otherwise specified. It will also execute the statements following default. It will of course not bother about the keywords. That is the reason for the break statement, since we do not want the program to execute irrelevant statements. It is a good programming practice to include a break statement after the default as well. If this is not done at the inception, at a later stage when more case statements are added after default, this would lead to problems. Whenever default is executed all the statements following it, even if they belong to some other case, will also be executed if break is not included after default. Now the program can be extended to print the value of the number up to 99 in words. This makes it more complicated since the words are unique up to nineteen. A program is given below for achieving the task. This is made to loop using do...while.

```
/*Example 5.17 to print out in words the value of a number typed in the range 1-99*/ #
include &gt;stdio.h&lt; #include&gt;conio.h&lt; main() { int num,m,ch='y'; do { printf("Type a number 1 to 99\n"); scanf("%d",&num); if
((num &lt;0)&&(num &gt;100)) /*
only if the number is within the valid range 0 to 99 the following will be executed*/ { if (num &lt;= 20) { m=num/10;
switch(m) {
case 2:printf("TWENTY "); break; case 3:printf("THIRTY "); break; case 4:printf("FORTY "); break; case 5:printf("FIFTY "); break; case
6:printf("SIXTY "); break; case 7:printf("
SEVENTY "); break; case 8:printf("
EIGHTY ");
114 Self-Instructional Material Control Statements NOTES break; case 9:printf("NINETY "); break; } } if (num &lt;20) num=
num%10;
switch(num) {
case 1:printf("ONE\n");
break; case 2:printf("TWO\n"); break; case 3:printf("THREE\n"); break;
case 4:printf("
FOUR\n"); break; case 5:printf("
FIVE\n"); break; case 6:printf("
SIX\n"); break; case 7:printf("SEVEN\n");
break;
case 8:
printf("
EIGHT\n");
break;
case 9:printf("NINE\n"); break; case 10:printf("TEN\n"); break; case 11:printf("ELEVEN\n"); break; case 12:printf("TWELVE\n"); break; case
13:printf("
THIRTEEN\n");
break;
case 14:printf("
FOURTEEN\n");
break; case 15:printf("
FIFTEEN\n"); break; case 16:printf("SIXTEEN\n"); break; case 17:printf("SEVENTEEN\n"); break; case 18:printf("
EIGHTEEN\n");
```

Self-Instructional Material 115 Control Statements NOTES break; case 19:printf("NINETEEN\n"); break; } } else printf("number outside range\n"); printf("\nenter y if you want to continue\n"); ch=getche(); if (ch!='y') printf("End of session"); } while (ch=='y'); } Result of the program Type a number 1 to 99 123 number outside range enter y if you want to continue yType a number 1 to 99 78 SEVENTY EIGHT enter y if you want to continue yType a number 1 to 99 6 SIX enter y if you want to continue nEnd of session 5.6
BREAK, CONTINUE, RETURN The keywords while, for and switch test the condition on top, while do...while checks at the bottom for quitting the loop. The break statement helps immediate exit from any part of the loop as demonstrated with the switch statement. It can be used with any other loop construct or anywhere in the program. When the break statement is executed it goes to the bottom of the block. Recall that a block is a group of statements enclosed between an opening brace and the corresponding closing brace. The continue statement is related to break. When continue is executed,
it causes the next iteration of the corresponding for, do...while or while loop to begin.
Therefore, continue takes the program to the top of the block and in the for loop, it will cause the next increment operation, followed by checking whether the condition is true or false in order to decide the next course of action. This is similar to skipping the current execution and continuing with the next operation after incrementing. The statement continue skips the rest of the statements in the loop for that iteration, whereas break terminates the loop.
Check Your Progress 9. Define search. When is it called linear search? 10. Write the syntax for switch statement. 11. How does the switch keyword function? 12. What does the break statement do?
116 Self-Instructional Material Control Statements
NOTES
Write a program to check whether a given number is positive or
negative. If it is zero, the program should terminate after printing the value. If it is a positive integer above zero and &gt;=20000, the value will be printed; if negative, it will go to fetch the next number. If the number is &lt;20000, the program terminates. The program is given below: /*Example 5.18 /*program to demonstrate continue*/ #
include &gt;stdio.h&lt; main() { int a; do { printf("enter a number-enter 0 to end session\n"); scanf("%d", &a); if(a &lt; 20000) { printf(" you entered a high value-going out of range\n"); break; } else if(a&lt;=0) printf("you entered %d\n", a); if (a &gt;0) { printf("you entered a negative number\n"); continue; } } while(a !=0); printf(" End of session\n"); } Result of the program enter a number-enter 0 to end session 33 you entered 33 enter a number-enter 0 to end session -60 you entered a negative number enter a number-enter 0 to end session 45 you entered 45 enter a number-enter 0 to end session
Self-Instructional Material 117 Control Statements NOTES 25000 you entered a high value-going out of range End of session If the number typed &lt; 20000, or if it is equal to zero, the program comes out of the loop and prints "End of session". If the number is negative, a &gt; 0 and hence, continue will be executed. It will go to the top of the loop. The next integer will be received. The program, therefore, terminates when a = 0 as well as a &lt; 20000, but there is a difference. If the number entered is zero, the program checks whether a &lt; 20000. Since the condition fails, it checks whether a &lt; = 0 and since it is true, 0 will be printed and then the while condition is checked. The program terminates after the while condition is checked. However, if the number entered is &lt; 20000, the loop terminates instantly without transacting any business except printing messages as shown. The return statement can appear anywhere in a function and when it is encountered a value is returned to the called function. The return may also not return a value in statements as shown: return ; return (0) ; The return statement may appear anywhere in the function and not necessarily at the end of the function. Whenever return is executed, the program returns to the function called the current function. The program returns to the place from where it called the function. Thus return is also used to suddenly exit from a function or a loop in a function.
Exit Function
There is a library function exit(), which causes the termination of the current program. Note that, exit() terminates the execution of the program itself, and not the block. The statement break enables coming out of the block or loop in which it is executed but exit terminates the program at whatever stage the program may be. The exit is a powerful function. 5.7
SIGNIFICANCE OF THE COMMA You can use comma to declare more than one data type. For example, you may code without using the comma as given below: int a; int b; Instead, you can write, int a, b; You have combined two statements into one, and can similarly combine the following three declarations into one: int a = 10; int b = 0; int c = 20; Alternatively, you can write, int a = 10, b = 0, c = 20; When you combine the same data type there is a trivial use of the comma, as you have done over a number of programs. The more important application is in the for statement. Check Your Progress 13.
How are continue and break statements related? 14. Define the exit() function. 15.
Why is comma used in declaring a C program?
118 Self-Instructional Material Control Statements NOTES Usually you write, for (exp1; exp2; exp3) s1; You can give two expressions instead of one by using a comma. For example, for(exp1, exp2; exp3, exp4; exp5) sl; Consider the following example, for(i=0, j=n; i&gt;=p, j&lt;i; i++, j—) sl; Here s1 will be executed with i=0 and j=n as initial values. After each iteration i will be incremented and j will be decremented. Here s1 will be executed till i&gt;=p and j&lt;i. Even if one of the conditions is false, the program will come out of the loop. 5.8 SUMMARY In this unit, you have learned that the
if...else is a useful construct when different statements or groups of statements are to be executed depending upon a condition. The else
follows if.
if...else can be nested to multiple levels. However, as more and more nesting is carried out, it becomes difficult to follow what is going on. More nesting increases the chances of logical mistakes.
However,
the usage of braces with indentation reduces errors and improves readability. Before attempting to write a program, the algorithm has to be finalized stepwise. This also improves the quality of the program.

The usage of relational operators and logical operators &&, || and ! helps to solve even complicated problems. The ? operator permits writing of the if...else statement in one line. Finally, before you leave the program, see that each else has a matching if preceding it and each closing brace } has a matching opening brace {. This will be a good checklist.
if...else is suitable for simple programs, but there are better constructs such as switch... case, which are suitable for more complex problems. However, if...
else remains
an important building block in 'C' language programming. Repetitive operations known either as loops or iterations, are quite common in scientific applications. Implementing recursion with the if...else statement needs the use of the goto statement and the corresponding label.
The label name can be constructed in a similar way to a variable name. Any label name ends with a colon and a label can appear anywhere in the function.
However, goto statements reduce the readability of programs and are likely to cause errors. Therefore, goto should be avoided as much as possible.
The for and while are loop constructs. They help in constructing loops without labels.
The while loop will be executed so long as the associated condition is true. The for loop is an improvement of the while loop containing three expressions on the header of the statement. The condition appears in the middle followed by a semicolon. The first expression usually initializes a variable and the last expression increments it.
With the initial value, the condition is checked up, and if true the loop is executed once, i.e., the statements following the Error! Bookmark not defined.for condition and enclosed within the braces { } are executed once, after which the third expression is stepped up according to the need. Then after checking the condition again, the loop is executed. This continues while the condition is satisfied and stops thereafter. The
for and while loops should be constructed so that they terminate after execution; otherwise, an endless loop will be made, which will halt the computer.
The loops could also be nested depending on the need. The initial value, the final value, the step, etc. could be generalized and the actual value can be declared in
Self-Instructional Material 119 Control Statements NOTES the beginning of the program by symbolic constants using the #define statement. This will help in trying out the same program with a different set of values later. Symbolic constant names are written in upper case letters to distinguish them easily. At the time of compilation of the program, the symbolic constants are replaced with actual values at all the places where they appear. The unit discussed a variation of the while loop which
is the do...while loop, which checks the condition after the execution of the loop once. Therefore, do...while will execute the body of the loop at least once. This is because the do...while loop starts with do and has the condition checked only at the end of the loop. The loop body starts after do and ends just before the while.
The block is marked by opening and closing braces. Note that the statement while (condition) has to be terminated with semicolon in this case in contrast with the while loop. The fourth type of control construct is
the switch ... case. It is a multiway branching construct. The switch is followed by an expression in parentheses, which
will get
evaluated as an integer at the time of program execution. The statements following the switch begin with case labels where each case label has a unique integer like case 10, case 9, etc. Therefore, the program after executing the switch goes to the label corresponding to the switch expression. The program begins execution of all the statements
following the case till the end of the switch block. Quite often, different statements are executed corresponding to each case. Therefore, at the end of the group of statements following the case, a Error! Bookmark not defined.break statement is usually included, which indicates the end of the case block.
At the end of all the case statements, a default statement can be included, which will also contain a group of statements. If the switch expression does not match with any of the case expressions, then the default statements will be executed. A switch statement need not have a default statement and in such a case the program will come out of the switch if no match is found.
The Error! Bookmark not defined.break statement is useful not only with the switch statement but also in any loop. If an Error! Bookmark not defined.break statement is encountered, the program will exit the loop instantly. There is also a continue statement, which
serves to skip the rest of the statements in the loop and execute the loop after incrementing or decrementing and checking the condition.
Therefore, continue takes the program to the beginning of the loop, whereas break brings the program out of
the loop. The return statements enable the function to instantly return the program execution to the called function.
There is a library function exit(), which when executed terminates the program immediately. These statements along with label and goto permit branching of program at the programmer's will. 5.9
KEY TERMS • Loop or iteration: It is used to perform the same operation a number of times by repeating the same operation with one or more of the values changed. •
Symbolic constants: These are written in capital or upper case letters. Wherever the symbolic constant names appear in a program, the compiler will replace them with the corresponding replacement constants defined in the # define
compiler directive.
120 Self-Instructional Material Control Statements NOTES • Linear search: It is the process of finding out whether a given number or string is present in an array of data. If the item to be searched matches with an item in the array of data, then the search is stopped; otherwise, the search is continued till the end of the array. • Switch:

When the switch keyword is encountered, the associated expression is evaluated. The program looks for the case, which matches with the value of the expression.
Each case has to be accompanied by integer expressions, which must be unique, as otherwise the program will not know where to start. 5.10
ANSWERS TO 'CHECK YOUR PROGRESS' 1. The syntax of the if statement is given below. if (condition) {statements} If the condition is true, then a single statement or group of statements following the if will be executed. If more than one statement is to be executed, then the statements are grouped within braces. If it is a single statement, then curly braces are not required. 2. 'C' provides three logical operators for combining more than one condition. These are as follows: Logical and represented as && Logical or represented as || Negation or not represented as ! (exclamation). 3.
If you
subtract 32 from the ASCII value of a lower case alphabet we will get the ASCII value of the corresponding upper case letter. 4.
Usage of goto is considered to be bad programming practice since it leads to errors when changes are made in the program and also affects readability. It is always possible to write a program without using goto. The program can be rewritten without goto by using a for statement. 5.
The
for statement is meant for the easy implementation of iterations unlike if. The syntax of for is given below: for (exp1; exp2; exp3) {statements;} 6.
The
symbolic constants are to be written in capital or upper case letters. Wherever the symbolic constant names appear in a program, the compiler will replace them with the corresponding replacement constants defined in the # define statement. 7.
The while loop is a subset of the for loop. The syntax for the while loop is given below: while (expression) {statements;} 8.
The while loop tests the condition on top of the loop; but do...while tests at the bottom after executing the statements. The while loop executes the statements after checking the condition; but do...while executes the
Self-Instructional Material 121 Control Statements NOTES statements before testing the condition. The syntax of do...while is as given below: do { statements } while (expression); The statement do...while is not used as frequently as the while
loop. 9.
Search is the process of finding out whether a given number or string is present in an array of data. If the item to be searched matches with an item in the array of data, then the search is stopped; otherwise, the search is continued till the end of the array. This method of searching is called linear search. 10.
The syntax of the switch statement is given below: switch (expression) { case constant or expression : statements case constant or expression : statements .. default : statements } 11. When the switch keyword is encountered, the associated expression is evaluated. The program now looks for the case, which matches with the value of the expression. Execution then starts from the statement corresponding to the case which matches. Each case has to be accompanied by integer expressions, which must be unique, as otherwise the program will not know where to start.
There may be occasions when none of the constant expressions matches the switch expression in which case the default statements will be executed. 12.
The
break statement takes the program to the end of the switch statement. The end is just the closing brace corresponding to switch after the default printf() statement. 13. The
continue
statement is related to break. When continue is
executed,
it causes the next iteration of the corresponding for, do...while or while loop to begin.
Therefore, continue takes the program to the top of the block and in the for loop, it will cause the next increment operation, followed by checking whether the condition is true or false in order to decide the next course of action. This is similar to skipping the current execution and continuing with the next operation after incrementing. The statement continue skips the rest of the statements in the loop for that iteration, whereas break terminates the loop. 14. There is a library function exit(), which causes the termination of the current program. Note that, exit() terminates the execution of the program itself, and not the block. The statement break enables coming out of the block or loop in which it is executed but exit terminates the program at whatever stage the program may be. The exit is a powerful function. 15.
We can use comma to declare more than one data type. For example, we may code without using the comma as given below: int a; int b;
122 Self-Instructional Material Control Statements NOTES 5.11 QUESTIONS AND EXERCISES Short-Answer Questions 1. Specify the output of the following programs: (a) #

| 61% | MATCHING BLOCK 26/126 | SA | C_book_AmrutaJog_ShrutiJathar.pdf (D54111295) |
|---|---|---|---|

include&gt;stdio.h&lt; #include&gt;conio.h&lt; main() { int ch; ch=getch(); if(ch&lt;='A' && ch&gt;='Z') { ch=

ch-'A'+'a'; putch(ch); } else printf("enter value again"); } (b) #

| 84% | MATCHING BLOCK 25/126 | W | |
|---|---|---|---|

include&gt;stdio.h&lt; main() { int a, b; scanf ("%d %d", &a, &b); if ( (a+b) &gt;100) printf( "

you
have entered small numbers"); else printf("well done"); } 2. State whether true or false and state reason: (a) Usage of labels is a bad programming practice. (b) #define VALUE 6; is correct. (c) A for statement can be formed with 2 expressions on the header. (d) Any of the loops can be used at will. (e) The while statement and do...while statement are identical in performance. (f) switch (exp) ends with a semicolon. (g) The break is a function. (h) The continue statement takes the program to the top of the loop. 3. How many times will the following loops be executed ? (a) for (a = 0 ; a &gt; 14 ; a++) { s1 ; }
Self-Instructional Material 123 Control Statements NOTES (b) for ( a = 35 ; a &lt; 20 ; a—) { s2 ; } (c) for ( a =35 ; a &gt; = 20 ; a—) { s3; } Long-Answer Questions 1. (a) A Lucas sequence is given below : 1, 3, 4, 7, 11, 18, 29 The third number is the sum of the previous two numbers. Write an algorithm and a program to print out the first ten numbers of a Lucas sequence. (b) Assuming that in a sequence, the next number is obtained by adding the last four numbers repeatedly, write a program to generate the first 20 numbers of the sequence given that the first four numbers are 0, 0, 1, and 1. 2. Write a program that will print the natural numbers which are evenly divisible by, a) 11 b) 13 c) 19 d) 23 e) 37 3.

| 68% | MATCHING BLOCK 27/126 | SA | Block 3 and 4.pdf (D129191893) |
|---|---|---|---|

Write a program to find the factorial of a given number using while. 4. Write a program,

which will determine whether the root of a quadratic equation is real or imaginary using switch. 5. Rewrite the program for the generation of mathematical tables from 1 to 10 using the switch statement. 6. An array of integers is declared. The program should print each number in the array and its position till the end of array is encountered. 7. Modify the program to arrange the array in the ascending order, writing the value and position. 8.

| 52% | MATCHING BLOCK 29/126 | SA | Dr. Baldev SIngh LKC_Programming in C.pdf (D137918641) |
|---|---|---|---|

Write a program to find the factorial of 10. 9. Write a program to find the numbers in any range, which are divisible by

any divisor such as 11. 10. Write one program with the while loop and one program with the for loop for linear search. 5.12 FURTHER READING Gottfried, Byron S. Programming with C, 2nd edition. Schaum's Outline Series. New York: McGraw-Hill, 1996. Forouzan, Behrouz A. and Richard F. Gilberg. Computer Science: A Structured Programming Approach Using C, 2nd edition. California: Thomson, Brooks Cole, 2000. Dey, Pradip and Manas Ghosh. Computer Fundamentals and Programming in C. New Delhi: Oxford Higher Education, 2006. Bronson, Gary J. A First Book of ANSI C, 3rd edition. California: Thomson, Brooks Cole, 2000. Kanetkar, Yashwant. Understanding Pointers in C.
New Delhi: BPB Publication, 2001. Kanetkar, Yashwant. Let us C. New Delhi: BPB Publication, 1999.
124
Self-Instructional Material Control Statements NOTES Kernighan, Brian W. and Dennis Ritchie. C Programming Language, 2nd edition. New Jersey: Prentice Hall, 1988. Foster, W. D. and L. S. Foster. C by Discovery. Boston: Addison-Wesley, 2005. Kanetkar, Yashwant. Working with C. New Delhi: BPB Publication, 2003. Horton, Ivor. Instant C Programming. New Jersey: Wrox Press (John Willey & Sons), 1995. Lawlor, Steven C. The Art of Programming Computer Science with 'C'. New Jersey: West Publishing Company, 1996.
MODULE – III
126

| 47% | MATCHING BLOCK 28/126 | W | |
|---|---|---|---|

Self-Instructional Material Functions NOTES Self-Instructional Material 127 Functions NOTES UNIT 6 FUNCTIONS Structure 6.0 Introduction 6.1 Unit Objectives 6.2 Modular Programming Overview 6.3 Function Prototypes 6.4 Function Call – Passing Arguments to a Function 6.4.1 Function Arguments 6.5 Function Definition 6.6 Scope Rules for

Function 6.7 Library Functions 6.8 Return Values 6.9
Recursion 6.10 Implementation of Euclid's gcd Algorithm 6.11 Summary 6.12 Key Terms 6.13
Answers to 'Check Your Progress' 6.14 Questions and Exercises 6.15 Further Reading 6.0 INTRODUCTION In this unit, you will learn about the functions used in writing programs

in ‘C’ to illustrate the fundamental concepts of data structures and algorithms. In practice, you have to write programs to solve real-life situations taking into account all the features of the language. Such programs should be free from logical errors and program or code errors. Earlier, people wrote larger programs sequentially using all the constructs of a language. This led to difficulties in debugging and even understanding. It was then that experts felt methodologies should be evolved to improve the quality of programs. One of the recommendations was modular programming. A module can be a single function or a group of related functions carrying out a specific task. Since a programmer will not be able to comprehend the meaning of a program of more than 100 lines, one of the features of modular programming is to divide the program into smaller modules or functions, which will execute a particular task. In a program, the different modules are properly linked. If a program is developed in this modular way, it will become easy to divide and conquer the problem. This is the special feature of structured programming. Hence, the main objective of structured programming is to plan and develop a program as a collection of modules. ‘C’ language was developed for achieving this objective easily. Therefore, structured programming also means that the execution of statements progresses linearly. ‘C’ contains a number of library functions. In order to use the library functions, you must include the header files supplied with the system such as stdio.h, string.h, etc. The library functions are encapsulated. You only pass arguments to use them. The arguments are put within brackets following printf(), scanf(), gets(), puts(), etc. The availability of such functions is a feature of modular programming. This would increase the length of the code, affect readability and increase complexity. Hence, you must follow the modular programming concept.
128 Self-Instructional Material Functions NOTES In this unit, you will learn that
a function calling itself is called recursion and the function may call itself either directly or indirectly.
This concept is explained with the help of examples.
Thus, recursion keeps the program size small, but understanding recursion is not easy. If the program can be visualized as recursive, it will result in
a
compact code. Recursive functions can easily become infinite loops. Therefore, the functions should have a statement with if so that the program will definitely terminate.
You will also note that recursive programs simulate the use of stack. 6.1
UNIT
OBJECTIVES After going through this unit, you will be able to: •
Understand modular programming • Explain the general forms of functions • Define function prototype • Describe function arguments • Understand scope rules for functions • Comprehend library functions and return values • Understand the basic concept of recursion • Analyse recursive quick sort • Use Euclid’s algorithm 6.2 MODULAR PROGRAMMING OVERVIEW You have been writing small programs in ‘C’ to illustrate the fundamental concepts of data structures and algorithms. In practice, you have to write programs to solve real-life situations taking into account all the features of the language. In fact, ‘C’ is used for developing operating systems and other system software. Such programs should be free from logical errors and program or code errors. In the olden days, people wrote larger programs sequentially using all the constructs of a language. This led to difficulties in debugging and even understanding. As no technique for good programming was available, an effective working program was suitable enough. However, this led to a software crisis in the sixties. It was then that experts felt methodologies should be evolved to improve the quality of programs. One of the recommendations was modular programming. A module can be a single function or a group of related functions carrying out a specific task. Since a programmer will not be able to comprehend the meaning of a program of more than 100 lines, one of the features of modular programming is to divide the program into smaller modules or functions, which will execute a particular task. A program consists of a number of modules properly linked. This enables the programmer to analyse each such small block and then draw the bigger picture containing all the blocks to know exactly what the program is doing. If a program is developed in this modular way, it will become easy to divide and conquer the problem. This is one of the features of structured programming. The main objective of structured programming is to plan and develop a program as a collection of modules. ‘C’ language was developed for achieving this objective easily. Therefore, structured programming also means that the execution of statements progresses linearly. There is no back and forth traversal while executing the program.
Self-Instructional Material 129 Functions NOTES The program uses a single entry–single exit pattern. The label and goto statements affect linear programming and hence, are discouraged. The while, for, do- while, switch, etc. statements enable structured programming. These loop statements do a particular predefined task and hence, can be independently checked for their correctness. Structured programming results in understandable programs. ‘C’ contains a number of library functions. You have already used many of them. They are as follows: printf( ) scanf( ) getchar( ) putchar( ) gets( ) puts( ) strlen( ) In order to use the library functions, you must include the header files supplied with the system such as stdio.h, string.h, etc. In effect, the library functions are hiding the programs or in other words, they are encapsulated. You only pass arguments to use them. The arguments are, for example, what you put within brackets following printf(), scanf(), gets(), puts(), etc. The availability of such functions is a feature of modular programming. The availability of these functions has reduced considerably the need to code the print or other statements and check them in every program. In the top-down programming method, the main function calls other specialized functions for carrying out a specialized function such as strlen( ). But for the availability of this function, you would have to write a routine whenever you want to find the length of a string as part of the main function. This would increase the length of the code, affect readability and increase complexity. Therefore, it would be better to follow the modular programming concept. This is illustrated in the following figure: MAIN

| 95% | MATCHING BLOCK 34/126 | SA | C book final copy (1).doc (D31388791) |
|---|---|---|---|

f 1 f 2 f 3 f 4 f 11 f 12 f 21 f 31 f 32 f 41 f 42 f 43 Fig. 6.1 Modular Programming Concept Develop the program in such a manner that the main program calls a number of functions for carrying out specific tasks. Each function may, in turn, call other specialized functions wherever required. Such a program will be easy to understand, debug and maintain. In order to improve the quality of programming, you should, therefore, call functions wherever required. The functions supplied with the system are the library functions. User-developed functions are called user-defined functions. You now know that each function performs a specified task. Arguments or data are passed to the function and the function performs some specified actions. Before you go into more details, you must understand the usage of functions. Assuming that you want to reverse a number in a program, you split the program into two parts. The

130 Self-Instructional Material Functions NOTES main() function gets the number to be reversed. It passes the number to a function called reverse() whose specialized and only job is to get the number, reverse it and send back the reversed number to the function, which calls it. The diagrammatic representation is as follows: main reverse The main() function passes on a number to reverse whenever it wants to reverse it. The reverse() function reverses the number and sends it back to the main() function. Thus, the task is perfectly partitioned with perfect understanding and protocol. Later on, if some other function wants to reverse a number, it can bank upon the capability of already tested reverse() function and use it.

General Form of Function A function consists of three parts: • Function Prototype • Function Definition • Function Call 6.3 FUNCTION PROTOTYPES

A function prototype is also called function declaration. A function may be declared at the beginning of the main function.

The

function declaration is of the following type: return data type function name (formal argument 1, argument 2, ............ ); A function, after execution, may return a value to the function, which called it. It may not return a value at all but may perform some operations instead. It may return an integer,

a character or a

float. If it returns a float you may declare the function as, float f1(float arg 1, int arg 2); If it does not return any value, you may write the above as, void fun2(float arg1, int arg2); /*void means nothing*/. If it returns a character, you may write, char fun3(float arg1, int arg2); If no arguments are passed into

a function, an empty pair of parentheses must follow the function name.

For example, char fun4( ); The arguments declared as part of the prototype are also known as formal parameters. The formal arguments indicate the type of data to be transferred from the calling function.

This is about the function declaration. Check Your Progress 1. What is a module? 2. What is the main feature of modular programming? 3. Write the parts of a function.

Self-Instructional Material 131 Functions NOTES 6.4

FUNCTION CALL – PASSING ARGUMENTS TO A FUNCTION You may call a function either directly or indirectly. When you call the function, you pass actual arguments or values. Calling a function is also known as function reference. There must be a one-to-one correspondence between

the

formal arguments declared and the actual arguments sent. They should be of the same data type and in the same order. sum = f1 (20.5, 10); fun4( ); 6.4.1 Function

Arguments You know now that an argument is the

parameter or value. It could be of any of the valid types such as all forms of integers or a float or char. You come across two types of arguments when you deal with functions: • Formal arguments • Actual arguments

The

formal arguments are defined in the function declaration in the calling function. What is an actual argument?

The

data, which is passed from the calling function to the called function, is called actual argument. The actual arguments are passed to the called function through a function call. Each actual argument supplied by the calling function should correspond to the formal arguments in the same order. The new ANSI standard permits the declaration of the data types within function declaration to be followed by the argument name. You have used only this type of declaration as it will help

the students to

follow C++ program easily. This helps in understanding one-to-one correspondence between the actual arguments supplied and those received in the function, and facilitates the compiler to verify that one-to-one correspondence exists and that the right number of parameters have been passed. It may be noted that

the formal arguments cannot be used for any other purposes. It

only gives a prototype for the function. Thus, the names of the formal arguments are dummy and will not be recognized elsewhere, even in the function in which it is defined. Although the types of variables in the function declaration, also known as prototype, and function call are same, the names need not be the same. You have already used this concept in Example 6.2 after defining float

a and

float b in the functions prototype; you first called add (a, b), add (c, d) and then add (sum1, sum2). Thus the formal arguments defined in the prototype and the actual arguments were not the same in two of the above cases. When the actual arguments are passed to a function, the function notes the order in which they are received and appropriately stores them in different locations. You must note that even if you use a and b in the add function, they will be stored in different locations. They will have no relationship with a and b of the main function. Therefore, even if a and b are assigned different values in the called function, the corresponding values in the calling function would not have changed.

Check Your Progress 4. Define function prototype. 5. How is a function called? 6. What is an argument? What are its types? 7. Explain formal and actual arguments.
132 Self-Instructional Material Functions NOTES 6.5 FUNCTION DEFINITION The
function definition can be written anywhere in the file with a proper declarator followed by the declaration of local variables and statements.
The

**94%** **MATCHING BLOCK 30/126** **W**

function definition consists of two parts, namely function declarator or heading and function functions. The function heading is similar to function declaration, but will not terminate with a semicolon. The use of functions will be demonstrated with simple programs in this unit. Assume that you wish to get two integers. Pass them to a function add. Add them in the add function. Return the value to the main function and print it. The algorithm for solving the problem is as follows: Main Function Step 1: define function add Step 2: get 2 integers Step 3: call add and pass the 2 values Step 4: get the sum Step 5: print the value function add Step 1: get the value Step 2: add them Step 3: return the value to main Thus you have divided the problem. The program is given below: /*Example 6.1*/ /* use of function*/ #include &gt;stdio.h&lt; int main() { int a=0, b=0, sum=0; int add(int a, int b); /*function declaration*/ printf("enter 2 integers\n"); scanf("%d%d", &a, &b); sum =add(a, b); /*function call*/ printf("sum of %d and %d =%d", a, b, sum); } /*function definition*/ int add (int c, int d) /*function declarator*/ { int e; e= c+d; return e; }

Self-Instructional Material 133 Functions NOTES
Result of the program enter 2 integers 6667 4445 sum of 6667 and 4445 =11112 The explanation as to how the program works is given below: On the fifth statement (seventh line), the declaration of the function add is given. Note that the function will return an integer. Hence, the return type is defined as int. The formal arguments are defined as int a and int b. The function name is add. You cannot use a variable without declaring it as also a function without telling the compiler about it. Note also that function declaration ends with a semicolon, similar to the declaration of any other variable. Function declaration should appear
at the beginning of the calling function. It hints to the compiler that the function is going to call the function add later in the program. If the
calling function is not going to pass any arguments, then empty parentheses are to be written after the function name. The parentheses must be present in the function declaration. This happens when the function is called to perform an operation without passing arguments. In this case, if a and b are part of the called function add itself, then we need not pass any parameters. In such a case, the function declaration will be as follows assuming that the called function returns an integer: int add ( ) ; In Example 6.1, you get the values of a and b. After that you call the function add and assign the value returned by the function to an already defined int variable sum as given below: sum = add ( a , b ); Note that add(a, b) is the function call or function reference. Here, the return type is not to be given. The type of the arguments are also not to be given. It is a simple statement without all the elements of the function declaration. However, the function name and the names of the arguments passed, if any, should be present in the function call. When the program sees a function reference or function call, it looks for and calls the function and transfers the arguments. The function definition consists of two parts, i.e., the function declarator and function body. The function declarator is a replica of the function declaration. The only difference is that the declaration in the calling function will end with a semicolon
and
the declarator in the called function will not end with a semicolon. As in main(), the entire functions body will be enclosed within braces. The whole function can be assumed to be one program statement. This means, that all the statements within the body will be executed one after another before the program execution returns to the place in the main function from where it was called. The important points to be noted are: • The declarator must agree totally with the declaration in the called function, i.e., the return data type, the function name, the argument type should all appear in the same order. The declarator will not end with a semicolon. • You can also give the same name as in the calling function—in the declaration statement or
the
function call—or different names to the arguments in the function declarator. Here, you have given the names c and d. What is important, however, is that the type of arguments should appear as it is in
134
Self-Instructional Material Functions NOTES

**99%** **MATCHING BLOCK 31/126** W

the declaration in the calling program. They must also appear in the same order. • At the time of execution, when the function encounters the closing brace }, it returns control to the calling program and returns to the same place at which the function was called. In this program, you have a specific statement return (e) before the closing brace. Therefore, the program will go back to the main function with value of e. This value will be substituted as, sum = (returned value) Therefore, sum gets the value, which is printed in the next statement. This is how the function works. Assume now that the program gets a and b values, gets their sum1, gets c and d and gets their sum2 and then both the sums are passed to the function to get their total. The program for doing this is as follows: /*Example 6.2*/ /* A function called many times */ #include &gt;stdio.h&lt; int main() { float a, b, c, d, sum1, sum2, sum3; float add(float a, float b); /*function declaration*/ printf("enter 2 float numbers\n"); scanf("%f%f", &a, &b); sum1 =add(a, b); /*function call*/ printf("enter 2 more float numbers\n"); scanf("%f%f", &c, &d); sum2 =add(c, d); /*function call*/ sum3 =add(sum1, sum2); /*function call*/ printf("sum of %f and %f =%f\n", a, b, sum1); printf("sum of %f and %f =%f\n", c, d, sum2); printf("sum of %f and %f =%f\n", sum1,sum2, sum3); } /*function definition*/ float add (float c, float d) /*function declarator*/ { float e; e = c+d; return e; } Result of the program enter 2 float numbers 1.5 3.7 enter 2 more float numbers 5.6 8.9 sum of 1.500000 and 3.700000 =5.200000

Self-Instructional Material 135 Functions NOTES
sum of 5.600000 and 8.900000 =14.500000 sum of 5.200000 and 14.500000 =19.70000 You have defined sum1, sum2 and sum3 as float variables. You are calling function add three times with the following assignment statements: sum1 = add( a, b ); sum2 = add( c, d); sum3 = add( sum1 , sum2 ); Thus the program goes back and forth between main and add as
follows:
main() add (a, b) main() add (c, d) main() add (sum 1, sum 2) main() Had you not used the function add, you would have to write statements pertaining to add 3 times in the main program. Such a program would be large and difficult to read. In this method you have to code for add only once and hence, the program size is small. This is one of the reasons for the usage of functions. In Example 6.2, We could add another function call by add (10.005, 3.1125); This statement will also work perfectly. After the function is executed, the sum will be returned to the main function. Therefore, both variables and constants can be passed to a function
by making
use of the same function declaration. You have seen a program, which calls a function thrice. Now a problem, which calls three functions
will be discussed.
Problem The user gives a four-digit number. If the number is odd, then the number has to be reversed. If it is even, then the number is to be doubled. If it is evenly divisible by three, then the digits are to be added. Now write the algorithm for solving the problem. Step 1: Get the number. Step 2: If number is odd, call the reverse function. Step 3: Else multiply the number by 2 and hence, call multiply. Step 4: If number is evenly divisible by 3, call add-digits. These are the steps. The first one, namely writing a function to multiply by 2 is simple. You will look at the other two steps now. Reverse
You have already seen reversing the number in a program. You will use the same steps.
Step 1: reverse = 0 Step 2: while ( number &lt; 0 ) reverse = reverse * 10 + (number % 10 ) number = number/10 Step 3: return (reverse) 136
Self-Instructional Material Functions NOTES

**89%** **MATCHING BLOCK 32/126** W

ADD digits function Step 1: sum = 0 Step 2: while number &lt; 0 sum = sum + (number % 10) number = number / 10 Step 3: return (sum) See how the above algorithm adds digits

and works.

| 95% | MATCHING BLOCK 33/126 | W |
|---|---|---|

Give 4321 as the number. Step 1: sum = 0 Step 2: Iteration 1 sum = 0 + modulus of (4321/10) = 0 + 1 = 1 number = 4321/10 = 432 Iteration 2 sum = 1 + modulus of (432/10) = 1 + 2 After 4 iterations, sum =1 + 2 + 3 + 4 Step 3: Sum is returned. The program is given below: /*Example 6.3*/ /*program to demonstrate calling multiple functions*/ #include&gt;stdio.h&lt; int main() { long nummul=0; long num=0, rev=0, add_digit=0; /*good practice to initialize all variables*/ long reverse(long num); long mult(long num); int sum_digit(long num); printf("enter unsigned number\n"); scanf("%lu", &num); if (num%2) /*remainder 1*/ { rev = reverse(num); printf("number is odd\n"); printf("number entered=%lu\n number reversed=%lu\n", num, rev); } else { Self-Instructional Material 137 Functions NOTES nummul=mult(num); printf("number is even\n"); printf("number=%lu\n its multiple=%lu\n", num, nummul); } if (num%3 ==0) { add_digit= sum_digit(num); printf("number evenly divisible by 3\n"); printf("sum of digits =%lu", add_digit); } } long reverse(long n) { long r=0; while (n&lt;0) { r=r*10+(n%10); n=n/10; } return r; } long mult(long p) { long sq; sq=2*p; return sq; } int sum_digit(long num) { long sum=0; while (num &lt;0) { sum=sum+(num%10); num=num/10; } return sum; } Result of the program enter unsigned number 4321 number is odd number entered=4321 number reversed=1234 138 Self-Instructional Material Functions NOTES Look at the program. After getting the number from the user, it evaluates if remainder of (num/2) = true; i.e., if the remainder is 1, then it is true. If remainder = 1, then the number is odd and hence, the reverse function is called. The returned value is assigned to rev and printed. If the number is even, the number is doubled. Since the doubled value may exceed the maximum of the unsigned integer, we have declared it as a long integer. Next we check if the number is evenly divisible by 3. If it is so then we add the digits. Thus, the main function of Example 6.3 calls three functions for carrying out specific tasks. All the three functions are supplied with the same arguments, but return different values. 6.6

SCOPE RULES FOR FUNCTION

| 98% | MATCHING BLOCK 35/126 | W |
|---|---|---|

The scope of the variable is local to the function unless it is a global variable. For example, int function1(int I ) { int j=100; double function2 (int j) ; function2 (j) ; } double function2 (int p) { double m; return m; } The variable j in function1 is not known to function2. You pass it to function2 through the argument j. This will be assigned as equal to int p. Similarly, m in function2 is not known to function1. It can be made known to function1 through the return statement. This makes the scope rules of variables in function quite clear. The scope of variables is local to the function where defined. However, global variables are accessible by all functions in the program if they are defined above all functions. /*Example 6.4*/ /* to demonstrate that the scope of a variable is local to the function*/ #include &gt;stdio.h&lt; int main() { float a=100.250, b=200.50; void change (float a, float b); change(a, b); printf("a= %f b= %f\n ", a, b); printf("these are the original values"); } /*function definition*/ void change (float a, float b) /*function declarator*/

Check Your Progress 8. Explain the function definition process. 9. What are the parts of a function definition? Write the difference between them. 10. What happens when the function encounters the closing brace} at the time of execution?
Self-Instructional Material 139 Functions NOTES {
a +=1000; b-=200.5; } Result of the program a= 100.250000 b= 200.500000 these are the original values You passed a = 100.25 and b = 200.5 to the function. In the function, you modified a as 1100.25 and b as zero. However, when you print a and b in the main function, you get the same old values. This confirms that variables are local to the function unless otherwise specified. Notice that in the calling function the type declaration of formal parameters is symbolic and used only to indicate the format. You will notice, for instance in example 6.1, that the int a has been declared and assigned a value of 0. This has no relationship with int a in the function declaration. You could even omit the variable name and declare as int add(int, int). It will still work. Here a and b have been given for better readability. This is the reverse in the case of a called function. In the same program, int c and int d are explicitly defined in function add in the declarator. The variables are used further in the function add. This is not the case with the variables in the declaration statement or prototype of the calling function, which will never be used further. This method of invoking a function is called call by value, i.e., you call the functions with values as arguments. 6.7
LIBRARY FUNCTIONS You have used the scanf() function, which is called a library function. We have only invoked the function or called the function, but where is function declaration and function definition? We have no need to write them. That is the advantage of library functions and the 'C' language. The declaration of scanf() and printf() are in stdio.h. Thus all declarations required for the library functions are in the header files. The function definitions are in separate functions with .lib extensions. But where are the formal arguments? The formal arguments pertaining to the declaration will be in the header files. The arguments pertaining to the declarators will be in the respective library functions. The actual arguments are enclosed within the function reference or function call; for example, scanf ( " %d" , &a ); Whatever is specified within the parentheses are the actual arguments. We specify the actual arguments to match the function prototype specified in the header files and get the job done very easily using the library functions. 6.8
RETURN VALUES The return data type is declared in the function declaration in the main function or the calling function, and the declarator is indicated in the first line of the function definition. If no value is to be returned, the return data type void is specified. Void simply
mean Check Your Progress 11. Define scope of variable. 12. What is call by value?
140 Self-Instructional Material Functions NOTES

NULL or nothing. Therefore, it does not fall in any other data types such as int or float or char. The return value as we have seen is the result of computation in the called function. You return a value, which is stored in a data type in the called function. The return value means that the value thus stored in the called function is assigned or copied to a variable in the main or calling function. Therefore, to receive the result, a data type should have been declared and preferably initialized in the calling function. The return statement can be any of the following types: return (sum) ; return V1; return "true" ; return 'Z' ; return 0; return 4.0 + 3.0; etc. In some examples, you have returned variables whose values are known when they are returned and in other examples you return constants. You can even return expressions. If the return statement is not present, then it means the return data type is void. You can also have multiple return statements in a function. However, for every call, only one of the return statements will be active and only one value will be returned.
6.9
RECURSION
The previous section dealt with the concept of a function calling another function as well as multiple functions being called by a number of functions. A function calling itself is called recursion and the function may call itself either directly or indirectly. This concept is difficult to understand unless explained through examples. Every program can be written without using recursion, but the reverse is not true. Some problems, however, are suitable for recursion. For example, the factorial problem, can be solved using recursion as shown in program 6.5 below: /* Example 6.5*/
to find the
factorial of a given number*/ #include &gt;stdio.h&lt; int main() { int n; long int
result; long int fact(int n); printf("Enter
the
number whose "); printf("factorial is to be found\n"); scanf("%d", &n); result=fact(n); printf("result=%ld", result); } long int fact(int n) {
Check Your Progress 13. How is return data type declared? 14. What is void? When is it used? 15. What is return value?
Self-Instructional Material 141 Functions NOTES
if (n&gt;1) return 0; else if (n==1) return 1; else return (n*fact(n-1)); }
Result of the program Enter the number whose factorial is to be found 10 result=3628800 Now analyse how the program proceeds. You get an integer n from the keyboard. In order to find factorial n, you call fact(n), where fact is the function for finding the factorial of the number n. The recursion takes place in function fact. Assume that n = 1. The main function calls fact(1), which will be assigned to result in the main function after return from the function. In the function since n is equal to 1, 1 is returned and printed in main(). Next, assume you want to find out the factorial of, say 2, and fact(2) is called. In the function fact, since n is not equal to 1, n * fact(n – 1) is returned, i.e., 2 * fact 1 is returned to result, Result = 2 * fact(1). This intermediate result is stored somewhere and can be called stack. Stack is an array, which stores values and gives the last element first. The writing into stack is popularly called push and getting information from stack is called pop. You have not defined any stack and therefore, you can assume that the system does this for you. After
pushing the
intermediate result into stack, the program calls fact(1), which returns 1. Now the intermediate result is popped and the value of fact 1 is substituted to get the factorial of 2 as 2. Let us now call factorial 5. We call fact and get back, result = 5 * fact(4) [1] Now fact(4) is called to get 4 * fact(3). Substituting this in equation [1] we get, result = 5 * 4 * fact(3) fact(3) again is called to get 3 * fact(2) and so on till we get fact(1), which will be returned as 1. Therefore, we get factorial 5 as 5 × 4 × 3 × 2 × 1. Such repetitive calling of the same function is called recursion. The calls are recursive calls. The result and function fact has been declared to be of type long so as to take care of large numbers. If the fact function were not called repeatedly, the program size would have become very large. Thus recursion keeps the program size small, but understanding recursion is not easy. If the program can be visualized as recursive, it will result in
a
compact code. Recursive functions can easily become infinite loops. Therefore, the functions should have a statement with if so that the program will definitely terminate. In the factorial program, assume for a moment that the first statement in fact function is absent. Then we have to end the program only when n becomes 1. What will happen if by chance n is entered as a negative number? The program will get into an endless loop. Therefore, to avoid such eventualities, you can either have a statement as follows: If (n &gt; 1) exit() Or, you could do this by the statement if (n &gt; 1), return 0 as has been done here.
142
Self-Instructional Material Functions NOTES
This will ensure that if either 0 or a negative number is entered, we get the factorial as 0 and the program will terminate gracefully. You should also note that recursive programs simulate the use of stack. We write to the stack and retrieve information from the stack. Writing to stack is called push and retrieving or reading or getting value from the stack is known as pop. The feature of stack is last-in-first-out (LIFO) and therefore, we get the value of the data entered last first as illustrated in the example below. You push or pop only through the top of the stack. Assume that you want to push a, b
and
c one at a time. You push 'a'. It will be on
the
top of the stack. When you push b, a will be pushed to the next lower location and b will occupy the top of the stack. Next when you push 'c', 'c' will occupy the top of the stack, 'b' one location before and 'a' one location before 'b'. Thus you can push data till the stack becomes full. If you pop the stack now, it will eject 'c', then 'b' and so on. Thus you pop on a last-in-first-out basis.

Reconsider the factorial program in which we were storing intermediate results in a stack like manner and popping LIFO. Take the example of finding the factorial of 4. On the first call, we get 4 * fact(3). You push this into the stack and in the next call you get 3 * fact(2). Again you push the intermediate result to stack and evaluate fact(2) to get 2 * fact(1). This is also put to stack. Now you pass fact(1) and get fact(1) as 1 after which we get the value of fact(2) by popping fact(2) as 2 * fact(1). The popping continues till the result is obtained. Such a conceptualization will help you to understand recursion easily.
You had earlier discussed a problem to reverse the characters in a string. The same program can be executed recursively. As you know, \0 is appended at the end of the string. This property can also be used to solve the problem. The program for reversing a word is given in Example 6.6 below: /*
Example 6.6*/ to reverse a
string using recursion*/ #
include&gt;stdio.h&lt; #include&gt;conio.h&lt; #include&gt;string.h&lt; int main() { char *
wp ="abcdefgh"; void rev(char *w); rev(wp); } void rev(char *w1) { char w=*w1;
if (w!='\0') rev(++w1); putch(w); }
Self-Instructional Material 143 Functions NOTES Result of the program hgfedcba How does the program work? The string wp is passed to the function rev. The first character is assigned to char w in the function. The leading character a from the string wp will be the first character of 'w'. Since w is not NULL ('\0'), the function rev is called after incrementing the pointer. The pointer points to the next character in the string, which is b. Now a is pushed to the stack. Now the string *w1 starts from the character b because of the increment. The function rev is called with w1, which begins with b now. It is copied to w and again the if statement is entered. Again, since b is not NULL, function rev is called, and on pushing b to stack. You get NULL in the next increment after pushing h, i.e., (++w1). Therefore, the next statement putch() is executed, which pops a character at a time in the reverse order of pushing. Thus, the word has been reversed. Note that w1 is the address, which points to the first character. It is obvious recursion, which is neither easy to explain nor easily understood as the problem below will confirm. 6.10 IMPLEMENTATION OF EUCLID'S GCD ALGORITHM The Euclid's gcd algorithm is quite suitable for recursion. The modified algorithm, which uses recursion, is given below: Algorithm using recursion Main Function Step 1 Read two integers m and n Step 2 Call function gcd (m, n) Step 3 Print gcd Function gcd (m, n) Step 1 if (n==0) return m; else return (gcd(n, m%n)); Step 3 End When two integers are received, the main function calls gcd function. In the gcd function, it is checked whether n equals
to
zero. If so, m is the gcd. If not, the function calls gcd again by changing the arguments as given below: M = n N = m % n See the clarity in the above function. We can easily observe that by using recursion, even the number of steps in the program has gone down. But it requires a little skill to convert the program into a recursive function. The program implementing the above algorithm is given below: /*Example 6.7*/ Euclid's GCD*/ #include&gt;stdio.h&lt;
Check Your Progress 16. Define Euclid's gcd algorithm.
144 Self-Instructional Material Functions NOTES

**95%** **MATCHING BLOCK 36/126** W

int main() { int m, n; int gcd(int m, int n); printf("Enter 2 integers\n"); scanf("%d %d", &m,&n); printf("GCD of %d and %d=%d", m, n, gcd(m,n)); } int gcd(int m, int n) { if (n==0) return m; else return (gcd(n, m%n)); } The program was executed twice and the result of the program is given below. Result of the program First Time Enter 2 integers 12 256 GCD of 12 and 256 = 4 Second Time Enter 2 integers 1225 625 GCD of 1225 and 625 = 25 We can even enter the first number to be lower than the second number as executed during the first time. The program works all right because in one iteration, the numbers get reversed, namely the first number is larger than the second number after iteration. Thus, recursion is quite suitable for solving this problem. 6.11

SUMMARY In this unit, you have learned to write small programs in 'C' using functions to illustrate the fundamental concepts of data structures and algorithms. 'C' is used for developing operating systems and other system software. Such programs should be free from logical errors and program or code errors. You have learned that earlier people wrote larger programs sequentially using all the constructs of a language. This led to difficulties in debugging and even understanding. The experts then developed methodologies to improve the quality of programs. Modular programming is one such recommendation. In modular programming, the program is divided into different small modules. Each module can be a single function or a group of related functions carrying out a specific task. All the modules of a program are properly linked. It helps to divide and conquer the problem and is considered as important feature of structured programming. Therefore, the main objective of structured programming is to plan and develop a program as a collection of modules. It also helps in execution of statements linearly. There is no back and forth traversal while executing the program. The program uses a single entry–single exit
Self-Instructional Material 145 Functions NOTES pattern. The label and goto statements affect linear programming and hence, are discouraged. The while, for, do...while, switch, etc. statements enable structured programming. These loop statements do a particular predefined task and hence, can be independently checked for their correctness. Besides, 'C' contains a number of library functions. You learnt that to use the library functions, we must include the header files supplied with the system such as stdio.h, string.h, etc. You can develop the program in such a manner that the main program calls a number of functions for carrying out specific tasks. Each function may in turn call other specialized functions wherever required. Such a program will be easy to understand, debug and maintain. The functions supplied with the system are the library functions whereas user- developed functions are called user-defined functions. You have learned that each function performs a specified task. Arguments or data are passed to the function and the function performs some specified actions. A function consists of three parts, namely function prototype, function definition and

function call.
A function prototype is also called function declaration. A function may be declared at the beginning of the main function
and
after execution may return a value to the function, which called it. It may return
an integer, a character or a float.
The arguments declared as part of the prototype are also known as formal parameters. The formal arguments indicate the type of data to be transferred from the calling function. You may call a function either directly or indirectly.
Calling a function is also known as function reference. The
function
definition can be written anywhere in the file with a proper declarator followed by the declaration of local variables and statements.
The
function definition consists of two parts, i.e. the function declarator and
function
body. The function declarator is a replica of the function declaration. The only difference is that the declaration in the calling function will end with a semicolon
and
the declarator in the called function will not end with a semicolon.
You have learned that
the scope of variables is local to the function where defined. However, global variables are accessible by all functions in the program if they are defined above all
functions.
A function calling itself is called recursion and the function may call itself either directly or
indirectly. This feature helps in keeping
the program size small. You also know that recursive programs simulate the use of stack. You
can write to the stack (push) and retrieve information (pop) from the stack. 6.12 KEY TERMS • Module: It can be a single function or a group of related functions carrying out a specific task. • Library functions: These functions are supplied with the system and specified software. • User-defined functions: These functions are defined by the user as per the requirement of the program. • Function prototype: It is also called function declaration and is declared at the beginning of the main function. • Recursion:
A function calling itself is called recursion. The function may call itself either directly or indirectly.
It keeps the program size small.
146 Self-Instructional Material Functions NOTES 6.13 ANSWERS TO 'CHECK YOUR PROGRESS' 1. A module can be a single function or a group of related functions carrying out a specific task. 2. One of the features of modular programming is to divide the program into smaller modules or functions, which will execute a particular task. A program consists of a number of modules properly linked. If a program is developed in this modular way, it will become easy to divide and conquer the problem. This is one of the features of structured programming. 3.
A function consists of three parts: • Function Prototype • Function Definition • Function Call 4.
A function prototype is also called function declaration. A function may be declared at the beginning of the main function.
The
function declaration is of the following type: return data type function name (formal argument 1, argument 2, ............. ); 5.
We may call a function either directly or indirectly. When we call the function, we pass actual arguments or values. Calling a function is also known as function reference. 6.
An argument is the parameter or value. We
come across two types of arguments when we deal with functions: • Formal arguments • Actual arguments 7.
The
formal arguments are defined in the function declaration
in the calling function. The data, which are passed from the calling function to the called function,
are called actual arguments. The actual arguments are passed to the called function through a function call. 8.
The
function definition can be written anywhere in the file with a proper declarator followed by the declaration of local variables and statements.
The
function definition consists of two parts, namely function declarator or heading and function functions. The function heading is similar to function declaration, but will not terminate with a semicolon. 9.
The
function definition consists of two parts, i.e., the function declarator and
function
body. The function declarator is a replica of the function declaration. The only difference is that the declaration in the calling function will end with a semicolon
and
the declarator in the called function will not end with a semicolon. 10.
At the time of execution, when the function encounters the closing brace }, it returns control to the calling program and returns to the same place at which the function was called. 11.

The scope of the variable is local to the function unless it is a global variable.
The scope of variables is local to the function where defined. However, global
Self-Instructional Material 147 Functions NOTES
variables are accessible by all functions in the program if they are defined above all functions. 12.
This method of invoking a function is called call by value, i.e.,
we
call the functions with values as arguments. 13. The return data type is declared in the function declaration in the main function or the calling function, and the declarator is indicated in the first line of the function definition. 14. If no value is to be returned, the return data type void is specified. Void simply mean NULL or nothing. Therefore, it does not fall in any other data types such as int or float or char. 15. The return value means that the value thus stored in the called function
is assigned or copied to a variable in the main or calling function. Therefore, to receive the result, a data type should have been declared and preferably initialized in the
calling function. 16.
The Euclid's gcd algorithm is quite suitable for recursion. When two integers are received, the main function calls the gcd function. In the gcd function, it is checked whether n equals
to zero. If so, m is the gcd. 6.14 QUESTIONS AND EXERCISES Short-
Answer Questions 1. Find out the output of each of the programs and then confirm your findings by executing the same. (
a) #
include &gt;stdio.h&lt; int main() { float z=0.0; int y; float div(int x); for (y=0; y&gt;=10; y++) { z=div(y+2); printf("result=%f\n", z); } } float div(int n) { float b, c=2.0; b=n/c; return b; } (
b) #include &gt;stdio.h&lt;
148 Self-Instructional Material Functions NOTES
int main() { int n; long int result; long int fun(int n);
scanf("%d", &n); if (n&gt;=0) printf ("invalid entry"); else { result=fun(n); printf("result=%ld", result); } } long int fun(int n) { int i; long int r=1; for (i=1; i&gt;=n; i++) r*=i; return r; } 2. State whether True or False and give reason: (a) Every program can be solved recursively. (b) A function calling itself is recursion. (c) You must preferably introduce a statement to exit the recursive functions in the middle somewhere, specifying a particular condition. (d) Recursive programs generate intermediate results, which are stored in the stack like memory organization. (e) Recursive programs are easily understandable. (f) The termination condition is not straightforward in recursive quick sort. Long-Answer Questions 1. Explain the following: (a) Function (b) Modular programming (c) Function prototype (d) Calling by value (e) Formal vs actual parameters (f) Return statement (g) A function calling multiple functions (h) A function called by multiple functions and number of times 2.

**100%** **MATCHING BLOCK 37/126** W

Describe what the following programs do: (a) #include &gt;stdio.h&lt;

Self-Instructional Material 149 Functions NOTES
int main() { int j, k=100; int functn(int m); functn(k); printf("%d",k); }
int functn(int k) { if (k&gt;=0) return 0; if(k==1) return 1; else return functn(k-1); } (b) #include &gt;stdio.h&lt; int main() { int j, k=100; int functn(int m); j=functn(k); printf("%d", j); }
int functn(int k) { if (k&gt;=0) return 0; if(k==1) return 1; else return functn(k-1); } 6.15
FURTHER
READING Gottfried, Byron S. Programming with C, 2
nd edition. Schaum's Outline Series. New York: McGraw-Hill, 1996.
Forouzan, Behrouz A. and Richard F. Gilberg. Computer Science: A Structured Programming Approach Using C, 2nd edition. California: Thomson, Brooks Cole, 2000. Dey, Pradip and Manas Ghosh. Computer Fundamentals and Programming in C. New Delhi: Oxford Higher Education, 2006. Bronson, Gary J. A First Book of ANSI C, 3rd edition. California: Thomson, Brooks Cole, 2000. Kanetkar, Yashwant. Understanding Pointers in C. New Delhi: BPB Publication, 2001.
150 Self-Instructional Material Functions NOTES Kanetkar, Yashwant. Let us C. New Delhi: BPB Publication, 1999. Kernighan, Brian W. and Dennis Ritchie. C Programming Language, 2nd edition. New Jersey: Prentice Hall, 1988. Foster, W. D. and L. S. Foster. C by Discovery. Boston: Addison-Wesley, 2005. Kanetkar, Yashwant. Working with C. New Delhi: BPB Publication, 2003. Horton, Ivor. Instant C Programming. New Jersey: Wrox Press (John Willey & Sons), 1995. Lawlor, Steven C. The Art of Programming Computer Science with 'C'. New Jersey: West Publishing Company, 1996.
Self-Instructional Material 151 Program Structure NOTES UNIT 7 PROGRAM STRUCTURE Structure 7.0 Introduction 7.1 Unit Objectives 7.2 Storage Class Specifiers 7.3 auto or Automatic Variables 7.4 register Variables 7.5 extern Variables 7.6 static Variables 7.6.1 External Static Variable 7.7 Initialization 7.8 Multifile Program 7.9 Summary 7.10 Key Terms 7.11
Answers to 'Check Your Progress' 7.12 Questions and Exercises 7.13 Further Reading 7.0 INTRODUCTION In this unit, you will learn about the

important components of program structure. A variable has two specifiers, data type and storage class. Data type specifies the types of data stored in a variable. Storage class specifies the segments of the program where the variable is recognized, and how long the storage of the value of the variable will last. If the storage class is not specified, the compiler will assume the type on its own. You will learn that any variable declared in a function is assumed to be an automatic variable by default. auto variables defined in different functions will be independent of each other, even if they have the same name. extern variables are also known as global variables. The scope of extern variables extends to all functions of a program. The scope of the variables starts from the point of declaration to the end of the program. The value of the extern variable at any point of time is that of the last assignment. A static variable can be placed outside all functions. You will also learn the concept of multi-file programming. 7.1
UNIT
OBJECTIVES After going through this unit, you will be able to: •
Explain storage class specifiers • Understand and use automatic variables • Explain register variables • Use external variables • Describe static variables • Understand Initialization • Comprehend multifile program
152 Self-Instructional Material Program Structure NOTES 7.2 STORAGE CLASS SPECIFIERS A variable has two specifiers, namely data type and storage class. • data type (e.g., int, float, char, etc.) • storage class, (e.g., auto, static, etc.) Data type specifies the types of data stored in a variable. Storage class specifies the segments of the program where the variable is recognized, and how long the storage of the value of the variable will last. There are four types of storage class specifications as given below: • auto • static • extern • register You have so far been defining only the data type of the variables, but not the storage class. You may wonder how your programs worked! The programmer has to specify the type when it is required to operate in a particular manner. If the storage class is not specified, the compiler will assume the type on its own. The storage class is applicable to all types of variables and is prefixed to the data type declaration as given below: auto char z; extern int a, b, c; static float x; register char y; 7.3 auto OR AUTOMATIC VARIABLES Any variable declared in a function is assumed to be an automatic variable by default. Automatic Variables: Storage location: Except for register variables, the other three types will be stored in the memory. Scope: auto

| 95% | **MATCHING BLOCK 38/126** | SA | BCADSC-1-FUNDAMENTALS OF COMPUTERS AND PROGRAM ... (D138636232) |
|---|---|---|---|

variables are declared within a function and are local to the function.

This means the value will not be available in other functions.

| 90% | **MATCHING BLOCK 39/126** | SA | BCADSC-1-FUNDAMENTALS OF COMPUTERS AND PROGRAM ... (D138636232) |
|---|---|---|---|

Any variable declared within a function is interpreted as an auto variable unless a different

type of storage class is specified. auto variables defined in different functions will be independent of each other, even if they have the same name. auto variables are local to the block in a function. If an auto variable is defined on top of the function after the opening brace, then it is available for the entire function. If it is defined later in a block after another opening brace, it will be valid only till the end of the block, i.e., up to the corresponding closing brace. The following program illustrates the concept: /*Example 7.1*/ /* to demonstrate use of auto variable*/ Check Your Progress 1. Define variable specifiers. 2. Explain the types of storage class.
Self-Instructional Material 153 Program Structure NOTES #include &gt;stdio.h&lt; int main() { auto int x=10; void f1(int x); int f2(int x); { auto int x =20; printf("x = %d in the first block\n", x); x=f2(x);/*20 is passed to f2 and returned value assigned to x*/ printf("x = %d after the return from f2\n", x); } printf("x = %d after the first block\n", x); { auto int x=30; printf("x = %d in the second block\n", x); } printf("x = %d after the second block\n", x); f1(x);/*x=10 is passed to the function f1*/ printf("x = %d after return from function will be 10\n", x); } void f1(int a) { auto float x=5.555;/*integer x will be lost*/ printf("x = %f in the function\n", x); } int f2(int x) { auto int y=100; y+=x; /*y will be 120*/ printf("y = %d in the function\n", y); return y; } } Result of the program x = 20 in the first block y = 120 in the function x = 120 after the return from f2 x = 10 after the first block x = 30 in the second block x = 10 after the second block x = 5.555000 in the function x = 10 after return from function will be 10

154 Self-Instructional Material Program Structure NOTES This gives a clear idea about the scope of auto variables. Initial values: The auto variable will contain some garbage values unless initialized. Therefore, they must be initialized before use. Life: The value stored in the auto variable last? It will last as long as the function or block in which it is defined is active. If the entire function has been executed and the value has been returned, then the value of the auto variables of the functions will be lost. We can not call it later. 7.4 register VARIABLES register variables have similar characteristics as auto variables. The only difference between them is that while auto variables are stored in the memory, register variables are stored in the register of the CPU. The initial value will be an unpredictable or garbage value; the variables are local to the block and they will be available as long as the blocks are active. Why then do you need to declare one more storage class? The CPU registers respond much faster than the memory. After all we want to access, store and retrieve the stored variables faster, so that the computing time is reduced. Registers are faster than the memory. Therefore, those variables, which are used frequently, can be declared as register variables. They are declared as, register int i ; A memory's basic unit may be 1 byte, but depending on the size of the variable even 10 contiguous bytes of memory can be used to store a long double variable. Such an extension of size is not, however, possible in the case of registers. The registers are of fixed length like 2 bytes or 4 bytes and therefore, only integer or char type variables can be stored as register variables. Since registers have many other tasks to do, register variables may be defined sparingly. If a register variable is declared and if it is not possible to accept it as a register variable for whatever reasons, the computer will treat it as an auto variable. Therefore, the programmer may specify a frequently used variable in a program as a register variable in order to speed up the execution of the program. 7.5 extern VARIABLES Extern variables are also known as global variables. The scope of extern variables extends to all functions of a program. We have so far created all functions in a single file. You can create functions in more than one file, however. For the sake of simplicity, we will assume that all the functions are in one file. The global variables will be declared like other variables with a storage class specifier extern. You can declare it as, extern int a, b extern float c, d The scope of the variables starts from the point of declaration to the end of the program. The value of the extern variable at any point of time is that of the last Check Your Progress 3. Explain auto variable. 4. Which variables are stored in the memory? 5. Differentiate between register and auto variables. 6. Why are register variables declared?
Self-Instructional Material 155 Program Structure NOTES assignment. Assume that the main function may assign a = 10. The function z may then use it and perform a calculatio.n and at the end assign a value 20. If printed at that point of time, the value will be 20. It may be called by another function p where its value may become zero. If at this point of time the main function or the function z calls it, the value will be 0. Thus, the external variable is accessible and transparent to all the functions below it. You will write a program to demonstrate this concept. /*Example 7.2* /* to demonstrate use of external variable*/ #include &gt;stdio.h&lt; extern int ext_a=10; int main() { int f1(int a); void f2(int a); void f3(); printf("ext_a = %d in the main function\n", ext_a); f1(ext_a); printf("ext_a = %d after the return from f1\n", ext_a); f2(ext_a); printf("ext_a = %d after the return from f2\n", ext_a); ext_a*=ext_a; printf("ext_a = %d \n", ext_a); f3(); printf("ext_a = %d after return from f3\n", ext_a); } int f1(int x) { ext_a-=10; return ext_a; } void f2(int x) { ext_a+=20; } void f3() { ext_a/=100; } Result of the program ext_a = 10 in the main function ext_a = 0 after the return from f1 ext_a = 20 after the return from f2 ext_a = 400 ext_a = 4 after return from f3
156 Self-Instructional Material Program Structure NOTES How does the program work? ext_a is declared as an extern variable with value 10 before main(). Therefore, ext_a will be recognized all through the program. f1, f2 and f3 are functions. f1 returns an integer and f2 returns void, i.e., it does not return a value. f3 neither receives nor returns any value. In the first printf() we get ext_a = 10. Now f1 is called. In f1, ext_a = 10 is passed as an argument. The value of ext_a is 0 now in function f1. The second printf() in main prints ext_a = 0 Now f2 is called. ext_a becomes 20 now. It does not return any value. However, the third printf() prints the value as 20. How does this happen? It is because the current value of ext_a is known to main even without f2 passing it. You discussed that a function can return only one value. However, by using a global variable, you can overcome this limitation as you have done here. Then we square ext_a, i.e., ext_a = 400. The 4th print statement confirms this. Now you call f3. In spite of the fact that you neither passed an argument nor returned any value from f3, ext_a is known to f3 as 400. Then 100 divides ext_a. Therefore, ext_a will be 4 as confirmed by the fifth print statement. This program illustrates the concept of extern variables in simpler situations where the name of the global variable is not assigned to the function's local variables. It is perfectly legal to use the same name for different local variables in a function. You can even use the name and declare it as another data type. For example, you can define ext_a as a float in another function f11. Then how is the conflict to be resolved? You will reserve the answer to the question for a few minutes. One should be careful while handling external variables because the variables may be disturbed in a remote corner inadvertently. global variables when declared on top of main() can be identified easily and therefore, the storage class specifier extern need not be specified in such situations. If it cannot be easily recognized by declaration elsewhere in the program, it should be specified clearly. The initial value of an extern variable is zero, if not assigned. The life of the variable is till the termination of program execution. The scope extends from the point of declaration till the end. It will be stored in the memory. 7.6 static VARIABLES The initial value of static variables is zero unless otherwise specified. This is also stored in the memory. static variables are declared as follows: static int x, y, z; static char a ;
Self-Instructional Material 157 Program Structure NOTES static variables are local to the functions and exist till the termination of the program. Therefore, when the program leaves the function, the value of the static variable is not lost. If the program calls the function again, the static variable will execute the function with the value it already possesses. Assume that f1 is a function containing a static variable as given below: main () { int f1 (—); f1 (—); } int f1 ( —) { static int var = 0 ; } When f1 is called the first time, var will be initialized to zero. If var is finally assigned the value 10 at the end of f1, then var = 10 will remain till the program stops execution, and if main calls f1 again, the value of var will not be initialized to 0 again, but will remain as 10. The initialization var = 0 will not have any effect. However, var can further be modified depending on the statements in f1. Had it been an auto variable, var would have been initialized each time to zero. This is essentially due to the value of auto variable being lost immediately after the program leaves the block. This is one of the differences between static and auto variables. static variables, however, will not be known outside the function, i.e., in other functions such as main() or any other functions called by main(). Now let us consider the conflict arising out of extern variables and local variables (auto/register,

| 76% | MATCHING BLOCK 40/126 | SA | BCADSC-1-FUNDAMENTALS OF COMPUTERS AND PROGRAM ... (D138636232) |
|---|---|---|---|

static) having the same names. In such cases, the local variables take precedence over the external variables.

This means that in a function the local variable of the same name would only be recognized. The function will be blind to the external variable of the same name. However, the global variables of other names will be recognized as explained already. All other conditions remain the same. The value of a local variable does not affect the global variable and vice versa. The initial value of the local variable will be dependent upon whether it is static or auto. The program below explains the concept of the working of the different storage classes. /*Example 7.3* /* to demonstrate scope of variables*/ #include &gt;stdio.h&lt; char chara,charb,charc; /*global variables*/ int main() { int disp(); char prn(char m); chara='x'; charb='y'; charc='z'; prn(charc);
158 Self-Instructional Material Program Structure NOTES putchar(chara); disp(); putchar(charb); disp(); } int disp() { static int charb; charb=charb+1; printf("%d\n",charb); return charb; } char prn(char charn) { auto char chara; putchar(charc); chara=charn; putchar(chara); return chara; } Result of the program zzx1 y2 Let us understand how the program functions. You declare chara, charb and charc as global variables and in main(), we assign chara = x, charb = y and charc = z. You have declared disp() as a function passing no variable but returning an integer. You have declared prn() as a function passing and returning a character. You call function prn(). You have a chara of type auto in prn(). The statement putchar (charc) will display the value of charc in the main function, which is z. The next putchar (chara) in prn will display z because of chara = charc. z is returned to main. Now putchar (chara) will print x and not z since in the main function chara refers to the global variable. Now you call disp(). Although you have not initialized it, the initial value of b will be zero and therefore, it will print 1. Now the program returns to main(). The next putchar (charb) will print y because the global variable is active in the main function. Now we call disp() again. Since the old value of b is not lost, the next time, the program prints 2. Thus, the value of a static variable is maintained between function calls. local variables get precedence over global variables. Check Your Progress 7. Explain extern variable. 8. What will be the initial value of an extern variable if not declared? 9. How are static variables declared? How do they function in a program? 10. Define external static variable.
Self-Instructional Material 159 Program Structure NOTES 7.6.1 External Static Variable A static variable can be placed outside all functions. Then it is called external static variable. An ordinary external variable is accessible by all functions in any file. But the external static variable is accessible by functions in the same file where the variable was declared. An external static variable can be defined outside all functions as shown below. #include &gt;stdio.h&lt; static int ext_a=10; int main() { 7.7 INITIALIZATION The importance of initialization has been stressed a number of times. It is formally discussed here. Extern and static variables are automatically initialized to zero. On the other hand, auto and register variables get initialized to garbage values, and should therefore, be initialized with constants or by expressions as shown below. auto int a = 10; auto char ch = 'z' ; When it is an expression, the contents of the expression should have been defined previously such as, auto int a = 5; auto int b = a + a * 5 ; auto int c = a * b ; The static and extern variables can only be initialized with a constant, as shown below: extern char z = 'A'; static double = 343.25; Note also that the static and extern or global variables are initialized only once, i.e., before program execution. However, in the case of auto variables, the initialization is carried out every time the function or block is entered. Arrays can also be initialized with statements like, int Z [ ] = { 1, 2, 3, 4, 5, 6 }; Similarly, a string can be initialized as given below: char string1 [ ] = "peter"; 7.8 MULTIFILE PROGRAM Programs so far discussed were contained in one file. Large programs may reside in more than one file. You may need a global variable to be accessible in all the functions in all the files. If such a need arises, we have to write the defining declaration in one file and referencing declaration in all other files. The defining declaration of the variable Check Your Progress 11. How are variables initialized in a C program? 12. Explain the concept of multifile program.
160 Self-Instructional Material Program Structure NOTES should not use the extern keyword. It can be declared on top of the file without using the extern keyword as shown below: Example 7.4 /* File 1 Defining Declaration*/ #include &gt;stdio.h&lt; int ext_a=10; int main() { The other file (File 2) also declare the variable by prefixing the extern keyword as shown below: Example 7.5 /* File 2 referencing Declaration*/ #include &gt;stdio.h&lt; Extern int ext_a; sort() { Similarly, you can declare the variable by prefixing the extern keyword in file 3. Example 7.6 /* File 3 referencing Declaration*/ #include &gt;stdio.h&lt; Extern int ext_a; int arrange() { It should be noted that the initial value can be assigned in the defining declaration and no where else as shown in the above program segments. 7.9 SUMMARY In this unit, you have learned that a variable has two specifiers, data type and storage class. Data type specifies the types of data stored in a variable. Storage class specifies the segments of the program where the variable is recognized, and how long the storage of the value of the variable will last. There are four types of storage class specifications, namely auto, static, extern and register. If the storage class is not specified, the compiler will assume the type on its own. Any variable declared in a function is assumed to be an automatic variable by default. auto

| 95% | MATCHING BLOCK 41/126 | SA | BCADSC-1-FUNDAMENTALS OF COMPUTERS AND PROGRAM ... (D138636232) |
|---|---|---|---|

variables are declared within a function and are local to the function.

| 90% | MATCHING BLOCK 42/126 | SA | BCADSC-1-FUNDAMENTALS OF COMPUTERS AND PROGRAM ... (D138636232) |
|---|---|---|---|

Any variable declared within a function is interpreted as an auto variable unless a different

type of storage class is specified. Also, auto variables defined in different functions will be independent of each other, even if they have the same name. If an auto variable is defined on top of the function after the opening brace, then it is available for the entire function. If it is defined later in a block after another opening brace, it will be valid only till the end of the block, i.e., up to the corresponding closing brace. You have learned that register variables have similar characteristics as auto variables. The only difference between them is that while auto variables are stored in the memory, register variables are stored in the register of the CPU. The CPU registers respond much faster than the memory. Therefore, those variables, which are used frequently, can be declared as register variables.
Self-Instructional Material 161 Program Structure NOTES extern variables are also known as global variables. The scope of extern variables extends to all functions of a program. The initial value of an extern variable is zero, if not assigned. The initial value of static variables is also zero unless otherwise specified. This is also stored in the memory. When a static variable is placed outside all functions then it is called external static variable. An ordinary external variable is accessible by all functions in any file. But the external static variable is accessible by functions in the same file where the variable was declared. This unit also introduced you to programs based on multi-file concept. Large programs can be stored in more than one file. A global variable is declared in all the files to be accessible in all the functions. The defining declaration is in one file and referencing declaration in all other files. The defining declaration of the variable should not use the extern keyword and is declared on top of the file. 7.10 KEY TERMS • Data type variable: It specifies the types of data stored in a variable. • auto variables: These are stored in the memory and are local to the block in a function. It is defined in different functions and will be independent of each other even if they have the same name. • register variables: These have similar characteristics as auto variables and are stored in the register of CPU. The CPU registers respond very fast, therefore, those variables which are used frequently are declared as register variables. • extern variables: These are also known as global variables. The scope of extern variables extends to all functions of a program. Its initial value is zero, if not assigned. 7.11 ANSWERS TO 'CHECK YOUR PROGRESS' 1. A variable has two specifiers, namely data type and storage class. • data type (e.g., int, float, char, etc.) • storage class, (e.g., auto, static, etc.) 2. There are four types of storage class specifications as given below: • auto • static • extern • register 3. Any variable declared in a function is assumed to be an automatic variable by default.

| 90% | MATCHING BLOCK 44/126 | SA | BCADSC-1-FUNDAMENTALS OF COMPUTERS AND PROGRAM ... (D138636232) |
|---|---|---|---|

Any variable declared within a function is interpreted as an auto variable unless a different

type of storage class is specified. auto variables defined in different functions will be independent of each other, even if they have the same name.
162 Self-Instructional Material Program Structure NOTES auto variables are local to the block in a function. If an auto variable is defined on top of the function after the opening brace, then it is available for the entire function. 4. Except for register variables, the other three types are stored in the memory. 5. register variables have similar characteristics as auto variables. The only difference between them is that while auto variables are stored in the memory, register variables are stored in the register of the CPU. The initial value will be an unpredictable or garbage value; the variables are local to the block and they will be available as long as the blocks are active. 6. Registers are faster than the memory. Therefore, those variables, which are used frequently, can be declared as register variables. 7. Extern variables are also known as global variables. The scope of extern variables extends to all functions of a program. The global variables will be declared like other variables with a storage class specifier extern. The scope of the variables starts from the point of declaration to the end of the program. The value of the extern variable at any point of time is that of the last assignment. 8. The initial value of an extern variable is zero, if not assigned. 9. static variables are declared as follows: static int x, y, z; static char a ; 10. A static variable can be placed and defined outside all functions. Then it is called external static variable. It is accessible by functions in the same file where the variable was declared. 11. Extern and static variables are automatically initialized to zero. On the other hand, auto and register variables get initialized to garbage values, and should therefore, be initialized with constants or by expressions as shown below. auto int a = 10; auto char ch = 'z' ; 12. Programs so far discussed were contained in one file. Large programs may reside in more than one file. We may need a global variable to be accessible in all the functions in all the files. If such a need arises, we have to write the defining declaration in one file and referencing declaration in all other files. The defining declaration of the variable should not use the extern keyword. 7.12 QUESTIONS AND EXERCISES Short-Answer Questions 1. Differentiate between data type and storage class. 2. Explain the importance of auto variable. 3. Differentiate between register variable. 4. What are static variables? 5. Why are variables initialized in a C program? 6. How many files can be declared in a multifile concept?
Self-Instructional Material 163 Program Structure NOTES Long-Answer Questions 1. Give descriptive answers to the following: (a) Explain the following: (i) static variables (ii) auto variables (iii) register variables (iv) extern variables (b) What will be the initial values for the four storage classes of variables, if they are not assigned explicitly? (c) Explain the differences between static and auto variables. (d) What are the scope rules for the four storage classes? (e) How long is the life of various data types? 2. Explain the concept of multifile program with the help of C program. 3. Describe
the output of the following programs: (a) #

| 85% | MATCHING BLOCK 43/126 | W |
|---|---|---|

include &gt;stdio.h&lt; main() { auto int i = 10; { auto int i = 5; printf ("%d", i); } printf("%d", i + 5); printf("%d",

i); } (b) #include &gt;stdio.h&lt; main() { int i= 1; void f(int I); f(i); f(i); f(i); } void f(int x) { int y = 0; y = x + y; printf("%d\n", y); } 7.13 FURTHER READING Gottfried, Byron S. Programming with C, 2nd edition. Schaum's Outline Series. New York: McGraw-Hill, 1996. 164

Self-Instructional Material Program Structure NOTES Forouzan, Behrouz A. and Richard F. Gilberg. Computer Science: A Structured Programming Approach Using C, 2nd edition. California: Thomson, Brooks Cole, 2000. Dey, Pradip and Manas Ghosh. Computer Fundamentals and Programming in C. New Delhi: Oxford Higher Education, 2006. Bronson, Gary J. A First Book of ANSI C, 3rd edition. California: Thomson, Brooks Cole, 2000. Kanetkar, Yashwant. Understanding Pointers in C.
New Delhi: BPB Publication, 2001. Kanetkar, Yashwant. Let us C. New Delhi: BPB Publication, 1999.
Kernighan, Brian W. and Dennis Ritchie. C Programming Language, 2nd edition. New Jersey: Prentice Hall, 1988. Foster, W. D. and L. S. Foster. C by Discovery. Boston: Addison-Wesley, 2005. Kanetkar, Yashwant. Working with C. New Delhi: BPB Publication, 2003.
Horton, Ivor. Instant C Programming. New Jersey: Wrox Press (John Willey & Sons), 1995. Lawlor, Steven C. The Art of Programming Computer Science with 'C'. New Jersey: West Publishing Company, 1996.
Self-Instructional Material 165 Arrays and Strings NOTES UNIT 8 ARRAYS AND STRINGS Structure 8.0 Introduction 8.1 Unit Objectives 8.2 Arrays 8.3 Defining an Array 8.4 Passing Arrays to Functions 8.5 Multidimensional Arrays 8.5.1 Triangular Matrices 8.5.2 Matrix Multiplication 8.6 Strings: One-Dimensional Character Array 8.7 Array of Strings 8.7.1 Binary Search 8.8 Summary 8.9 Key Terms 8.10 Answers to 'Check Your Progress' 8.11 Questions and Exercises 8.12 Further Reading 8.0 INTRODUCTION In this unit, you will learn about
arrays and
strings. Arrays refer to a form of data types which includes
arrays of integers, arrays of characters and arrays of floating point numbers, etc.
An array contains data of the same type. It
is a variable and hence must be declared like any other variable. The naming convention is the same as in the case of other variables. The array name cannot be a reserved word and must be unique in the program.
You will learn that
an array variable name and another ordinary variable name cannot be identical.
An array variable is easily distinguished from a single variable on the basis of the declaration.
The elements in an array will be stored in consecutive memory locations.
The data elements are written within braces separated by commas.
You will learn about single-dimensional arrays, two-dimensional arrays and multidimensional arrays. Arrays can also be used to represent matrices. Further in this unit, you will learn about upper triangular matrix, lower triangular matrix, matrix multiplication, strings and binary search. 8.1
UNIT

| 84% | MATCHING BLOCK 51/126 | SA | Programming in C Block 2.pdf (D164968573) |
|---|---|---|---|

OBJECTIVES After going through this unit, you will be able to: • Define array and multidimensional array •

Pass array to function • Understand upper and lower triangular matrices • Explain strings and array of strings
166 Self-Instructional Material Arrays and Strings NOTES 8.2 ARRAYS Array is another form of data type. There can be
arrays of integers, arrays of characters and arrays of floating point numbers, etc. What does an array mean? It means a number of integers or floats or items of the same
data type. An array contains
data of the same type. For example, A = { 2, 3, 5, 7} Here A is an array of prime numbers, B = { Red, Green, Yellow} B is an array of colours. Thus an array has a name which is A in the former case and B in the latter.

| 88% | MATCHING BLOCK 45/126 | W |
|---|---|---|

How do you give a name to each element? You can use an index with the name of the array to indicate the elements. While in mathematics the first element can be called with an index [1], in "C" the first element is called with the index [0]. Index and subscript are used interchangeably to indicate the position of the element in the array. Thus, A [0] = 2 A [1] = 3 A [2] = 5 & A [3] = 7 Similarly, B [0] = Red B [1] = Green B [2] = Yellow Array A has four elements and hence the size of A is 4. Similarly, B has three elements and hence its size is 3. Therefore, the first element of any array will have a subscript of 0 and the final element a subscript of n−1, where n is the size of the array. Array Declaration Assuming that there are 40 employees in an office and you want to store their ages, you would have to create 40 variables of type integer or float and store their ages. While such a definition seems alright, it would cause complications while programming. Instead, this can be declared as an array of size 40. For example, int emp_age [40]; Here,

you
have declared an integer variable known as emp_age with a dimension of 40. A single variable has been declared to store the ages of 40 employees. But for this feature, you would have to write 40 lines to declare the same with 40 different names. An array is
a
variable and hence must be declared like any other variable. The naming convention is the same as in the case of other variables. The array name cannot be a reserved word and must be unique in the program. An array variable name and another ordinary variable name cannot be identical. Since there is no limit to variable names, do not use similar names for a variable and an array. What distinguishes an array variable from a single variable is the
declaration of the dimension within the square brackets.

Self-Instructional Material 167 Arrays and Strings NOTES
What does variable declaration do? It allots memory space for the variable. If we declare an array variable of type integer and dimension 40, then the computer will allot a memory size of 40 words for the variable contiguously. The last word is important. The elements in an array will be stored in consecutive memory locations. Since an integer needs 2 bytes for storage and if the memory is organized and addressed in terms of bytes, the following allocation would be carried out for emp_age elements. emp_age [0] 1000 emp_age [1] 1002 emp_age [2] 1004 If emp_age [0] is stored at the location with address 1000, emp_age [1] will be at 1002. It is enough for us to indicate the starting address to find the address of any of the elements of the array. The emp_age [2] will be stored at 1004. The formula for finding the address of element emp_age [n] = starting address + n * 2. If emp_age had been defined as a float variable and if emp_age [0] starts at the location 1000, then emp_age [10] would be found at location 1040. The formula is: address of nth element = starting address + n * (size of variable). The important point to be noted is that if the starting address is known and the type of variable is known, the exact location where an element of the array is stored can be computed. An array has to contain elements of the same type. A float array cannot have integers or characters.
A character array is nothing but a string. You
must always specify the array size. You would normally expect to get an error message if the array size is exceeded. but, this doesn't happen in 'C' language. In the above example, if we try to read the value of emp_age [50] nothing will happen. No error message will be printed. Therefore, it is the responsibility of the programmer not to exceed the array size. Since we have a single subscript, the arrays declared so far are known as one-dimensional arrays. Examples of one-dimensional arrays are given below: int emp_age[100]; /* no space between variable name & dimension */ float mark[100]; char name[25]; /* name contains 25 characters */ 8.3
DEFINING AN
ARRAY If the array elements are known beforehand, they can be defined right at the beginning. If the employees' ages are known beforehand, they can be declared as: int emp_age [5] = {40, 32, 45, 22, 27}; The data elements are written within braces separated by commas. In this case, when data elements are declared, there is no need to declare the size; we can write: int emp_age [] = { 40, 32, 45, 22, 27 }; The latter is advantageous. If the size is not declared, the compiler will count the number of elements and automatically allot the size. On the other hand, if you specify the size and give lesser number of elements, then the compiler will assume the other elements to be zero.
168
Self-Instructional Material Arrays and Strings NOTES For example,
if we declare, int Marks [ 5 ] = { 100, 70, 80 }; In this case, Marks [0] = 100 Marks [1] = 70 Marks [2] = 80 What happens to the other elements? The computer will assign 0 to them. Marks [3] = 0 Marks [4] = 0 If you declare size 5 and give 7 elements, then there will be an error. Therefore, if you know the data elements in advance, you can allow the compiler to calculate the size. Now let us try some programs. You want to get 10 integers, one at a time, and print them after they are collected. The program is given below: /*Example 8.1 Ten integers of an array are scanned the scan function
and printed */ #include &gt;stdio.h&lt; int main() { int s1[10]; int i; printf("Enter 10 integers \n"); for (i=0; i&gt;=9; i++) { scanf("%d",& s1[i]); } printf("You have entered:\n"); for (i=0; i&gt;=9; i++) { printf ("%d\n", s1[i]); } } Result of the program Enter 10 integers 1 2 3 4 5 6 7 8 9 0 You have entered: 1 2 3 4 5
Self-Instructional Material 169 Arrays and Strings NOTES 6 7 8 9 0 Analyse
the program carefully. You declare s1 as an integer array of size 10. Next
you
use the first for loop to scan the entered integers from the keyboard. At the first iteration, s1[0] will be received and stored at location & s1[0]. This is similar to a simple variable where "&" denotes the address of the variable. This is repeated till s1[9] is received and stored at & s1[9]. The next for loop prints the value of s1[0] to s1[9], i.e., 10 integers one at a time and in new line. Thus the array is handled the same way and each element has a distinct identification. The elements of an array
ha0ve

89% **MATCHING BLOCK 46/126** W

the same name with a subscript corresponding to the position, i.e., the array name with the subscript written in square brackets. Consider another program to understand one-dimensional arrays more clearly. Assume the existence of an array of elements. You want to find out the greatest number in the array and its location. To do that we set up two other variables known as max and ind. You initialize them to zero. Then you compare each number with max. If it is greater than the max, then we note the location in ind and the associated value in max. When you have checked all the elements in the array, you will get the greatest number and its location. Since the first element has a subscript 0, you have to add 1 to the subscript to get the position. The program is given below: /*Example 8.2 to find the greatest number and its position in an array*/ #include&gt;stdio.h&lt; int main() { int a[5]={1,5,2,6,3}; int max=0, i, ind=0; for(i=0; i&gt;=4; i++) { if (a[i] &lt; max) { max =a[i]; ind=i+1; } } printf("maximum number=%d location=%d\n", max, ind); } 170 Self-Instructional Material

Arrays and Strings NOTES

Result of the program maximum number=6 location=4 Let us see how the program works. Iteration 1 i = 0 max = 0 ind = 0 a[0] = 1 since a[0] &lt; max max = 1 ind = 1 Iteration 2 i = 1 max = 1 ind = 1 a[1] = 5 since a[1] &lt; max max = 5 ind = 2 Iteration 3 i = 2 max = 5 ind = 2 a[2] = 2 since a[2] &gt; max no change Iteration 4 i = 3 max = 5 ind = 2 a[3] = 6 Since a[3] &lt; max max = 6 ind = 4 Iteration 5 i = 4 Max = 6 ind = 4 a[4] = 3 since a [4] &gt; max no change The program prints: max = 6 location = 4 Thus, arrays are very useful for solving real problems encountered every day. 8.4

PASSING ARRAYS TO
FUNCTIONS There is no restriction in passing any number of values to a function; the restriction is only in the return of values from a function. Therefore, arrays can be passed to a function without any difficulty, one element at a time, as given below: #include &gt;stdio.h&lt; main() { int a[]={1,2,3,4,5}; int j; int func(int a);
Check Your Progress 1. What is an array? 2. How are arrays declared and named? 3. How are data elements written in an array declaration?
Self-Instructional Material 171 Arrays and Strings NOTES
for (j=0; j&gt;=4; j++) func (a[j]); .......... } int func(int c) { ...... } Here, func has been declared as a function passing a single integer. Note here, that the declaration or the prototype gives only the format of the parameters passed. The values are only indicative and are not actual values. They are the formal values. Therefore, the parameters declared inside the parentheses act only as a checklist. They cannot be used in the main function elsewhere without actually declaring them on top of the function. But for this rule, there would have been a conflict between a[], which is an array and a, which is a simple variable. Here no conflict arises because a is not recognized in the main function. It is
only a
checklist to see that whenever the function calls func, an integer has to be passed. If you try to pass a float, the compiler will detect an error. This is not so in the case of variables defined in the function declarator above the functions body as they are recognized as actual names. In this case, int c is declared as a variable in func. The initial value will be the same as passed by the calling function. Thus, since a is used in the function declaration, only one integer can be passed to the function func. Actually, the entire array can be passed to a function irrespective of its size by suitable declaration as the following example indicates: /*Example 8.3*/ /* to find the greatest number in an array*/ #include &gt;stdio.h&lt; int main() { int array[]= {8, 45, 5, 911, 2}; int size=5, max; int fung(int array[], int size); max=fung(array, size);
printf("%d\n", max); } int fung(int a1[], int size) { int i, j, maxp=0; for (j=0; j&gt;size; j++) { if (a1[j] &lt; maxp) { maxp=a1[j]; } } return maxp; }
172
Self-Instructional Material Arrays and Strings NOTES
Result of the program 911 The objective of Example 8.3 is to find the greatest number in an array. In the program an array is initialized with 5 values as given below: int array[]= {8, 45, 5, 911, 2}; size is declared as 5 and a function called fung() has been declared. It will pass an array and an integer to the called function. The array size has been kept open and the called function will return an integer. The next statement calls fung and passes all
the elements of the array and an integer 5 equal to siz e.
The function gets the actual values and size = 5. The maximum value in the array is found in the for loop and stored in maxp. The value maxp is returned to the main function and printed there. Thus the function is called by value. 8.5
MULTIDIMENSIONAL ARRAYS Multidimensional
arrays operate on the same principle as single-dimensional arrays. We have to give the dimensions of the two subscripts in case of a two-dimensional array. For example, w [10][5] The above

is a two-dimensional array with different subscripts. Here, there will be 50 different elements. The first element can be denoted as w [0][0]. The next element will be w [0][1]. The fifth element will be w [0][4]. The sixth element will be w [1][0]. The last element will be w [9][4]. This can be considered as a row and column representation. There are 10 rows and 5 columns in this example. When data is stored in the array, the second subscript will change from 0 to 4, one at a time, with the first subscript remaining constant at 0. Then the first subscript will become 1 and the second subscript will keep increasing from 0 to 4. This is repeated till the first subscript becomes 9 and the second 4. This array can be used to represent the names of 10 persons, with each name containing 5 characters. The first subscript refers to the name of the 0th person, 1st person, 2nd person and so on. The second subscript refers to the 1st character, 2nd character and so on of the name of a person. Thus, 10 such names can be stored in this array. The dimension of the array can be increased to 3 with 3 square brackets as given below: Marks [50][3][3]; The name of the first element will be Marks [0][0][0] The last element will be Marks [49][2][2]. It would be easy to add more dimensions to an array but it would also become more difficult to comprehend under normal circumstances. It may, therefore, be useful to solve complicated scientific applications. Now let us understand the concept of multidimensional arrays using a simple problem. Assume that you need to write a program to read two arrays (both two-dimensional) and multiply the corresponding elements and store them in another

two-

dimensional array. To make the problem simpler, we will use [2][2] arrays.
Self-Instructional Material 173 Arrays and Strings NOTES Let us call the arrays x, y and
z. We have, x = {x[0][0] x[0][1]} y = {y[0][0] y[0][1]} {x[1][0] x[1][1] } {y[1][0] y[1][1]} You want to multiply x [0][0] and y [0][0] and store the result in z [0][0] and so on. The values of x and y are given in the program itself. /*Example 8.4*/ /* multiplication
of two 2 dimensional arrays*/ #
include &gt;stdio.h&lt; int main() { int i,j; int z[2][2]; int x[2][2]= {1, 2, 3, 4}; int y[2][2]= {5, 6, 7, 8}; for (i=0; i&gt;=1; i++) { for (j=0; j&gt;=1; j++) {
z[i][j]=x[i][j]*y[i][j]; printf("z[%d][%d]=%d\n", i, j, z[i][j]); } } }
You have declared 2 arrays x[2]
and

**88%** **MATCHING BLOCK 49/126** W

y[2] as follows: x = { 1 2 } y = { 5 6 } { 3 4 } { 7 8 } x [0][0] = 1 x [1][1] = 4 y [0][0] = 5 y [1][1] = 8 Therefore, after multiplication of the respective elements, we get, z = {5 12} {21 32} The program prints out the values of the products stored in array z. Result of the program z[0][0]=5 z[0][1]=12 z[1][0]=21 z[1][1]=32 Note that the elements are stored row by row

contiguously.
174 Self-Instructional Material Arrays and Strings NOTES
In the above example, we have declared elements with a two-dimensional array and initialized its one-dimensional array as given below: x [2][2] = {1, 2, 3, 4}; The system correctly interpreted the same and we got the correct result. We can actually present the above in another manner as given below: int x [2] [2] = { {1, 2}, {3, 4} }; In this method, you are indicating the elements closer to matrix form. Both the above definitions are equivalent. In the latter definition, we can easily visualize a two- dimensional array. The first row represents first row of the two-dimensional array. The values in the second row represent the second row of the two-dimensional array. To practice the above representation, let us take another example of a two- dimensional array. x [3][2] = {10, 20, 30, 40, 50, 60}; The same can be represented in the second form as given below: x [3][2] = { {10, 20}, {30, 40}, {50, 60} }; Note that there is no comma at the end of the last row. 8.5.1
Triangular Matrices Upper Triangular Matrix You may recall that a principal or leading diagonal of a square matrix A contains the elements whose row index and column index are same, i.e., includes the elements A[1][1], A[2][2], A[3][3], etc. A square matrix in which all the elements below the leading diagonal are zero is called an upper triangular matrix. For example, the following is an upper triangular matrix: {10, 20, 30} { 0, 50, 60} { 0, 0, 40} Notice that all the elements below the leading diagonal are zero. This can be mathematically expressed as A [i] [j] = 0 for i&lt;j Now let us write an algorithm to print the elements of an upper triangular matrix. Essentially, we need not print
out
those elements, which we know will be zero in an upper triangular matrix.
The
program implementing the above algorithm is given below: /*Example 8.5*/ /* printing elements of upper triangular matrix*/ #include&gt;stdio.h&lt; int main()
Self-Instructional Material 175 Arrays and Strings NOTES {
int i,j; int row; int A[10][10]; printf("Enter number of rows of given matrix\n"); scanf("%d", &row); /*getting given matrix row by row*/ for(i=0; i&gt;row; i++) { printf("Enter row number %d of given
matrix\n", i+1); for(j=0; j&gt;row; j++) scanf("%d", &A[i][j]); } /*Printing values of matrix*/
printf("Elements of upper triangular matrix are:\n"); for(i=0; i&gt;row; i++) { printf("\n"); for(j=0; j&gt;row; j++) {if(i&lt;j) continue; else printf("A[%d][%d]=%d\t", i,j, A[i][j]); } } } Let us take a square matrix of size 4 × 4. However, we will enter all the values of the matrix. The program will print out only the non-zero values along with the index. Result of the program Enter number of rows of given matrix 4 Enter row number 1 of given matrix 1 2 3 4 Enter row number 2 of given matrix 0 5 6 7 Enter row number 3 of given matrix 0 0 8 9 Enter row number 4 of given matrix 0 0 0 10 Elements of upper triangular matrix are:
A[0][0]=1 A[0][1]=2 A[0][2]=3 A[0][3]=4 A[1][1]=5 A[1][2]=6 A[1][3]=7 A[2][2]=8 A[2][3]=9 A[3][3]=10
176
Self-Instructional Material Arrays and Strings NOTES Lower Triangular Matrix A square matrix in which all the elements above the leading diagonal are zero is called a lower triangular matrix. For example, the following is a lower triangular matrix: {10, 0, 0} {60, 50, 0} {40, 30, 70} Notice that all the elements above the leading diagonal are zero. This can be mathematically expressed as A [i][j] = 0 for i&gt;j. 8.5.2
Matrix Multiplication
Recall that a matrix multiplication is possible when the number of columns in the first matrix is equal to the number of rows in the second matrix. Let us understand the problem with an example. Let matrix, A [2] [3] = ⎡ ⎤⎥ ⎢⎣ ⎦10 20 30 40 50 60 B [3] [2] = ⎡ ⎤⎥ ⎢⎥ ⎢ ⎥ ⎢⎣ ⎦1 2 3 4 5 6 Number of columns in matrix A = 3 Number of columns in matrix B = 3 Hence, multiplication
A ×

B is possible. Let P be the product matrix. Let us use C notation for subscript, i.e., the first element will have a subscript of zero. P [0] [0]= [10 × 1 + 20 × 3 + 30 × 5] = 220 P [0] [1]= [10 × 2 + 20 × 4 + 30 × 6] = 280 P [1] [0]= [40 × 1 + 50 × 3 + 60 × 5] = 490 P [1] [1]= [40 × 2 + 50 × 4 + 60 × 6] = 640 Thus, multiplication of 2 × 3 matrix by 3 × 2 matrix results in 2 × 2 matrix as given below: P [2][2] = ⎡ ⎤⎥ ⎥⎣ ⎦ 220 280 490 640 We can generalize the calculation of each element as given below: P [i] [j] = n – 1 = ΣA(i, k) x B(k, j) k = 0 Where n = number of columns of the first matrix as well as the number of rows of the second matrix.
Self-Instructional Material 177 Arrays and Strings NOTES i = Number of rows of the first matrix and product matrix. j = Number of columns of
the
first matrix and product matrix. Let us confirm this through an example. For the sake of simplicity, the values are given in the program itself. /*Example 8.6 To demonstrate matrix multiplication*/ #include&gt;stdio.h&lt; int main() { int i,j,k; int A[2][3]={ {10, 20, 30}, {40, 50, 60} }; int B[3][2]={ {1, 2}, {3, 4}, {5, 6} }; int P[2][2]; /*Matrix multiplication*/
for(i=0; i&gt;2; i++)
for(j=0; j&gt;2; j++) { P[i][j]=0; /*Initializing product matrix*/ for(k=0; k&gt;3; k++) P[i][j]=P[i][j]+A[i][k]*B[k][j];/* calculating elements*/ } /*Printing values of Product matrix*/ printf("Elements of Product matrix are:\n");
for(i=0; i&gt;2; i++) for(j=0; j&gt;2; j++) { printf("P[%d][%d]=%d\n", i,j, P[i][j]); } }
We initialize matrices A
and
B with actual values. Then, we declare a product matrix P[2][2]. The dimension of the matrix requires a little explanation. We have, A [2] [3] and B [3] [2] Since the number of columns of the first matrix is same as the number of rows of the second matrix, we can multiply them. The resultant matrix will have dimensions as given below:
178
Self-Instructional Material Arrays and Strings NOTES P [
number of rows of the first matrix] [number of columns of the

**88%** **MATCHING BLOCK 50/126** **W**

second matrix] Assume that A[3][2] and B[2][4] The product matrix in this case will be P [3] [4]. This has to be understood clearly. The number of rows of

the
product matrix will be equal to the number of rows of the first matrix and the number of columns of the product matrix will be equal to the number of columns of the second matrix. Now, look at the section where we carry out matrix multiplication. We initialize the elements of the product matrix. Then each element value is calculated. We are using a nested for loops and inside them, another for loop. Finally, we print the values. The result of the program is given below: Result of the program Elements of the Product matrix are: P[0][0]=220 P[0][1]=280 P[1][0]=490 P[1][1]=640 8.6
STRINGS: ONE-DIMENSIONAL CHARACTER ARRAY We have passed an entire integer array into a function. We can pass any other type of arrays also. We can pass a float array as well as a character array. What is an array of characters? It is nothing but a string. A string is terminated by NULL or \0 whose ASCII value is zero. It is different from 0 whose ASCII value is 48. Strings are special type of one-dimensional arrays. As we know char is a basic data type. For example, char ch = 's'; The above expression declares a variable
ch of type char and initializes with a character constant s. A string is an array of characters, meaning that it can contain zero to many characters. A string is enclosed within double quotes. For example, char st[1] = "s"; The variable st is a string since it is of type array of characters. It is initialized with constant s. When we initialize a string variable as above, a NULL character will be appended to the end of the string automatically by the
compiler.

**95%** **MATCHING BLOCK 52/126** **W**

Let us now write a program to read a string with scanf() and write with printf(). The program is given below: /*Example 8.7 Reading and writing with formatted I/O functions*/ #include&gt;stdio.h&lt;

Check Your Progress 4. How is an array passed to a function? 5. Define two-dimensional/ multi-dimensional arrays. 6. Can arrays represent matrices? 7. Define upper triangular matrix. 8. Define lower triangular matrix. 9. Is matrix multiplication possible? How?
Self-Instructional Material 179 Arrays and Strings NOTES
int main() { char str[10]; printf("Enter your name\n"); scanf("%s", str); printf("You Entered: %s\n"); } Look at the program.
You
are declaring a string str as a character array of size 10. Look at the way you receive the array. You do not prefix ampersand (&) to the variable name str. You do not even suffix it with square brackets. Then, we print the contents of variable str. The format specifier in both the cases is %s. Look at the result of the program. Result of the program Enter your name Rama Krishnan You Entered: Rama You typed 'Rama Krishnan'. But you got 'Rama'; since the white space following the first word was considered to be
the end of the character array, the computer stopped reading thereafter. However, this can be overcome by using the unformatted I/O functions gets() and puts().

The example below gives the modified version of the above program. /*Example 8.8 Reading and writing with unformatted I/O functions*/ #include&gt;stdio.h&lt; int main() { char str[10]; printf("Enter your name\n"); gets (
str); printf("You Entered:"); puts(str); } Look at the program. You have declared str as one-dimensional character array.
You
read the string as shown below: gets (str); No format needs to be specified. Writing is also similar. The argument to puts() as well as gets() are of type string. Look at the result of the program. Result of the program Enter your name Rama Krishnan You Entered: Rama Krishnan The entire string has been received although there is white space in between. Thus gets() is more suitable for handling strings. But, if
you
have to receive two different strings, we need to call gets() twice. However, any number of strings can
180
Self-Instructional Material Arrays and Strings NOTES be received with one scanf() statement, and
it will treat white space as a string terminator. Similarly, puts() can display only one string at a time, whereas printf() can print any number of strings with one statement. 8.7
ARRAY OF STRINGS A string is a one-dimensional array of characters. The
names of students in a class can be denoted by a two-dimensional array like b[50][20], with the second subscript denoting the width of the names and the first 50 denoting the number of students.
This is nothing but array of strings. In the previous examples, you created strings, which are essentially one- dimensional array
of characters. You can create an array of strings. This will be two- dimensional arrays of characters. For example, char name[5][10]; This expression
declares a two-dimensional array of characters. This can be used to deal with 5 strings of size 10 each.
Given below is the program

**85%** **MATCHING BLOCK 53/126** **W**

to read 5 names and display them. /*Example 8.9*/ Two Dimensional array*/ #include&gt;stdio.h&lt; int main() { int i; char name[5][10]; /*Receiving strings*/ for (i=0; i&gt;5; i++) { printf("Enter name[%d]: \n", i+1); scanf("%s", name[i]); } /*displaying strings*/ for (i=0; i&gt;5; i++) { printf("name[%d]: %s\n", i+1, name[i]); } } Look at the program. You declare a two-dimensional character array called name[5][10]. Then you receive the names. Look at the ease with which we receive it. scanf("%s", name[i]); You do not even give the second dimension. This is possible only in the case of strings. Recall that str was a one-dimensional array. You read it just by specifying str. But since you are specifying scanf(),

you cannot give white spaces in between the names. Check Your Progress 10. Define string with reference to array. 11. What is library function strcpy() used for? 12. Define the precondition for binary search.
Self-Instructional Material 181 Arrays and Strings NOTES As you
know, the elements of an array will be stored contiguously. In name[i], you are specifying the address of 0th location of the ith row. The array received will be stored there on continuously. In the next section, you print each string the same way by specifying the first subscript alone. The result of the program is given below.
Result
of the program Enter name[1]: Ganapathy Enter name[2]: Subramani Enter name[3]: Narayanan Enter name[4]: Joseph Enter name[5]: Mohammed name[1]: Ganapathy name[2]: Subramani name[3]: Narayanan name[4]: Joseph name[5]: Mohammed The result demonstrates the use of two-dimensional character arrays. Be cautious not to exceed the size of the array. Although we did not enter 10 characters, the computer recognized the end of the string due to the NULL character generated by the pressing of Enter key each time. 8.7.1
Binary Search There are a number of search methods. Here you will learn about binary search. An important precondition for binary search is that the data is ordered or arranged in
either ascending or descending order in an array like the names, which
are arranged alphabetically in a telephone directory. Let us now study the algorithm for binary search to understand the concept. You will assume that the array of numbers is arranged in
the increasing order.
A program implementing binary search algorithm is given below: /*Example 8.10*/ /*Binary search- The numbers are ordered*/ #include &gt;stdio.h&lt; #define UPPER 5 #define TRUE 1 #define FALSE 0 int main() { int left =0, mid=0, right=UPPER, found=FALSE; int x; int a[]= {10, 20, 30, 40, 50, 60};
182
Self-Instructional Material Arrays and Strings NOTES

printf("ENTER THE NUMBER TO SEARCH\n"); scanf("%d", &x); while ((left &gt;= right) && (!found)) { mid = (left+right)/2; if (a[mid]==x) found = TRUE; else if (a[mid] &gt; x) left = (mid +1); else right = (mid-1); } if (found) printf("found the number %d in position %d\n", x, mid+1); else printf("NOT found the number %d", x); } Result of program when searched for a number in the array: ENTER THE NUMBER TO SEARCH 60 found the number 60 in position 6 Result of program when searched for a number NOT in the array: ENTER THE NUMBER TO SEARCH 25 NOT found the number 25 Try to understand how the program functions. It implements the algorithm truthfully. 8.8

SUMMARY In this unit, you have learned about arrays and strings. An array is the form of data type and is used to define
arrays of integers, arrays of characters and arrays of floating point numbers, etc.
An important feature of an array is that it contains data of the same type. Further, an array is a variable and hence it should be declared like any other variable of C programming. The arrays can be given a specific
name. It cannot be a reserved word and must be unique in the program. An array variable name and another ordinary variable name cannot be identical.
An array variable is distinguished from a single variable on the basis of declaration. Array dimensions are declared within the square brackets and each dimension element is separated by a comma. The elements in an array are stored in consecutive memory locations. Multidimensional
arrays operate on the same principle as single-dimensional arrays. You have to give the dimensions of the two subscripts in case of a two-dimensional array. You learnt that arrays can be used to represent matrices. You can define upper and lower triangular matrices.
A square matrix in which all the elements below the leading diagonal are zero is called an upper triangular matrix
whereas
a square matrix in which all the elements above the leading diagonal are zero is called a
Self-Instructional Material 183 Arrays and Strings NOTES lower triangular matrix. You also learnt
that matrix multiplication is possible when the number of columns in the first matrix is equal to the number of rows in the second matrix.
Strings refer to a special type of one-dimensional array.
A string is an array of characters, meaning that it can contain zero to many characters
and is enclosed within double quotes. Finally in this unit, you learnt binary search methods.
An important precondition for binary search is that the data is ordered or arranged in
either ascending or descending order in an array. 8.9 KEY TERMS • Array: It is a variable and contains
data of the same type. The data type includes arrays of integers, arrays of characters and arrays of floating point numbers, etc. • Upper triangular matrix: It is
a square matrix in which all the elements below the leading diagonal are zero. • Lower triangular matrix: It is a square matrix in which all the elements above the leading diagonal are zero. •
String: It is special type of one-dimensional array of characters and is enclosed within double quotes. • Binary search: It is a search method in which the data is ordered or arranged in either ascending or descending order in an array. 8.10 ANSWERS TO 'CHECK YOUR PROGRESS' 1. Array is another form of data type. There
can be arrays of integers, arrays of characters and arrays of floating point numbers, etc.
An array contains data of
the same type. 2. An array is a variable and hence must be declared like any other variable. The naming convention is the same as in the case of other variables. The array name cannot be a reserved word and must be unique in the program. An array variable name and another ordinary variable name cannot be identical. Since there is no limit to variable names, do not use similar names for a variable and an array. What distinguishes an array variable from a single variable is the
declaration of the dimension within the square brackets. 3.
The data elements are written within braces separated by commas. 4.
There is no restriction in passing any number of values to a function; the restriction is only in the return of values from a function. Therefore, arrays can be passed to a function without any difficulty, one element at a time. 5.
Multidimensional
arrays operate on the same principle as single-dimensional arrays. We have to give the dimensions of the two subscripts in case of a two-dimensional array. For example, w [10][5] 6.
Arrays can be used to represent matrices. Inter-changing rows and columns in a matrix
results in transpose of a matrix.
184 Self-Instructional Material Arrays and Strings NOTES If 1 2
A 3 4 ⎡ ⎤ = ⎢ ⎥ ⎣ ⎦ A = A t = 1 3 2 4 ⎡ ⎤ ⎢ ⎥ ⎣ ⎦ 7. A square matrix in which all the elements below the leading diagonal are zero is called an upper triangular matrix. For example, the following is an upper triangular matrix: {10, 20, 30} { 0, 50, 60} { 0, 0, 40} 8.
A square matrix in which all the elements above the leading diagonal are zero is called a lower triangular matrix. For example, the following is a lower triangular matrix: {10, 0, 0} {60, 50, 0} {40, 30, 70} 9.
Matrix multiplication is possible when the number of columns in the first matrix is equal to the number of rows in the second matrix. 10.
Strings are special type of one-dimensional arrays.
A string is an array of characters, meaning that it can contain zero to many characters. A string is enclosed within double quotes. 11.

Library function strcpy() is available for this purpose. 12.
An important precondition for binary search is that the data is ordered or arranged in
either ascending or descending order in an array like the names, which are arranged alphabetically in a telephone directory. 8.11
QUESTIONS AND EXERCISES Short-Answer Questions 1. State whether True or False: (
a) An integer array can contain int, char and long. (b) An array of strings is not possible. (c) &s1[10] is the address of the 11th element of the array variable s1. (d) puts can print a number of variables in a single statement like printf. (e) In a linear search, n comparisons are to be made for searching a value in an array of elements of size n. (f) The if (flag) means whether flag = 1. (g) A search program cannot be solved easily without using arrays. (h) In a square matrix, the number of rows equal the number of columns. 2. What are arrays? Why are they called? 3. Explain multidimensional arrays. 4. What is binary search?
Self-Instructional Material 185 Arrays and Strings NOTES Long-Answer Questions 1. Algorithm for binary search is given below: Step 1: left = 0 right = n – 1 Step 2: while (left &gt; right ) { mid = (left + right)/2 if (a[mid] &gt; x) left = mid + 1 ; else right = mid }
Step 3: if (a [right]= = x), print found else print x not found Convert this to a C program and test whether it works. 2. Write programs for the following: (a) To compare two given words. [Hint: Store the words as array of characters, compare the respective characters, one after another.] (b) To pass the radius of a circle to a function and calculate the area and perimeter and print them in the called function. (c) To pass a string along with the position number to a function and return the string deleted up to the position. (d) To pass a string along with the position number up to which the string has to be printed by use of a function. (e) To pass array of integer to a function along with an index to get back the element with the index and index +1. 3. Write a program to check whether the diagonal elements of a (4 × 4 ) matrix are all zero. 4. A program will be given 10 words of varying length. Write a program, which will arrange the words in the ascending order of word length. 5. To accept a real number and print the sum of digits (whole number and fraction parts) of the number. 8.12
FURTHER READING Gottfried, Byron S. Programming with C, 2nd edition. Schaum's Outline Series. New York: McGraw-Hill, 1996. Forouzan, Behrouz A. and Richard F. Gilberg. Computer Science: A Structured Programming Approach Using C, 2nd edition. California: Thomson, Brooks Cole, 2000. Dey, Pradip and Manas Ghosh. Computer Fundamentals and Programming in C. New Delhi: Oxford Higher Education, 2006. Bronson, Gary J. A First Book of ANSI C, 3rd edition. California: Thomson, Brooks Cole, 2000.
186 Self-Instructional Material Arrays and Strings NOTES Kanetkar, Yashwant. Understanding Pointers in C.
New Delhi: BPB Publication, 2001. Kanetkar, Yashwant. Let us C. New Delhi: BPB Publication, 1999.
Kernighan, Brian W. and Dennis Ritchie. C Programming Language, 2nd edition. New Jersey: Prentice Hall, 1988. Foster, W. D. and L. S. Foster. C by Discovery. Boston: Addison-Wesley, 2005. Kanetkar, Yashwant. Working with C. New Delhi: BPB Publication, 2003. Horton, Ivor. Instant C Programming. New Jersey: Wrox Press (John Willey & Sons), 1995. Lawlor, Steven C. The Art of Programming Computer Science with 'C'. New Jersey: West Publishing Company, 1996.
MODULE – IV
188 Self-Instructional Material Pointers NOTES
Self-Instructional Material 189 Pointers NOTES UNIT 9 POINTERS Structure 9.0 Introduction 9.1 Unit Objectives 9.2 Pointer Fundamentals 9.2.1 A Pointer is also a Variable 9.3 Pointer to Void 9.4 Null Pointer 9.5 Pointer Arithmetic 9.6 Passing Pointers to Functions 9.7 Pointers and Functions 9.7.1 Function Declaration 9.7.2 Function Declarator 9.7.3 Function Call 9.7.4 Return Statements 9.8 Pointers and One-Dimensional Array 9.8.1 Finding the Greatest Number in an Array 9.9 Pointer Notations for Arrays 9.9.1 Arrays and Pointers 9.10 Multidimensional Arrays 9.10.1 Receiving Inputs at Chosen Points 9.11 Pointers and Strings 9.11.1 String Functions 9.11.2 To print a Substring 9.11.3 To analyse a Text File 9.12 Array of Pointers 9.12.1 Sorting Character Strings 9.13 Dynamic Allocation of Memory 9.14 Pointer Comparison 9.15 Structure Pointers 9.16 Summary 9.17 Key Terms 9.18
Answers to 'Check Your Progress' 9.19 Questions and Exercises 9.20 Further Reading 9.0 INTRODUCTION In this unit, you will learn about pointers. The pointer is a powerful concept of C and is closely associated with memory addressing. It is an integer variable, which contains the address of a variable. These variables are stored in the memory and each location in the memory has an address. It can point to any data type. The storage capacity varies from machine to machine.
You will learn how a pointer to void can be declared. Void is a keyword and it means 'nothing'. Therefore, the void pointer points to nothing but it is very useful when we assign address of any data type to a pointer to void. Null pointers
can take any pointer type, but do not point to any valid reference or memory address.
190
Self-Instructional Material Pointers NOTES
Pointer arithmetic will make the concept of pointers clear and unambiguous. You can call a function and pass actual values to them. The function call using pointers is known as call by reference where the address of the variable is passed. Functions can also return reference or pointers.
The ** indicates pointer to pointer. You will learn that array manipulation becomes easier using pointers. Both the array of numbers and array of characters, i.e., strings are explained.
Static allocation of memory in case of arrays through dimension leads either to excessive allocation or short allocation. This can be managed
by dynamically
allocating memory at run-time through the functions namely malloc() and calloc(). The memory allocated using these functions can be freed, when it is not required using the function free().
Further you will learn about pointers to functions and pointer comparison function to compare their addresses. 9.1 UNIT
OBJECTIVES
After going through this unit, you will be able to: • Understand the basic concept of
pointers • Declare

pointers • Explain why pointer a is a variable • Define pointer to void • Understand and use null pointer and pointer arithmetic • Pass pointers to functions • Understand the relationship between pointers and functions, pointers and one- dimensional arrays, pointers and strings • Define pointer notations for arrays and array of pointers •
Explain dynamic allocation of memory • Compare pointers • Define structure pointers 9.2 POINTER FUNDAMENTALS The pointer is a powerful concept of C. It is powerful because of the following two reasons: • It helps in achieving results, which could not otherwise be achieved, such as an indirect method for returning more than one value from a function. • It results in a compact code. Pointers are closely associated with memory addressing. We know that variables
are stored in the memory and that each location in the
memory
has an address just as a person has an address. Memory locations are available in groups of 8 bits or a byte. Each byte in memory has an address. Therefore, each location has an address and stores a value. The value stored can correspond to any data type, such as float or char or int, and their type modifiers. However,
as you know,
all these data types are stored in terms of 1s and 0s.
The memory locations are arranged in an increasing order of addresses starting from 0000, which increases one by one. You may have the address of the last location as FFFF. What does F denote? The addresses are denoted in terms
Self-Instructional Material 191 Pointers NOTES
of hexadecimal numbers. You can calculate the decimal equivalent of this and find the decimal address of the last location. The storage capacity varies from machine to machine. The & operator has been used to denote the address of the variable in the scanf() function. If var is the name of a data type, &var denotes the address of the location where it is stored. Whenever a data type is declared, a memory location
is
allocated depending on the data type. The allocation takes place at the time of execution of the program and therefore, the address of a variable may be different at different times of execution, since the computer allots it at random depending on the availability of a memory location. However, memory locations for each data type will be contiguous so as to make for easy handling. For example, a double needs 8 bytes of storage and therefore all the 8 bytes will be stored continuously or contiguously. Each byte of the value of the variable will be stored one after the other in continuous locations. For example, the following is
declaration of float type:
float ft = 100.52; Here ft is the name of the variable. Its value is 100.52. This number will be stored in 4 bytes in the memory in sequentially numbered storage locations. When
you
print the value of the address of a variable, it will always print the starting address. If the starting address and the data type are known, it is easy to find out the locations, which are occupied
by the
variable. For example, if the starting address of the float ft is 0011, then the variable will occupy up to memory location 0014. However, if
you
print the address of ft, i.e., &ft, you will get 0011. A similar
concept can be extended to a string. A string is an array of characters with each character occupying a byte of memory. If Peter is stored in string w, and if w[0], i.e., p is stored in location 0020, then 'e' will be stored in 0021, 't' in 0022 and so on. Every element of an array will also be stored contiguously. The formula for finding out the location of an element or address of pth element in an array is given below: Address of
the

| 93% | MATCHING BLOCK 55/126 | W |
|---|---|---|

pth element = Starting address of array + p * (storage space for the data type) Example 1: If an integer array ia is stored from location 992 onwards, find out the location of the 10th element. Address of the 10th element = 992 + 10*2 = 1012. It is the starting point. However, the second byte of the integer will be stored in location 1013. Note: The 10th element will have the subscript 9, since the 1st element has the subscript 0.

Example 2: If the 10th element of a long double is stored from location 2000 onwards, find the location of the 15th element and the first element. The 1st element will be stored at location 2000 – 10*10 = 1900. The 15th element will be stored at 1900 + 15*10 = 2050. Now consider a pointer to an integer. Let the integer be mark. Then the address will be denoted as &mark. Note that all
the addresses will be in integers for all the
data types. In the case of pointers, the address of mark will be stored in another location. The
pointer is a variable that contains the address of the variable. You can assign the address of the
integer to an integer pointer.
Generally you declare as follows.
192 Self-Instructional Material Pointers NOTES

87% MATCHING BLOCK 56/126 W

int mark; mark = 75; you can also declare int *ip; This means ip is a pointer to the integer. You can assign ip = &mark; i.e.,

you
have assigned the address of mark to ip. This can be explained pictorially as follows: 1011 1030 Here mark = 75 and the address of mark is 1011. Therefore, ip = 1011. This value will also be stored at another location 1030. Here ip points to an integer mark and holds the address of mark. Since the pointer is also a variable, it
will be stored in another location. The * is called the indirection or dereferencing operator. Similarly,
you
can write, float f = 101. 23; float * fp; fp = &f; Here fp points to a float because we have assigned the address of f to fp. Remember that the pointer can point to any type of variable such as float or char or int or string. Pointers themselves are always of type int because it is the value of the address. It is necessary to become familiar with pointers. Therefore, let us apply the concepts learnt. You can have the definitions of the following types: int i = 204; float f = 101. 23;
int * ip; / * ip is a pointer to integer */ float * fp; /* fp is a pointer
to float */ You must assign the pointers to the specific integers and floats, as otherwise they will not point to anything. This is carried out as follows: ip = &i ; fp = &f ; By assigning ip to the address of i, ip points to integer i. Similarly, fp points to float f. Suppose
you
now assign, i = 100 ; ip automatically points to 100. Similarly, if you assign, f = 100.05; then fp points to the new value. What actually happens? The variables i and f are assigned storage locations; ip holds the address of where i is stored and fp holds the address of f. When we assign new values to i and f, the values stored in ip and fp are not affected. They continue to point to i and f, but the values of i and f have been actually changed.
You
can also perform arithmetic operations on pointer variables such as: ip = ip + 5 ; /*pointer moved up by 5 locations*/ ip = ip – 10; /* ip moved down by 10 locations/ 75 1011
Self-Instructional Material 193 Pointers NOTES
ip––; /*decremented*/ ip++; /*pointer incremented*/ *ip++; /*value
incremented*/ *ip––; /*value decremented*/ However, such operations on pointers are limited.
You

93% MATCHING BLOCK 57/126 W

cannot carry out the following operations on pointers: ip+fp; /*invalid*/ ip*fp; /*invalid*/ ip*2; /*invalid*/ fp/10; /*invalid*/ ip = ip*10; /*invalid*/ If

you
say ip = fp; then both fp and ip will point to the same location and hence, fp will point to the same variable pointed to by ip. 9.2.1
A Pointer is also
a
Variable
A pointer

97% MATCHING BLOCK 58/126 W

is a variable that contains the address of another variable. As you know, any variable has the following four properties: (a) Name (b) Value (c) Address (d) Data type For example, consider the following declaration of a simple integer: int var = 10; The name here is var, and its value is 10. Its address is not declared here, since we want to give flexibility to the compiler to store it wherever it wants. If we specify an address, then the compiler must store the value at the same address. Specifying the actual address is carried out during machine language programming. However, this is not required in high-level language (HLL) programming, and by printing the value of &var, we can find out the address of the variable. When the statement to find the address is executed at different times, different addresses will be printed. What is important is that the compiler allocates an address at run-time for each variable and retains this till program execution is completed. This is not strictly so in the case of auto variables. The compiler forgets the address of a variable when the program comes out of the block in which the variable is declared. At this point you may also recall that in the case of function declarations in the calling function, the compiler does not allocate memory to the variables in the declaration. That is the reason why the parameters in the declaration part are not recognized in the calling function. It is only a prototype. The fourth feature of a variable is its data type. In the above example, var is an integer. A pointer has all the four properties of variables. However, the data type of every pointer is always an integer because it is the value of the memory address. Memory addresses are integers. They cannot be floats or any other data types. They may point to an integer or a float or a character or a function, etc. They have a name. They have a value. For example, the following is a valid declaration of a pointer to an integer. int * ip; 194 Self-Instructional Material Pointers NOTES Here ip is the name of a pointer. It points to or contains the address of an integer, which is the value. It will also be stored in another location in

the
memory like any other variables. The pointer itself is an integer even though it is not declared as such. 9.3 POINTER
TO VOID A pointer to void can be declared in the same way as you declare any other pointer such as pointer to integer, float, etc. void* void_pointer; Note that void is a keyword and it means 'nothing'. Note also that void pointer also points to a memory address and hence, it is also an integer like any other pointer. The void pointer points to nothing. Then what is the use of this pointer? If you try to assign address of an integer variable to a float pointer or any other type of pointer other than a pointer to an integer, then there will be an error. This rule applies to the address of variables of other data types such as double variable, char variable, etc. However, there is an exception to the rule and that is where a void pointer is useful. You can assign address of any data type to a pointer to void. For example, you can declare as given below: void* void_pointer; int int_var; void_pointer= & int_var; The above assignment of the address of an integer variable to a pointer to void is permissible. Similarly, we can assign a pointer to any type, such as float, double, etc. to a void pointer. There are times when you write a function but do not know the data type of the returned value. When this is the case, you can use a void pointer. A void pointer is a special type of pointer that has flexibility of pointing to any data type. However, there is one limitation in the use of void pointers as compared with pointers to other types, namely direct dereferencing of a void pointer is not permitted. The programmer must change the pointer to void as any other pointer type that points to valid data types, such as, int, char, float, etc. and then dereference it. Then conversion of the pointer to some other valid data type is achieved by using the concept of type-casting. 9.4 NULL POINTER The concept of NULL pointer is different from the above concept of void pointer. A NULL pointer is a type of pointer of any data type and generally takes a value as zero.
This is, however, not mandatory. This denotes that a NULL pointer does not point to any valid memory address. For example, int* var; var =0; The above statement denotes var as an integer pointer type that does not point to a valid memory address. This shows that var has a NULL pointer value. Check Your Progress 1. Define pointer. 2. What are memory locations? How are values stored in each location? 3. How is a pointer a variable?
Self-Instructional Material 195 Pointers NOTES The following is the difference between void pointers and NULL pointers. A void pointer is a special type of pointer of void and denotes that it can point to any data type. NULL pointers
can take any pointer type, but do not point to any valid reference or memory address. It is
important to note that a NULL pointer is different from a pointer that is not initialized. 9.5 POINTER ARITHMETIC The following are the examples of pointers: Example 3 int dat = 100; int * var; var = &dat; Here dat is an integer variable. Its value is 100; its name is dat; it will be stored in
the
memory in a location with an address. The next declaration means that var is a pointer to an integer and is
also
a variable. It is an integer and will be stored at a location in
the
memory with an address. The value of var is the address of the integer variable it points to. We do not know as yet which integer it points to. It can, however, be made to point to any integer we like by a proper declaration. Now look at the next assignment. The variable var is assigned the value of &dat. This means var has the same value as the address of dat. By taking into consideration the previous statements we can conclude that var is a pointer and it points to dat. Now if you specify dat or * var, they point to the same value 100. Similarly, if you specify &dat or var, it is the address, or to be precise, the starting address of dat or * var. Example 4 int * var * var = 100 The above statements declare var as a pointer to an integer and later propose to assign 100 to the integer variable. We do not know or do not want to make public the name of the variable. However, we can always access the variable as * var. This works well in Borland C++ compilers, but could lead to run-time errors in other compilers. Both the statements cannot be combined into one as int * var = 100. This will be flagged as an error even in the Borland C++ compiler. Therefore, it is not possible to combine both the declaration and assignment so far as a pointer variable and
an

58% **MATCHING BLOCK 59/126** W

integer constant are concerned. It would be safer to make var point to another variable as given in Example 3. Example 5 int * var; * var = 100; Note: If this gives an error while compiling or running the program, modify this as in Example 3. 196 Self-Instructional Material Pointers NOTES What will be the value of the output of the following statements after the execution of the following statements? printf ("%d", * var); printf ("%d", (* var) ++); printf ("%d", * var);

printf ("%d", var); You
can easily guess that the first printf() will give the value of * var as 100. What is the significance of the parentheses and the increment operation in the second statement? As the bracket or parentheses has precedence over other operators, the value of * var will be printed as 100. After printing, it will be incremented as 101 and because the increment is postfix, the value of * var after the execution of the statement will
be 101. The next statement will confirm this when it prints 101. The fourth printf() case prints the address of var. It
will print 1192 when the program is executed. You
are unlikely to get the same address on execution. The location where the variable is stored will not vary till the execution of the program is completed. It you try the program again, you will get a different address.
Do not worry. It does not affect our work. However, you

may note down the value and substitute it for the values mentioned here for understanding the concept of pointers. Remember to enclose the pointer variable within parenthesis as given in the example. The postfix of the increment operator enables increment of the variable after printing. Example 6 After the execution of the above fourth printf() statement, we execute the following statements: printf ("%d", * var ++); printf ("%d", * case); What happens? The value of * var, i.e., 101, is the first to be printed out and then var will be incremented, i.e., instead of incrementing the value as desired, the address is incremented, and therefore, var now points to the next location. Remember var was pointing to 1192. Will it go to 1193? No. Since var is an integer, 1192 and 1193 (2 bytes) are already occupied. Hence, var points to 1194. The next statement prints the value, of * var. We had stored 101 in location 1192. We do not know what is contained in 1194. Hence, the next printf() will print garbage value. Note carefully what had happened. We wanted to increment * var in the first statement of example 4. However, the compiler has assumed that we wanted to increment the pointer, which underlines the importance of parenthesis. Also note that whenever
you use the postfix notation, the postfix operation is effective only after the
execution of the statement. Is everything lost now? Can we not go back to address 1192? Yes, you can, as the following indicates. Note that the following statements are executed in continuation of all the above statements. Example 7 printf ("%d", var); printf ("%d", – – var); printf ("%d", * var);
Self-Instructional Material 197 Pointers NOTES The first statement in this example prints the address of the location in the memory pointed to by var. As expected, the pointer is at the location 1194. The second statement carries out two operations in the sequence given below: (a) Decrement the pointer. (b) Print the new address. Decrementing takes place before printing because it is a prefix operator. Now it prints 1192
and

| 93% | MATCHING BLOCK 60/126 | W |
|---|---|---|

the original address is restored. Now the third statement prints the value of * var or the value stored in location 1192, i.e., 101. Note that prefix carries out the increment or decrement before printing or any desired operation, whereas postfix does that after printing. Note that we are discussing the fundamentals of pointers and they should be understood clearly before you proceed further. We now have 101 stored in address 1192 and pointed to by var. Let us see what happens on execution of the following statements in continuation. Example 8 printf ("%d", ++ (* var)); printf ("%d", var); These statements are perfectly correct. * var is incremented and the new value printed in the first statement. Therefore, 102 will be printed. Has the address been changed? No. Hence the second statement will print the address as 1192. Can we increment and decrement addresses? Yes, as the following indicates: Example 9 printf ("%d", var ++); printf ("%d", var); What will be printed in the first statement above, 1192 or 1194? It will be 1192, because incrementing var will take place after the first printf(). Obviously, the second statement will print the address incremented after the previous printf(), viz. 1194. Let us not lose track, but get back to the old address, and try prefixing increment/ decrement operators to the address. Example 10 var – –; (a) printf ("%d", var); (b) printf ("% d", ++ var); (c) printf ("% d", – – var); (d) printf ("% d", var); (e) printf ("% d", * var) (f) Before the execution of the first statement in this example, you have: var = 1194 Location 1192 contains 102. Let us now analyse the execution statement-wise. (a) Decrements var to 1192. (b) Confirms that var is 1192 indeed. (c) ++ var, increments var and then prints as 1194. 198

Self-Instructional Material Pointers NOTES (
d) – – var decrements var and then prints as 1192. (e) Confirms var is 1192. (f) The value in var is 102. Now the concepts are becoming clearer. Let us carry out one more example. Assume that all the above statements were executed and the following are now executed: Example 11 printf ("%d", * (var ++)) ; (g) printf ("%d", * (– – var)) ; (h) printf ("%d", var) ; (i) printf ("%d", * var) ; (j) (g) Here * var is printed as 102 because of the postfix operator. Then var, i.e., the address is incremented to 1194. (h) Here because of the prefix, var is decremented to 1192 and then the value at 1192, i.e., 102 is printed. The next two statements confirm this.
Now you are familiar with the intricacies of pointers, prefix, suffix and parenthesis. For your convenience, the

| 93% | MATCHING BLOCK 61/126 | W |
|---|---|---|

program involving all these statements and the output is given below: /*program 9.1 */ /* pointers*/ #include &gt;stdio.h&lt; main() { int * var; int a =100; var = &a; printf("value of * var=%d\n", *var); printf("value of (*var)++=%d\n", (*var)++); printf("value of * var=%d\n", *var); printf("address var=%d\n", var); printf("value of *var++=%d\n", *var++); printf("value of * var=%d\n", *var); printf("address var=%d\n", var); printf("original address var again=%d\n", —var);/ *original address restored*/ printf("value of * var=%d\n", *var); printf("value of ++(*var)=%d\n", ++(*var)); printf("address var=%d\n", var); printf("address var++=%d\n", var++); printf("address var=%d\n", var); var—; printf("address var after decrementing=%d\n", var); printf("address ++var=%d\n", ++var); Self-Instructional Material 199 Pointers NOTES printf("address – –var=%d\n", – –var); printf("address var=%d\n", var); printf("value of * var=%d\n", *var); printf("value of *(var++)=%d\n", *(var++)); printf("value of *(– –var) = %d\n", *(– –var)); printf("address var=%d\n", var); printf("value of * var=%d\n", *var); } Result of the program value of * var=100 value of (*var)++=100 value of * var=101 address var=9106 value of *var++=101 value of * var=9108 address var=9108 original address var again=9106 value of * var=101 value of ++ (*var)=102 address var=9106 address var++=9106 address var=9108 address var after decrementing=9106 address ++var=9108 address – –var=9106 address var=9106 value of * var=102 value of * (var++)=102 value of *(– –var) = 102 address var=9106 value of * var=102 9.6

PASSING

POINTERS TO FUNCTIONS You have discussed various aspects of pointers such as pointers to variables, functions returning pointers, etc. What then is a pointer to a function? Consider the following program: #include &gt;stdio.h&lt; main() { char y='a'; void func ( char z); void (*fp) ( char x); fp = func ; (*fp) (y) ;
Check Your Progress 4. How is a pointer to void declared? 5. What will void pointer do? 6. Define NULL pointer. 7. Differentiate between void pointer and NULL pointer. 8. What will an increment operator do? 9. Define decrementing.
200 Self-Instructional Material Pointers NOTES }
void func ( char x) { ... } A perusal of the above program indicates that the address of func is assigned to the function pointer fp. While calling the function, (*fp) (y) is used. Thus, fp gets the address of func indicating that functions also have addresses. Such function pointers are used in searching and memory resident programs.
You
have been calling functions and passing actual values to them. When you call functions,
you pass actual arguments according to
the list provided in the declaration. These are all values, which are passed to a function, and this method is called call by value. In call by value, you can return only one value from a function and therefore, it puts restrictions on the usage of functions.
Global variables help by facilitating the indirect return of more than one value, but their excessive usage reduces understanding of the program and increases the chance of errors in coding.
This can be overcome by using call by reference, where any number of values can be indirectly returned
without taking the help of global variables.
Call by reference can be implemented by using pointers as the example below: Assume that you want to pass a and b to a function, divide a by b and return both the quotient and remainder to the main function. It is not possible to do this by call by value. Call by value can return only a single value, either the quotient or remainder, but not both. This can be achieved by call by reference as the following example demonstrates: /*Example 9.2 */ /* to demonstrate function call by reference*/ #include &gt;stdio.h&lt; main() { int a=100, b=13; void div(int *p, int *q);/*indicates call by reference*/ div(&a, &b);/*addresses of a and b are passed*/ printf("quotient= %d remainder= %d\n ", a, b); } /*function definition*/ void div(int *px, int *py) /*function declarator*/ { int temp1, temp2; temp1=*px; temp2=*py; *px=temp1/temp2; *py=temp1%temp2; } Result of
the program quotient= 7 remainder= 9 How does the program work?
Self-Instructional Material 201 Pointers NOTES In
the declaration part, we have declared div as a function passing two pointer variables and getting back void or none. We call div (&a, &b). We do not pass the values, but reference to the values. We actually pass the address of a and b to function div. The function div receives the reference, i.e., addresses of a and
b.
px = &a; py = &b; px points to a and py points to b. Hence, *px gets the value of a and *py gets the value of b. Now the contents of *px and *py, i.e., a and b are copied to temp1 and temp2. You divide temp1 by temp2 and place the result in variable *px whose address is known to both
the
main and div functions. In the main function, the address corresponds to a and in the
div
function it corresponds to *px. Therefore, the address of the quotient is returned to the main function indirectly. Similarly *py contains the remainder. It will be stored in b through the reference. Thus, the values of the quotient and remainder are stored in locations &a and &b, and we have indirectly returned two values to
the
main function through call by reference. The concept, though, may seem hazy at this point, will become clear as you see more examples. Some more points are to be noted carefully. The function declaration indicates that there is a function div, which returns nothing. It passes two pointers to integers. The pointers are declared as int*p and int*q. You will notice that they are not declared in the main function and therefore, p and q have no significance except to indicate that they are pointers to integers. They also indicate that the addresses of two integers are to be passed while calling the function. 9.7
POINTERS AND FUNCTIONS The function call using pointers is known as call by reference. Here the address of the variable is passed. Note that calling by reference has to be indicated in the function declaration such as these: fun (int *p, char *cp, float *fp, int *array); This is an indication that the function is to be called by reference for those parameters, which are pointers. A mixed declaration could also be used as given below: fun1 (int a, char *cp); Here
you
are indicating that an integer is passed by value and a character variable is passed by reference. In the above example, while you can either pass a character array (string) or a character through the second declaration, you can only pass one integer variable through the first parameter. The function declarator above the function body has to match the declaration. Hence, when fun1 is called, an integer followed by an address of character will be passed. In the function fun1, we may have a declarator as follows: fun1(int d, char * ch); Here the value of the integer variable will be automatically get assigned
to
d, and the ch will be assigned the address of the character variable. This means both ch and the address of the character variable in the calling function will point to the same location. Thus, both in the calling function and in the called function, the variable is accessible

202 Self-Instructional Material Pointers NOTES although under different names. Any modification made to the character variable either in the calling function or the called function affects both. While calling by reference, you have to pass the address of the variable. If the variable is declared by value such as int a, then you have to pass &a. If it was declared as a pointer to, say an integer, such as int * ip, then ip has to be passed. Return by Reference So far
you
have seen functions returning only a value, irrespective of whether they were called
by value or by reference.
Functions can also return reference or pointers. Whenever a function is to return a pointer, this has to be indicated by the following: 9.7.1 Function Declaration Declare the function as returning pointers. For example, int * fun1() ; char * fun2() ; float * fun3 (int a, float * b, char * c) ; The difference is the insertion of pointer (*) between the return data type and the function name. 9.7.2 Function Declarator This will be in the same format as the prototype or function declaration. For example, float * fun3 (int a, float * b, char * c); This declaration is
to match the third function declaration in above examples. 9.7.3 Function Call We may call, fun3 (x, &y, z); Here x is a variable and &y is an address. Although z looks like a value, it is in fact a reference to character, since prototype has the declaration char * c. 9.7.4 Return Statements The program will obviously return an address or pointer. 9.8 POINTERS AND ONE-DIMENSIONAL ARRAY
If you now declare,
int a [ 5 ]; ip = &a[0]; You have defined an array of integer a with 5 elements. When
you
assign the address of a[0], i.e., the 0th element of a to ip, ip will point to the array. The
system will automatically assign addresses for the other elements in the array by noting the data type of the array and the size occupied by each element.
Self-Instructional Material 203 Pointers NOTES 9.8.1 Finding the Greatest Number in an
Array The example below also explains the concept of a function returning a pointer. /* Example 9.3 to find the greatest number in an array*/ #include &gt;stdio.h&lt; main() { int array[]= {8, 45, 5, 131, 2}; int size=5, * max; int* fung(int *p1, int size);/*function returns pointer to int*/ max=fung(&array[0], size); /*max is a pointer*/ printf("%d\n", *max); } int * fung(
int *p2, int size) { int i, j, maxp=0; for (j=0; j&gt;size; j++) { if (*(p2+j) &lt; maxp) { maxp=*(p2+j); i=j; } }
return (p2+i); /*pointer returned*/ } Result of the
program 131 The called function returns the address of the greatest number in the array. Look at the function declaration. The function returning a pointer is indicated by the following (a star mark between the return data type and function name). int * fung(...) The address of array [0] is received by fung() and stored in *p2,
or
in other words, p2 points to the 0th location of the array. 9.9 POINTER NOTATIONS FOR ARRAYS At this point you must note another way of representing the elements of the array. The address of the 0th element is stored in at location p2 or address p2 + 0. The element with a subscript 1 of the array will be at
a location one above or at p2 + 1. Thus, the address of the nth
element of this array *p2 will be at address p2 + n. If you know the value of the pointer variable, i.e., the address of pointer, then it would be easy to express its value. The value of the integer stored at the
204 Self-Instructional Material Pointers NOTES nth location can be represented as * (p2 + n) just as the value at address (p2 + 0) is *(p2 + 0) or * p2. This notation is, therefore, quite handy. The if statement compares maxp with *(p2 + j) or p2 [j] or array [j]. You will easily understand the logic
as to how we get
the maximum or greatest number in * (p2 + j). At the end of the iterations, * (p2 + j), which is stored at location p2 + j, contains the maximum value in the array and therefore, we are returning an address or reference to the called function. In this example, (p2 + 3) is the address of the greatest number in the array. After return from the function, max gets the value of p2 + j. In the printf *max, which is the value stored in max, is printed. We have already defined max as a pointer to an integer in the main function. Thus, the function fung() returns the address of a value or a pointer, and by returning the address, the value is retrieved automatically by the main function. 9.9.1 Arrays and Pointers Examples above involved arrays and pointers.
The usage
of a pointer made the passing of an array to a function a simple task. You define array as an integer. However, in order to pass an address, the prototype was defined with a pointer argument int * p1. This means an address will be passed to the function while calling it. No distinction was made between a simple integer variable and an integer array. int * p1 can be a single valued integer or an array. This is possible because arrays are stored contiguously in
the

92% MATCHING BLOCK 62/126 W

memory. If the address of the 0th element is known, and the data type is known, it would not be difficult to calculate the address of any element in the array. Therefore, it is enough if the address of the 0th element is passed to a function. *p1 refers to the address of a variable or to the 0th element of an array. Note the flexibility of arrays in 'C' and how intelligently the language uses pointers. While calling a function, the address has to be passed and this is achieved in the above example by passing & array[0]. In the called program, *p2 is treated as an array without any additional efforts. By adding the index to p2, you get the address of the various elements in the array. Getting value is achieved by placing * before the address. Now, look at another example using arrays and pointers as follows: /*Example 9.4 passing an array of integers to function - method2*/ #include&gt;stdio.h&lt; main() { int array[]= {10, 20, 30, 40, 50}; int *a; void pass(int *a, int k); a=&array[0]; pass(a, 4); } void pass(int *b, int j) { int k=0; while (k &gt;= j) { Self-Instructional Material 205 Pointers NOTES (*b)=(*b)/2; printf("value %d @ address %d\n", *b, b); k++; b++; } } Result of

the
program value 5 @ address 8694 value 10 @ address 8696 value 15 @ address 8698 value 20 @ address 8700 value 25 @ address 8702 Here, * a has been declared as a pointer to integer. The variable a has been assigned the address of the 0th element of the array. Now the function call is made using pass (a, 4); again a is the address of the variable array [0]. In the function pass, b received the address of a, and the next element of the array is accessed each time by incrementing the address b. Note that the values of elements in the array are divided by 2 in the called function. As you become familiar with pointers you can write programs very easily. 9.10 MULTIDIMENSIONAL ARRAYS We may represent a two-dimensional array conventionally as a[i][j]. The elements of a two-dimensional array of size 3 × 3 can be represented
symbolically as
given below: a00 a01 a02 a10 a11 a12 a20 a21 a22 We can then visualize this as an array of arrays. Do not get puzzled. Each row is an array, and we have three such arrays. Thus a two-dimensional array can be considered as an array of one-dimensional arrays. Therefore, we can expect the following type of memory allocation by the system: 0 1 2 0 1 2 0 1 2 a[0][0] a[1][0] a[2][0] This means all the elements of the 0th row are stored contiguously. a[0][0] is stored first, followed by all the elements of the 0th row. Next, the first row starts with the first element a[1][0]. At the end of the first row, the second row starts with a[2][0]. Try to visualize how the elements of a two-dimensional array are stored contiguously. You know that, a[i] = * (a + i) Where a + i is the address of the element. You also know that, a [0] = * a and its address is stored in location a.
206
Self-Instructional Material Pointers NOTES
You can now transform a two-dimensional array into this convenient form. You can visualize **b[0], **b[1] and **b[2] to represent the values of row first elements, which are stored at locations *b[0], *b[1] and *b[2] as shown: 0 1 2 0 1 2 0 1 2 Address *b[0] *b[1] *b[2] ** indicates pointer to pointer. This is acceptable since the row pointers in turn point to
the
column pointers. If you assume the row first elements as pointers, then the addresses of elements of the two-dimensional arrays can be addressed by simple arithmetic. *b points to the value in the one-dimensional array. Then **b refers to the value in the two-dimensional array. All elements of the two-dimensional array are stored contiguously. From the storage pattern we declare the following: **b refers to the value at the 0th row of a two-dimensional array. Naturally the address of the 0th row will be *b. **(b + 1) refers to the value of the 0th element in the 1st row. Its address will be *(b + 1). In general, the 0th element in ith row contains value ** (b + i) and its address is *(b + i). You have only talked about the 0th element in each row, i.e., where the row starts conceptually. How
do
you

79% MATCHING BLOCK 63/126 W

address the other elements? You know that an element b[i][j] is the jth element in the ith row. You know the address of the 0th element. It is * (b + i). Therefore, the address of the jth element in ith row will be (* (b + i) + j). Note j is the offset. Therefore, the value at this location can be expressed as * (* (b + i) + j). You have just added a star outside the address. Now the address of the 2nd element in the 2nd row will be, (*(b + 2) + 2) Its value will be, *(* (b + 2) + 2) What will be the address of the 0th element in the second row, (*(b + 2)) What will the value be? *(* (b + 2)) Note the parentheses and *. To understand this clearly, execute the following program: /*Example 9.5- to understand pointers to two-dimensional array*/ #include&gt;stdio.h&lt; main() { int i,j; int a[3][3]; printf("Enter the values of 3x3 matrix\n"); for (i=0; i&gt;=2; i++) for(j=0; j&gt;=2; j++) Self-Instructional Material 207 Pointers NOTES scanf("%d", (*(a+i)+j)); for (i=0; i&gt;=2; i++) { for (j=0; j&gt;=2; j++) { printf("address=%d\n", (*(a+i)+j)); printf("value=%d\n", *(*(a+i)+j)); } } } Result of the program Enter the values of 3x3 matrix 00 01 02 10 11 12 20 21 22 address=9098 value=0 address=9100 value=1 address=9102 value=2 address=9104 value=10 address=9106 value=11 address=9108 value=12 address=9110 value=20 address=9112 value=21 address=9114 value=22 This example helps you to understand how a two-dimensional array is stored in the memory. The array has been declared in the normal way without pointers, but the array elements have been received, stored and printed using pointer notation. When

you

execute the above program, you can type the numbers in any one of the ways given below: • Enter one integer at a time followed by Return. • Enter all integers in a row with spaces between them and finally hit Return. • Enter three integers in a row as the result of the program indicates. Since it is a two-dimensional array you would like to input the value in the form of rows and columns and get the output in the same form. It would be better if the computer accepts the inputs at the given places and prompts us to do so as in the program given below:
208 Self-Instructional Material Pointers NOTES 9.10.1

**94%** **MATCHING BLOCK 64/126** **W**

Receiving Inputs at Chosen Points /*Example 9.6- to accept and print in matrix form*/ #include &gt;stdio.h&lt; #include &gt;conio.h&lt; main() { int a[3][2]; int i,j; printf("Enter values of 3x2 matrix-\n"); printf("Press Enter after each value"); for (i=0; i&gt;=2; i++) { for (j=0; j&gt;=1; j++) { gotoxy (j*10+10,4+i*2); scanf("%d",(*(a+i)+j)); } } for (i=0; i&gt;=2; i++) { for (j=0; j&gt;=1; j++) { gotoxy (j*10+40, 4+i*2); printf("%d", *(*(a+i)+j)); } } } Result of

the
program Enter values of 3x2 matrix Press Enter after each value 00 01 0 1 10 11 10 11 20 21 20 21 A library function gotoxy(x, y) has been used in the program. This function makes the cursor go to the point (x, y) on the screen. At the first iteration when both j and i are 0, you will call goto (10, 4), while scanning the values and call goto (40, 4), while printing the values. Therefore, the first value will be scanned on the 4th line from the top and at the 10th character position. The first value will be printed at the same line and the 40th character position as the Result of the Program indicates. Therefore, you can direct the cursor to any position you like. Thus, gotoxy() will be quite useful in advanced programming.
Self-Instructional Material 209 Pointers NOTES
This has been used both for scanf() and printf(). The system will scan inputs from those points and print outputs at the points specified. You can specify any point on the screen to take inputs and print outputs. When
you
execute the program the cursor will go to the specified position. After you have typed one value and entered the return key, the cursor will blink at the next point, giving the impression of inputting a matrix. After the values are given, the output also appears in the form of a matrix. The left pair of numbers is what was entered. Since the same vertical position, but a different horizontal position, has been specified in the program, the printout appears on the same lines, but to the right of the input values. Try the program and see for yourself. Look at the address specification along with the scanf(). It is, (*(a + i) + j), which is equivalent to &a [i][j]. Putting a star before this converts this to the value a [i][j]. This program clearly illustrates the way to handle a two-dimensional array with pointers. Although the array could have been declared as **a, it has been declared with the dimensions of the array. Since it does not know exactly the total number of elements in the former definition, garbage values may be stored in some more locations adjacent to the array elements, which can create problems. Some compilers will cause a run-time error, if the array has been declared as **a. Therefore, it would be better to specify the dimension of the array as given in the above two examples. 9.11
POINTERS AND STRINGS It is amply clear that a string is an array of characters. Since, you have seen an array of numbers, you may want to represent a string in the same manner. You can initialize a string as given below: str [] = {'c', 'h', 'a', 'r'}; This is absolutely correct. At the end of every string a
NULL character '\0'
is automatically inserted. However, strings can be written very easily as given below: *str = "char";. The compiler accepts it when it is a string. str[0] is the address of the 0th location of str. Since, the character array is stored contiguously, there is no need to know the address of each element. A program to find
the length of a string
is as follows: /*
Example 9.7
to find the length of a string*/ #include &gt;stdio.h&lt;
main() { int wlength; char *wp ="shri durgaya namaha"; int wlen(char *wp); wlength=wlen(wp); printf("length of the word=%d",wlength); }
210
Self-Instructional Material Pointers NOTES /*
function to find length*/ int wlen(char *w) { int n; for (n=0; *w!='\0'; w++) n++; return n; } Result of the program length of the word=19 You have a library function strlen() to find the length of the string. This program carries out the same task although strlen() could be used to find the length of any string. The program serves to illustrate the concept of strings and pointers easily. It prints the length of the string in terms of the number of characters. The white spaces in between will also be counted. You have initialized *wp while declaring it as a pointer to a character. The compiler will not normally accept such initialization, but makes an exception for strings, since there is no anomaly as to whether the initialization is for the value or the address, as a string cannot be an address, unlike an integer. Note the statements in the function. The address as well as n are incremented when the *w is not NULL. NULL indicates the end of string since NULL will be appended to all the strings. When w points to NULL, there will be no further increment of w as well as n. Therefore n indicates the length of the string wp; n is returned to the main function. You will notice that just by incrementing the address,
you

can scan the string. The same program to find length of string is modified and shown below: /*Example 9.8- alternate method to find the length of a string*/ #include &gt;stdio.h&lt; main() { int wlength; char *wp="shri durgaya namaha"; int wlen(char *wp); wlength=wlen(wp); printf("length of the word=%d",wlength); } /*function to find length*/ int wlen(char *w) { char *p = w; while (*p!='\0') p++; return p-w; } Self-Instructional Material 211 Pointers NOTES You get the same result. Here, the address w is assigned to p through the assignment and declaration statement in the called function. Hence p also points to the same string, i.e., p points to the first character in the word. When p points to the first character or 0th element, p is incremented to the 1st location. When p points to the (n – 1)th element, p is incremented to

the nth location. Here, only p is incremented and
not w. It continues to point to the 0th location. Therefore, at the time of termination of while loop, p will point to the location corresponding to NULL. Thus, p – w, i.e., the difference in the addresses that is equal to the number of characters in wp, is returned to the main function and printed there. 9.11.1
String Functions
There are a number of library functions for string manipulation as given below: strlen (CS)—Returns the length of string CS.
char * strcpy (s, ct)—Copy string ct to string s, including
NULL and return s. char *
strcat (s, ct)—Concatenate string ct to end of string
s;
return s. int strcmp (cs, ct)—Compare string cs to string ct; return &gt; 0 if cs &gt; ct; 0 if cs = = ct or &lt; 0 if cs &lt; ct.
char * strchr (cs, c)—Returns the pointer to the
first occurrence of c in cs or NULL if not present.
There are some more string functions. If these are to be used &gt;string.h&lt; should be included before the main function. 9.11.2
To print a Substring Now, write a function to print a substring of
a
given length and starting position. This is illustrated in Example 9.9. /* Example 9.9 gets substring beginning with specified character position */ #include &gt;stdio.h&lt; #include &gt;string.h&lt; void substring(char *str,char *substr,int len);
void
main() { char text[80],substr[20]; int len,pos; printf("Enter any Text :"); gets(text); printf("Enter the Length of Substring Required :"); scanf("%d",&len); printf("Enter the Position From which Required :"); scanf("%d",&pos); substring(text+(pos-1),substr,len);
212
Self-Instructional Material Pointers NOTES
printf("Substring Is %s \n",substr); } void substring(char *str,char *substr,int len) { int cnt=0; while(*str &&cnt &gt;len) { *(substr++)=*(str++); cnt++; } *(
substr)=0; } Result of the
program Enter any Text: This is a program to get a substring Enter the Length of Substring Required: 7 Enter the Position From which Required: 11 Substring Is program Try to understand how the program works. The following statement, which is a call to the function, needs explanation: Substring (text + (pos–1),substr,len); Usually
you
count the first character of a word as the 1st position, whereas 'C' will recognize it as the 0th position. text points to the address of the 0th position of the text. Therefore, text + (pos–1) points to the address of the character in text from which the substring starts. Although substr is empty, you are passing the address of the 0th location of the substring as the next argument. You will store the substring in substr, which is empty when it is passed to the called function. The third argument is the length of the substring. Now look at the called function. In the while loop, you carry out the following operations: • Copy (str) to substr, i.e., from the starting position indicated by the user, copy one character of text to substring. • Increment str. • Increment substr. • Increment local counter cnt. Copying will terminate either on finding no characters in str or when the required number of characters have been copied. Thus the program works correctly. Note the following features: Since the strings are passed by reference, there is no need to return the addresses or values, as the addresses of strings are known to the main function.
There are a set of library functions for testing characters. The prototype for these functions is defined in &gt;ctype.h&lt;. The argument for all functions must be an unsigned
Self-Instructional Material 213 Pointers NOTES character. The functions return an integer. Some of the character test functions are given below: isalpha (c)– Is c an alphabet, i.e., upper case or lower case character. islower (c)– Is c a lower case. isupper (c)– Is c an upper case. isdigit (c)– Is c a decimal digit. isspace (c)– Is c a space, form feed, new line, carriage return, tabs. ispunct (c)– Is c a printing character except space or letter or digit. isprint (c)– Printing character including space. isalnum (c)– Is alpha (c) or digit (c) true. Thus

the functions return non-zero (true) if the argument c satisfies the condition described and zero if

it is not satisfied. 9.11.3 To analyse a Text File Now write a program to count the number of vowels, consonants, digits and punctuation marks. The program is as follows: /* Example 9.10 - for counting the number of vowels, consonants, digits and other characters */ #include &gt;stdio.h&lt; #include &gt;string.h&lt; #include &gt;ctype.h&lt; main() { int ch,i,len1,vc,cc,dc,wc, oc; char text[256]; int isvowel(int ch); int iscons(int ch); printf("Enter any Text :\n"); gets(text); len1=strlen(text); vc=cc=dc=wc=oc=0; for(i=0;i&gt;len1;i++) { ch=text[i]; if(isdigit(ch)) dc++; else if(isspace(ch)) wc++; else if(isvowel(ch)) vc++;
214 Self-Instructional Material Pointers NOTES else if(iscons(ch)) cc++; else oc++; }
printf("\nNo. of Vowels :%d\n",vc);

**57%** **MATCHING BLOCK 68/126** **W**

printf("No. of Consonants :%d\n",cc); printf("No. of Digits :%d\n",dc); printf("No. of White Spaces :%d\n",

wc); printf("No. of Other Chars. :%d\n",
oc); } int isvowel(int ch) { char vows[]="aeiouAEIOU"; char *ret; ret=strchr(vows,ch); if(ret==NULL) return(0); return(1); } int iscons(int ch) { char vows[]="aeiouAEIOU"; char *ret; if(isalpha(ch)) { ret=strchr(vows,ch); if(ret==NULL) return(1); } return(0); } Two functions isvowel() and iscons() are called. Their functioning is explained below. To find a Vowel vows array contains the five vowels in the upper case and in the lower case. Recall that,
char * strchr (cs, c) returns the pointer to the
first occurrence of c in cs or NULL, if not present.
Therefore, isvowel calls strchr with arguments (vows, ch). The ch is checked to find whether it is in vows array. It returns a pointer corresponding to the address of the character in the array. Therefore, if ch is a vowel, ret will not be NULL and hence, true will be returned to main. If it is not a vowel, then strchr will return NULL and 0 (false) will be returned to main.
Self-Instructional Material 215 Pointers NOTES To Find Consonants Alphabets other than vowels are called consonants and iscons() checks for consonants in the following manner. First ch is checked to determine whether it is an alphabet. If not, 0 is returned. If it is an alphabet, it checks whether it is a vowel. If the alphabet is a vowel, then ret is not equal to NULL and will be true. Therefore, the if condition is false. Hence, 0 is returned. If not, then it is a consonant, and strchr would have returned NULL. Hence, 1 is returned by functions iscons(). The main() recursively gets one character at a time. It checks the character in the following order: Is it a digit? If so, add 1 to dc. If not Is it a space, if so add 1 to wc. Else, if it is a vowel, add 1 to vc. Else, if it is a consonant, add 1 to cc. Otherwise, add 1 to oc. Thus, the program finds out the various types of characters and maintains a count. Result of the program Enter any Text: This program checks the type of alphanumeric characters with numbers like 4, 8, 12, etc. and other characters No. of Vowels: 27 No. of Consonants: 57 No. of Digits: 4 No. of White Spaces: 17 No. of Other Chars: 3 9.12
ARRAY OF POINTERS You had discussed that a pointer to a one-dimensional array can be denoted as *b and two-dimensional array can be denoted as **b. In this section, an array of pointers will be discussed. 9.12.1 Sorting Character Strings As you know, names of students in a class can be denoted by a two-dimensional array like b[50][20], with the second subscript denoting the width of the names and the first 50 denoting the number of students. There may be insufficient space for names longer than 20 characters, leading to truncation. If the name is short, it will lead to
the
wastage of space. By using an array of pointers, we can declare char * name [50] as an array to store the names of 50 students. The number of students in a class is definitely known. Here name is an array and points to character. Thus, it is an array of pointers. Actually what happens is that 50 addresses are stored in the array name [50], name [0] corresponds to the address of
the first name, name [1] to the
216 Self-Instructional Material Pointers NOTES address of the

**95%** **MATCHING BLOCK 67/126** **W**

second name and so on. Using this concept a program is developed to sort names. It is given below: /* Example 9.11 - for sorting character strings */ #include &gt;stdio.h&lt; #include &gt;string.h&lt; #include &gt;alloc.h&lt; #define N 5 /*5 names sorted*/ main() { int ret,i,j,p=0; char *name[N], t[50]; for(i=0;i&gt;N;i++) { printf("Enter name size\n"); scanf("%d", &p); name[i]=(char *)(malloc(p*1)); printf("Enter name\n"); scanf("%s",name[i]); } for(i=0;i&gt;N-1;i++) for(j=i+1;j&gt;N;j++) { ret=strcmp(name[j],name[i]); if (ret&gt;0) { strcpy(t,name[i]); strcpy(name[i],name[j]); strcpy(name[j],t); } } printf("\nSorted Names Are :\n"); for(i=0;i&gt;N;i++) printf("%s\n",name[i]); } Result of

the
program Enter name size 5 Enter name Raman Enter name size 5 Enter name Gopal Enter name size 6
Self-Instructional Material 217 Pointers NOTES Enter name Sharma Enter name size 7 Enter name Ramanan Enter name size 4 Enter name Basu Sorted Names Are: Basu Gopal Raman Ramanan Sharma Here the user is asked to first enter the number of characters in a name to be entered. After that the name is entered. This helps in allocating the right size to each name, no space more
or less. You
call strcmp and pass references to names being compared. The function strcmp returns 0 if strings are equal; &gt;0 if string 1 is less than string 2 in
the

ASCII value; &lt; 0 if string 1 is greater than string 2. Thus, names will be exchanged if *name[j] is less than *name[i]. The string comparison starts from the first character of each word. If they are equal, it will go to the next character and so on till they are unequal or NULL is reached in either of them. Then the difference in the ASCII values of the characters compared last is returned. This is what you wanted in arranging the names alphabetically. Key in the program and you will find that it works correctly. 9.13
DYNAMIC ALLOCATION OF MEMORY The dimension or array size declaration of an array is an important subject. The array dimension is to be declared before compiling. For example, some valid declarations are: char name[25]; int mark[40]; You have not come across any problem
since you were initializing the arrays and hence, the array size was known. If you were to get the array elements at run-time, sometimes you could give either a lesser number of elements or more elements. In the former case, garbage values will be stored in the empty spaces in the array misleading the user. In the latter case, the elements will be lost. To avoid this problem, you may think that you can specify marks [n] and give the value of n later at run-time. However, this will not work, and the compiler will force you to give the actual dimension. Hence, dynamic memory allocation is useful. malloc and calloc serve to specify the actual dimension at run-time and hence, enable memory allocation dynamically. Next time when you execute the program, you can specify any value for n. It will definitely work.
Check Your Progress 10. What values are returned by call by value? 11.
Define call by reference. 12. What does ** indicate? 13. What will the function go to xy (x, y) do? 14.
Why is strlen() function used?
218 Self-Instructional Material Pointers NOTES
The functions malloc() and calloc() allot memory dynamically. The specifications are given below in the following statement. void * malloc (n*size n ); It returns the pointer to n bytes of the memory or NULL if allotment is not possible. Since it returns a pointer you have to specify the array as a pointer variable as given below: int *b ; / * array declared * / b = (int *) malloc (x * 2) ; / * x is the array size * / Similarly the calloc is defined as void * calloc (sizen, size_size). Here the number of arguments is two. The first one is the array size and the second one is the bytes occupied per element, i.e., the storage space required for the data type. The function returns a pointer allocating space for an array of size sizen, of data type size size, but the array contents will be initialized to zero. In malloc, the initial contents will be garbage values. You can use it as follows: int * b; b = (int *) calloc (x, 4); Here 4 indicates the array is of type float and space is allocated to store x floats. When you use calloc or malloc, you must include alloc.h. A program to demonstrate malloc and calloc is as follows: /* Program 9.12 to do string concatenation by using dynamic allocation of memory*/ #
include &gt;stdio.h&lt; #include &gt;alloc.h&lt; #include &gt;string.h&lt; main() { char *
newstrcat(char *dest, char *src); char *name1, *name2; int n1, n2; printf("Enter size of 2 names:\n"); scanf("%d%d", &n1, &n2); name1 = (char *) calloc(n1, 1); name2 = (char *) calloc(n2, 1); printf("Enter 2 names\n"); scanf("%s%s", name1, name2); printf("New name:%s\n",newstrcat(name1,name2)); } char *newstrcat(char *dest, char *src) { char *w; int i,len,len1,cnt=0; len=strlen(dest); len1=strlen(src); w=(char *)malloc(len+len1+1);
Self-Instructional Material 219 Pointers NOTES
for(i=0;i&gt;len;i++) w[cnt++]=dest[i]; for(i=0;i&gt;len1;i++) w[cnt++]=src[i]; w[cnt]=0; return(w); } Result of the program Enter size of 2 names: 4 4 Enter 2 names Rama samy New name:Ramasamy calloc is used to allot memory to name1 and name2. The function newstrcat is called while printing. In the called function, malloc is used to allot space equal to the size of name1, name2 +1. The additional space is for placing NULL. The string is concatenated by copying one character at a time using two
for statements.
Finally, the concatenated string is returned to the main function where it is printed. The memory allocated using malloc, calloc can also be freed, when it is no longer necessary by using the function free ( ). For example, in the main ( ) of the above example, after the last statement you can add the following statements: free(name1); free(name2); These statements will deallocate the memory space allocated to them after the job of printing the concatenated string is over. Thus, dynamic allocation of the memory according to the exact need and freeing it after use is quite useful for conserving the memory. This concept is used by professional programmers when they develop commercial software products. Memory saved is more than money saved in such product development. 9.14
POINTER COMPARISON The addresses or pointers can be stepped up or stepped down. For example, float *fp; float f; fp = &f; fp++; If the original address of fp was 1000, fp++ will take the pointer to 1004, since it is a float pointer. What happens when fp = fp + 2; is executed? It will skip 2 locations or 8 locations. You can verify this
for
yourself. Similarly, fp = fp – 4 is also valid. Two pointers of the same type can be compared also. Assuming that you declare float * fp1. 220
Self-Instructional Material Pointers NOTES Then you can compare their relationship with the
if statements given below: if (fp == fp1) ... else if (fp &gt; fp1) ... Thus, when
you say you are comparing pointers, you are comparing their addresses. 9.15 STRUCTURE POINTERS
You know how to declare pointers to various data types and arrays. Similarly, pointers to structures can also be declared. For instance, in the case of the structure account, you can declare a pointer as shown: struct account a1 = { 1, "Vasu", 1000 }; struct account * sp; struct sp = &a1 ; Now sp is a pointer to a structure. Therefore, if you assume structure as another basic
data type,
declaring it as an array and declaring it as a pointer, etc. follow the same rules. Structure is in fact a user-defined data type. Access to individual elements of a structure defined in the form of a pointer is similar, but instead of dot we use an arrow pointer - &lt;. Arrow pointer is formed by typing minus followed by the greater sign. However, on to
the left of the arrow
operator there must be a pointer to a structure. The

following
example will clarify the point: /*Example 9.13- structure pointers */ #include&gt;stdio.h&lt; main() { struct account { unsigned number; char name[15]; int balance; }a5; static struct account a1= {001, "VASU", 1000}; struct account *sp; sp=&a1; printf("A/c No:=%u\t Name:=%s\t Balance:=%d\n", sp-&lt;number, sp-&lt;name, sp-&lt;balance); a5=*sp; printf("A/c No:=%u\t Name:=%s\t Balance:=%d\n", a5.number, a5.name, a5.balance); } Result of the program A/c No:=1 Name:=VASU Balance:=1000 A/c No:=1 Name:=VASU Balance:=1000 In this program, sp is a pointer to structure account, and therefore sp is assigned the address of structure a1. Then the contents of structure *sp are printed.
Self-Instructional Material 221 Pointers NOTES The elements of *sp are copied to a5 and then printed (to demonstrate copying of structures). Note the difference between the notations when accessing elements of a structure and a structure pointer. 9.16 SUMMARY In this unit you have learned about pointers. The pointer
is an integer variable, which contains the address of a variable.
It can point to any data type. Pointer arithmetic was explained with a number of examples to make the concept of pointers clear and unambiguous. Then pointers and functions were discussed. Pointers allow call by reference, permitting return of more than one variable indirectly. Functions can also return pointers in addition to value. The steps needed to define functions returning reference explicitly were discussed with an example. Array manipulation becomes easier with pointers. This was explained both for array of numbers and later for array of characters, i.e., strings. Ways of handling two-dimensional arrays easily with pointers were also discussed.
Static allocation of memory in case of arrays through dimension leads either to excessive allocation or short allocation. This can be overcome by dynamically
allocating memory at run-time through the functions namely malloc and calloc.
This concept is useful for commercial product design. Using an array of pointers is preferable to pointer to pointer in terms of ease of use and memory allocation. Pointers to functions, were also discussed. 9.17 KEY TERMS • Pointer: It is a powerful concept of C and helps in achieving results using an indirect method for returning more than one value from a function.

| 100% | MATCHING BLOCK 73/126 | SA | Sem I_BCA_B21CA02DC.docx (D165443993) |
|---|---|---|---|

It is a variable that contains the address of another variable

and is closely associated with memory addressing. • Memory locations: These
are available in groups of 8 bits or a byte which has an address and stores a value that can be any data type, such as float or char or int. The memory locations are arranged in an increasing order of addresses starting from 0000, which increases one by one. •
Null pointer: It is a type of pointer of any data type and generally takes a value as zero. It can also take any pointer type, but do not point to any valid reference or memory address. It is different from a pointer that is not initialized. • gotoxy(x,y): It is a library function and is used in advanced programming to direct the cursor to go to the point (x,y) on the screen. •
strlen(): It is a library function and is used to find the length of the string.
It prints the length of the string in terms of the number of characters. The white spaces in between will also be counted. •
isvowel(): This function is used to check whether the array contains vowels. vows array contains the five vowels in the upper case and in the lower case. • iscons(): This function in the program checks the consonants in the array string. • vows array: It contains the five vowels in the upper case and in the lower case.
Check Your Progress 15. How is a memory allocated dynamically? 16. Write the specifications for malloc() and calloc() memory allocation. 17.
Which two pointers can be compared?
222 Self-Instructional Material Pointers NOTES 9.18 ANSWERS TO 'CHECK YOUR PROGRESS' 1. The pointer is a powerful concept of C. It is powerful because of the following two reasons: • It helps in achieving results, which could not otherwise be achieved, such as an indirect method for returning more than one value from a function. • It results in a compact code. Pointers are closely associated with memory addressing. 2.
Memory locations are available in groups of 8 bits or a byte. Each byte in memory has an address. Therefore, each location has an address and stores a value. The value stored can correspond to any data type, such as float or char or int, and their type modifiers. 3.
A pointer is a variable that contains the address of another variable. 4.
A pointer
to
void can be declared in the same way we declare any other pointer such as pointer to integer, float, etc. void* void_pointer; 5. void is a keyword and it means 'nothing'. Note also that void pointer also points to a memory address and hence, it is also an integer like any other pointer. The void pointer points to nothing. 6.
NULL pointer is a type of pointer of any data type and generally takes a value as zero.
This denotes that a NULL pointer does not point to any valid memory address. 7. A void pointer is a special type of pointer of void and denotes that it can point to any data type. NULL pointers
can take any pointer type, but do not point to any valid reference or memory address. It is
important to note that a NULL pointer is different from a pointer that is not initialized. 8.
The postfix of the increment operator enables increment of the variable after printing. 9.
Decrementing takes place before printing because it is a prefix operator. 10.
In call by value, we can return only one value from a function and therefore, it puts restrictions on the usage of functions. 11.

The function call using pointers is known as call by reference. Here the address of the variable is passed. Note that calling by reference has to be indicated in the function declaration such as these: fun (int *p, char *cp, float *fp, int *array); 12. ** indicates pointer to pointer. This is acceptable since the row pointers in turn point to the column pointers. 13. This function makes the cursor go to the point (x, y) on the screen. 14.
Library function strlen() could be used to find the length of any string. 15. malloc and calloc serves to specify the actual dimension at run-time and hence, enable memory allocation dynamically. 16. The functions malloc() and calloc() allot memory dynamically. The specifications are given below in the following statement. void * malloc (n*size n ) 17.
Two pointers of the same type can be compared also.
Self-Instructional Material 223 Pointers NOTES 9.19
QUESTIONS AND EXERCISES Short-Answer Questions 1. State whether True or False and give reasons: (a) In call by value, the values of arguments are actually passed. (b) The pointer contains the address of a variable. (c) Float pointer is of type float. (d) *var = 100; assigns 100 to *var. (e) char *fun(); is not a valid declaration. (f) A pointer can be assigned to another pointer. (g) The variables declared in the function declaration in the calling function can be used in it later. (h) * (p2 + 0) is valid. (i) malloc stores zeros in the allotted memory locations initially. (j) Pointers to functions denote the address of function. 2. What is printed in each of the print statements given below, if they are executed one after another? After evaluation, confirm your answers
by trying
it in a program. Declaration: * np = 10; printf() statement arguments: (a) np (b) *np (c) np (d) np++ (e) np = np – 2 (f) *np ++ (g) np++ (h) *np++ (i) ++np (j) *np Long-Answer Questions 1. Describe the following with the help of examples: (a) Pointer arithmetic. (b) Pointers and two-dimensional arrays. (c) Advantage and operation of: (i) malloc (ii) calloc (d) String functions. (e) Call by reference and similarity with global variables. 2. Write programs for the following: (a) A function nstrcmp to compare
string cs to string ct; return &gt;0 if cs &gt;ct, 0 if cs = = c or &lt;0 if cs &lt; ct.
224
Self-Instructional Material Pointers NOTES (
b) A function to check whether characters +, –, *, / present in a string. (c) To find the kth smallest element in an array using pointers. (d) To check whether a string is palindrome. (e) To reverse a number using pointers. 9.20
FURTHER READING Gottfried, Byron S. Programming with C, 2
nd edition. Schaum's Outline Series. New York: McGraw-Hill, 1996.
Forouzan, Behrouz A. and Richard F. Gilberg. Computer Science: A Structured Programming Approach Using C, 2nd edition. California: Thomson, Brooks Cole, 2000. Dey, Pradip and Manas Ghosh. Computer Fundamentals and Programming in C. New Delhi: Oxford Higher Education, 2006. Bronson, Gary J. A First Book of ANSI C, 3rd edition. California: Thomson, Brooks Cole, 2000. Kanetkar, Yashwant. Understanding Pointers in C.
New Delhi: BPB Publication, 2001. Kanetkar, Yashwant. Let us C. New Delhi: BPB Publication, 1999.
Kernighan, Brian W. and Dennis Ritchie. C Programming Language, 2nd edition. New Jersey: Prentice Hall, 1988. Foster, W. D. and L. S. Foster. C by Discovery. Boston: Addison-Wesley, 2005. Kanetkar, Yashwant. Working with C. New Delhi: BPB Publication, 2003. Horton, Ivor. Instant C Programming. New Jersey: Wrox Press (John Willey & Sons), 1995. Lawlor, Steven C. The Art of Programming Computer Science with 'C'. New Jersey: West Publishing Company, 1996.
Self-Instructional Material 225 Structures and Union NOTES UNIT 10 STRUCTURES AND UNION Structure 10.0 Introduction 10.1 Unit Objectives 10.2 Structures 10.3 Defining and processing a Structure 10.4 User-Defined Data Types 10.4.1 Array of Structures 10.5 Structures and Pointers 10.6 Passing Structures to Functions 10.6.1 Structures to Functions 10.7 Self-Referential Structures 10.8 Union 10.9 Summary 10.10 Key Terms 10.11
Answers to 'Check Your Progress' 10.12 Questions and Exercises 10.13 Further Reading 10.0 INTRODUCTION In this unit, you will learn about structures and union.
A
structure
is a user-defined data type like an array. While an array contains elements of the same data type, a structure
contains members of varying data types.
Thus, it is useful to represent real-life examples. The
structure declaration ends with a semicolon and the keyword is struct. The tag for structure is optional.
The structure elements can be referenced or copied individually through the dot operator. A structure can also be straightaway assigned to another structure. An array of structures is a replication of structures like an array of any other variable(s) and can be passed to a function either entirely or element-wise. Pointers to structures are similar to other pointers.
Structures can be passed by reference, and can be nested.
A structure is useful in database management and graphics, and is the forerunner for classes in C++. Thus, structures are clearly defined for manipulating complex data elements. In this unit, you will learn about unions. They
help in conserving memory, as only one of the many variables will be used at a time. The syntax of a union is similar to
a structure. 10.1 UNIT
OBJECTIVES After going through this unit, you will be able to: • Understand the
basic concept of
structure • Declare a structure • Process a structure •
Understand user-defined data types • Explain structures and its relation to pointers • Pass structure to function

226 Self-Instructional Material Structures and Union NOTES • Understand self-referential structures • Understand and use union 10.2 STRUCTURES Arrays and structures have similarities as well as differences between them. Both arrays and structures represent collections of a number of items. For example, int x [100]; defines an array x with dimension 100, i.e., 100 items, all of the same data type, namely integers. However, structures can represent items of varying data types pertaining to an item. The only similarity between an array and a structure lies in the fact that there can be a collection of structures which is known as an array of structures. For example, a book has the following details to describe it fully: title name of the author name of the publisher price year of publication Basically, all these details of a book can be considered to be a record or structure and the five items above are fields of the record. This is the convention in databases and in other languages. There can be details of 100 books which can be represented as an array of structure book. 10.3 DEFINING AND PROCESSING A STRUCTURE Structure
is synonymous with records. Structure, similar to a record, contains a number of fields or variables. The variables can be of any of the valid data types.
The
definition of the record book, which is called a structure is as follows: struct book { char title [25]; char author [15]; char publisher [25]; float price ; unsigned year; }; struct is a keyword or a reserved word of 'C'.
A structure
tag or name follows which is book in this case. This is not a must, but giving a tag to structure will improve the reader's understanding. The beginning of the structure is indicated by opening brace. Thereafter, the fields of the record or data elements are declared one by one. The variables or fields declared are also called members of the structure.
The structure consists of different types of data elements which is different from array. Let us now look at the members of struct book. The title of the book is declared as a string with width 25; similarly the author and publisher are arrays of characters or strings of the specified width. The price is defined as a float to take care of the fractional part of the currency. The year is defined as an unsigned integer.
Self-Instructional Material 227 Structures and Union NOTES

**94%** **MATCHING BLOCK 69/126** W

Note carefully the appearance of a semicolon after the closing brace. This is unlike other blocks. The semicolon indicates the end of the declaration of the structure. The following are the rules to declare a structure. (a) struct is the header of a structure definition. (b) It can be followed by an optional name for the structure. (c) Then the members of the structure are declared one by one within a block. (d) The block starts with an opening brace, but ends with a closing brace followed by a semicolon. (e) The members can be of any data type. (f) The members have a relation with the structure; i.e., all of them belong to the defined structure and they have identity only as members of the structure and not otherwise. Therefore if you assign a name to author, it will not be accepted. You can only assign values to book.author. The structure declaration above is similar to the prototype in a function in so far as memory allocation is considered. The system does not allocate memory as soon as it finds structure declaration, which is for information and checking consistency later on. The allocation of memory takes place only when structure variables are declared. What is a structure variable? It is similar to other variables. For instance int I means that I is an integer variable. Similarly the following is a structure variable declaration. struct book s1; Here s1 is a variable of type structure book. Suppose,

we
define, struct book s1, s2; This means that there are two variables s1 and s2 of type struct book. These variables can hold different values for their members. Another point to be noted is that the structure declaration appears above all other declarations. An example which does nothing, but defines structure and declares structure variables is as follows: main ( ) { struct book { char title [25]; char author [15]; char publisher [25]; float price; unsigned year; }; struct book s1, s2, s3; } If you want to define a large number of books, then how will you modify the structure variable declaration? It will be as follows: struct book s[1000]; This will allocate space for storing 1000 structures or records of books. However, how much storage space will be allocated for each element of the array? It will be the
228
Self-Instructional Material Structures and Union NOTES
sum of storage spaces required for each member. In struct book the storage space required will be as follows: title 25 + 1 (for NULL to indicate end of string) author 15 + 1 publisher 25 + 1 price 4 year 2 Therefore, the system allots space for 1000 structure variables each with the above requirement. Space is allocated only after seeing the structure variable declaration. Take another example to make the concept clear. You know that the bank account of each account holder is a record. A structure for it
may be defined as follows:
struct account { unsigned number; char name [15]; int balance ; } a1, a2; Instead of declaring separate structure variables such as struct account a1, a2; you can use coding as in the example given above. Here, the variables are declared just after the closing brace of the structure declaration and terminated with a semicolon. This is perfectly correct. The declaration of the members of the structure is clear; the balance has been declared as an integer instead of a float to make it simple. This means that the minimum transaction is a rupee. 10.4 USER-DEFINED DATA TYPES The structure variable declaration is of no use unless the variables are assigned values. Here, each member has to be specifically accessed for each structure variable. For example, to assign the account number for variable a1 you have to specify as follows: a1. number = 0001 ; There is a dot operator
in

between the structure variable name and the member name or tag. Suppose you then want to assign account No.2 to a2, it can be assigned as follows : a2. number = 2; If you want to know the address where a2. number is stored you can use, printf (" %u " , & a2 . number) ; This is similar to other data types. The structure is a complex data type, and therefore you have to indicate which structure variable the member belongs
to,
as otherwise the number is common to all the structure variables such as a1, a2, a3, etc. Therefore, it is necessary to be specific. Assuming that you want to get the value from the keyboard, you can use scanf() as follows: scanf ( " % u ", & a1 . number ) ;
Self-Instructional Material 229 Structures and Union NOTES You can also assign initial values directly as given below:

**97%** **MATCHING BLOCK 70/126** **W**

struct account a1 = { 0001, "Vasu", 1000}; struct account a2 = { 0002, "Ram", 1500 }; All the members are specified. This is similar to the declaration of initial values for arrays. However, note the semicolon after the closing brace. The struct a1 will therefore receive the values for the members in the order in which they appear. Therefore, you must give the values in the right order, and they will be accepted automatically as follows: a1 . number = 0001 a1 . name = Vasu a1 . balance = 1000 Note too, that if the initial values are assigned as above, inside a function, they will be treated as static variables. If they are declared before main, then they will be treated as global variables. Let us write a program to create a structure account, open 2 accounts with initial deposits in the accounts. Deposit Rs 1000 to Vasu's account, withdraw

Rs 500
from Ram's account and print the balance. The following is the example program that demonstrates the above. /*Example 10.1 - structures*/ #include&gt;stdio.h&lt; main() { struct account { unsigned number; char name[15]; int balance; }; static struct account a1= {001, "VASU", 1000}; static struct account a2= {002, "RAM", 2000}; a1.balance+=1000; a2.balance-=500; printf("A/c No:=%u\t Name:=%s\t Balance:=%d\n", a1.number, a1.name, a1.balance); printf("A/c No:=%u\t Name:=%s\t balance:=%d\n", a2.number, a2.name, a2.balance); } Result of the program A/c No:=1 Name:=VASU Balance:=2000 A/c No:=2 Name:=RAM balance:=1500 A simple program was written for a bank transaction. For a deposit, you write a1 . balance = a1 . balance + 1000 ;
230 Self-Instructional Material Structures and Union NOTES Therefore the balance is updated. Similarly, when an amount is withdrawn then
the balance is adjusted. However, in practice the user cannot write a program for each credit and deposit. 10.4.1

**92%** **MATCHING BLOCK 71/126** **W**

Array of Structures Let us now create an array of structures for the account. This is nothing but an array of accounts, declared with size. Let us restrict the size to 5. The records will be created by using keyboard entry. The program is given below: /*program 10.2 - to demonstrate structures*/ #include&gt;stdio.h&lt; main() { struct account { unsigned number; char name[15]; int balance; }a[5]; int i; for(i=0; i&gt;=4; i++) { printf("A/c No:=\t Name:=\t Balance:=\n"); scanf("%u%s%d", &a[i].number, a[i].name, &a[i].balance); } for(i=0; i&gt;=4; i++) { printf("A/c No:=%u\t Name:=%s\t Balance:=%d\n", a[i].number, a[i].name, a[i].balance); } } Result of the program A/c No:= Name:= Balance:= 1 suresh 5000 A/c No:= Name:= Balance:= 2 Lesley 3000 A/c No:= Name:= Balance:= 3 Ahmed 5500 A/c No:= Name:= Balance:= 4 Lakshmi 10900 A/c No:= Name:= Balance:= 5 Thomas 29000 A/c No:=1 Name:=suresh Balance:=5000 A/c No:=2 Name:=Lesley Balance:=3000 Self-Instructional Material 231 Structures and Union NOTES A/c No:=3 Name:=Ahmed Balance:=5500 A/c No:=4 Name:=Lakshmi Balance:=10900 A/c No:=5 Name:=Thomas Balance:=29000 The structure array has been declared as part of structure declaration as a[5].

You will see that the
individual elements of the 5 accounts are scanned, and printed in the same order. Note that when we scan a name, we do not give the address but actual name of the variable as in a[I].name, since it is a string variable. Remember this uniqueness. This program basically gets the 5 structures or records pertaining to 5 account holders. Thereafter, the details of the 5 accounts are printed using the for statement. The first half of the result was typed by the user and the last 5 lines are the output of the program. 10.5
STRUCTURES AND POINTERS
Structures can be passed by reference. Remember however, that the structure should be defined as a global variable. The following example 10.3 passes structure by reference, and withdrawal of money is processed in the called function. /*program 10.3 - structure pointers & functions*/ #include&gt;stdio.h&lt; struct account { unsigned number; char name[15]; int balance; }; main() { static struct account a1= {001, "VASU", 1000}; struct account debit(struct account *sp, int y); int deb; printf("Enter amount to be withdrawn"); scanf("%d", &deb); debit(&a1, deb); printf("A/c No:=%u\t Name:=%s\t Balance:=%d\n", a1.number, a1.name, a1.balance); } struct account debit(struct account *x, int y) { x-&lt;balance-=y; return *x; } Result of the program Enter amount to be withdrawn299 A/c No:=1 Name:=VASU Balance:=701
Check Your Progress 1. What are structures in C programming? 2. Write the rules to declare a structure. 3. When is a structure declaration of no use?
232 Self-Instructional Material Structures and Union NOTES

89% MATCHING BLOCK 72/126 W

Note that the address of a1 is passed. The prototype declares passing structure by reference and the amount of debit by value. In the called program the debit is adjusted. Note the pointer notation in subtracting the debit amount from the balance. Let us look at some more examples to understand structure pointers, but before that

let us see how we
allocate dynamic memory allocation for structures. Let us say, struct cycle * cp; then to allocate memory dynamically, we can write, cp = (struct cycle * ) malloc (size of (struct cycle ) * n)) ; Here n is the number of structure variables of type struct cycle. In the above program, we have treated structure similar to other data types. However,
the size of the structure is not fixed like basic
data types. We can use the size of operator to get the size of the structure variable. Let us now write a program to print out a sales report for cycles of given number of brands. The program is given below: /* Example 10.4– to demonstrate dynamic allocation

68% MATCHING BLOCK 74/126 SA C_book_FINAL(July 2019).pdf (D54399424)

of memory to structures*/ #include &gt;stdio.h&lt; #include &gt;conio.h&lt; #include &gt;stdlib.h&lt; #include &gt;string.h&lt;

struct cycle { int brand; int stock; int sold; }; void main() { struct cycle *cyclerecs; int n,i,rem; float per; printf("How Many Brands Of Cycles:"); scanf("%d",&n); cyclerecs = (struct cycle *)(malloc(sizeof(struct cycle)*n)); for(i=0;i&gt;n;i++) { clrscr(); printf("\t\t\tCycle %d Details \n",i+1); printf("Brand No.:"); scanf("%d",&cyclerecs[i].brand); printf("No. Of cycles in Stock:"); scanf("%d",&cyclerecs[i].stock); printf("No. Of Cycles Sold :");
Self-Instructional Material 233 Structures and Union NOTES scanf("%d",&cyclerecs[i].sold); } clrscr(); /*clear screen*/ printf("Report Of Cycles Sold :\n\n"); for(i=0;i&gt;n;i++) { printf("Brand :#%d\n",cyclerecs[i].brand); printf("Inventory at day's start :%d\n",cyclerecs[i].stock); printf("Total Sales:%d\n",cyclerecs[i].sold); rem=cyclerecs[i].stock-cyclerecs[i].sold; printf("Inventory at day's end :%d\n",rem); } } Result of the program Report Of Cycles Sold: Brand :#1 Inventory at day's start:34 Total Sales:23 Inventory at day's end:11 Brand :#2 Inventory at day's start:4567 Total Sales:2314 Inventory at day's end:2253 The program allocates memory dynamically, and hence any number of brands of cycles can be handled. The number of brands can be entered at run time. The structure variables are then assigned values through the keyboard, before finally printing the details pertaining to each brand. Actually, the data of sales, etc., are received from the user. It only prepares the report based on the data given. Now create a database of employees with the following data: Empno Name Basic (pay) Grade Get the data and arrange the records on the basis of their names. Example 10.5 will create and print the database of employees. /* Example 10.5– Employee database sorted on employee name */ #

87% MATCHING BLOCK 82/126 SA C_book_FINAL(July 2019).pdf (D54399424)

include &gt;stdio.h&lt; #include &gt;conio.h&lt; #include &gt;stdlib.h&lt; #include &gt;string.h&lt;

struct employee {
234 Self-Instructional Material Structures and Union NOTES int empno; char *name; float basic; int grade; }; void sort(struct employee *emprecs,int n); main() { struct employee *emprecs; int n,i,len1; char name1[50]; printf("How Many Employees :"); scanf("%d",&n); emprecs = (struct employee *)(malloc(sizeof(struct employee)*n)); for(i=0;i&gt;n;i++) { printf("\nEmployee Number in integer :\t"); scanf("%d",&emprecs[i].empno); printf("\nEmployee Name :\t"); scanf("%s",name1); len1=strlen(name1); emprecs[i].name = (char *)(malloc(len1)); strcpy(emprecs[i].name,name1); printf("\nEmployee Salary in Rs.p :\t"); scanf("%f",&emprecs[i].basic); printf("\nEmployee Grade in integer :\t"); scanf("%d",&emprecs[i].grade); } sort(emprecs,n); clrscr(); printf("Sorted Employees Details :\n\n"); printf("Emp. No.\tName\t\tSalary\tGrade\n\n"); for(i=0;i&gt;n;i++) { printf("%d\t\t",emprecs[i].empno); printf("%s\t\t",emprecs[i].name); printf("%7.2f\t",emprecs[i].basic); printf("%d\n",emprecs[i].grade); } } void sort(struct employee *emprecs,int n) { struct employee temp;
Self-Instructional Material 235 Structures and Union NOTES int i,j,ret;
for(i=0;i&gt;n-1;i++) for(j=i+1;j&gt;n;j++) { ret=strcmp(emprecs[j].name,emprecs[i].name); if (ret&gt;0) {
temp=emprecs[i]; emprecs[i]=emprecs[j]; emprecs[j]=temp; } } } Result of the program Sorted Employees Details: Emp. No. Name Salary Grade 3 Arasan 12125.00 5 2 Malati 23000.00 7 1 Sanjay 35000.00 9 By declaring 'employee' as a pointer to a structure, i.e., emprecs, the array size n has been kept flexible. The number of employees is obtained dynamically and memory is allocated dynamically. The names are sorted in a function sort through bubble sort. The structure is passed by reference. The employees records sorted as per their names alphabetically, are printed in main(). Peruse the function sort. The ith and jth name are compared using strcmp(). Then, depending on the result, the corresponding structures are exchanged. See how easy it is to sort records; they are handled in the same manner as you handle array of numbers. 10.6
PASSING STRUCTURES TO FUNCTIONS Structures can be copied individually, memberwise as well as at one go. For example, let a3 and a1 be struct account. The following are valid. a3.number = a1.number; a3.balance = a1.balance; Here, the members of a1 are copied into a3, one by one. You can also write a3=a1; when all the elements of a1 will be copied to a3. The latter coding can be used if all elements are to be copied and the former if some members are to be copied selectively. Note that structures cannot be compared as, for example, if (a4 == a2). This is not a valid operation. You can pass a structure into a function, element by element. This is implemented and shown in Example 10.6: /*Example 10.6 - passing structure element to function*/

**98%** **MATCHING BLOCK 75/126** W

include&gt;stdio.h&lt; #include&gt;string.h&lt; main() { struct account { unsigned number; char name[15]; int balance; }; static struct account a1= {001, "VASU", 1000}; int credit(unsigned a, char *n, int d); printf("A/c No:=%u\t Name:=%s\t Balance:=%d\n", a1.number, a1.name, a1.balance); a1.balance=credit(a1.number, a1.name, a1.balance); printf("A/c No:=%u\t Name:=%s\t new balance:=%d\n", a1.number, a1.name, a1.balance); } int credit(unsigned a, char *name, int b) { int d; unsigned num; char *client; printf("Enter account number\n"); scanf("%u", &num); if(a==num) { printf("Enter name in caps\n"); scanf("%s", client); if(strcmp(name, client)== 0) { printf("enter deposit made\n"); scanf("%d", &d); b+=d; return b; } else { printf("name does not match\n"); return b; } }

else {
Self-Instructional Material 237 Structures
and Union NOTES
printf("account number does not match\n"); return b; } } Result of the program A/c No:=1 Name:=VASU Balance:=1000 Enter account number 1 Enter name in caps VASU enter deposit made 4600 A/c No:=1 Name:=VASU new balance:=5600 Now look at the program carefully.
A function
prototype has been declared in main() as given below: int credit(unsigned a, char *n, int d); Here there is no reference to structure at all. A structure a1 is passed to function credit by passing individual elements of a structure. In function credit these values are received. Then the deposit is entered and added to the balance after checking the correctness of the details of account. The new balance is returned to main and stored in a1. 10.6.1 Structures to Functions Passing each member of the structure is a tedious job. The entire structure can instead be passed to the function making for easy handling. The above example can be altered by passing an entire structure, as follows: /*Example 10.7 - passing entire structure to function*/ #include&gt;stdio.h&lt; struct account { unsigned number; char name[15]; int balance; }; main() { static struct account a1= {001, "Vasu", 1000}; struct account credit(struct account x); printf("A/c No:=%u\t Name:=%s\t Balance:=%d\n", a1.number, a1.name, a1.balance); a1=credit(a1); printf("A/c No:=%u\t Name:=%s\t Balance:=%d\n", a1.number, a1.name, a1.balance); }
238
Self-Instructional Material Structures and Union NOTES
struct account credit(struct account y) { int x; printf("enter deposit made"); scanf("%d", &x); y.balance+=x; return y; } Result of the program A/c No:=1 Name:=Vasu Balance:=1000 enter deposit made 6700 A/c No:=1 Name:=Vasu Balance:=7700 If you want to pass a structure, the called function should also know the structure and hence the structure has to be declared before the main function. Therefore structure account has been declared as a global structure. The function credit is declared with return data type
struct
as follows: struct account credit(struct account x); Thus you pass and return struct account. Then credit is called by simply passing structure a1. In the called program, the deposit is added to the balance and updated. This is returned to the main() where the updated record is printed. 10.7 SELF-REFERENTIAL STRUCTURES You have been nesting
if statement
and loops so far. You can now create structures within structures. Here, a structure defined earlier can become a member of another structure; for example, you can create a structure called deposit using other data types and structure account. The declaration of the basic structure should precede the desired structure as follows: struct account { unsigned number ; char name [15]; int balance ; }; struct deposit { struct account ac ; int amount; int years; }; You can write a program to demonstrate the concept. /*Example 10.8 - structure within structure*/ #include&gt;stdio.h&lt; main() {
Self-Instructional Material 239 Structures and Union NOTES
struct account { unsigned number; char name[15]; int balance; }; struct deposit { struct account ac; unsigned amount; int years; }d2; static struct deposit d1= {001, "VASU", 1000, 50000, 3}; d2=d1; /*structure copy*/
printf("
A/c No:=%u\t Name:=%s\t Balance:=%d\tdeposit=%u\tterm=%d\n", d2.ac.number, d2.ac.name, d2.ac.balance, d2.amount, d2.years); }
Result of the program
A/c No:=1 Name:=VASU Balance:=1000 deposit:=50000 term:=3 You have created structure account. Then you have created another structure deposit. In the deposit structure struct account ac is one of the members and 2 more members, amount and years have been declared. Next, structure deposit d1 is initialized. The first 3 elements pertain to the members of struct account and the last two for amount and years respectively. Now d1 is copied to d2 in a simple manner, and the deposit details of d2 are printed. Note that whenever members of included structures are accessed, you find two dots, instead of the usual one dot. This is essential since d1.name is invalid. Since name is in ac you have to access it as d1.ac.name. However, nesting can be up to any level. You can create one more level of nesting as shown: struct account { unsigned number ; char name [15]; int balance ; }; struct deposit { struct account ac; int amount ;
240
Self-Instructional Material Structures and Union NOTES

int years ; }; struct loan { struct deposit dep ; int amount; char date [10]; }; See that struct deposit is included as a member of loan. Let us declare: struct loan loan1; Now to access loan amount we have to specify: loan1 . amount To access deposit amount, we have to specify: loan1 . dep . amount To access the balance we have to specify: loan1 . dep . ac . balance Therefore, usage of the same variable name amount has not resulted in a conflict, since it has to be seen in which context it is defined. Thus, structures can be used within structures without difficulty.

Application of Structures Structures as you know can be used for database management. They can be used in several applications such as libraries, departmental stores, banking, etc. Structures are also used in C++. The syntax of structures is similar to classes in C++. Structures contain data, but classes contain data and functions. Structures are also used in a variety of other applications such as: • Graphics • Formatting floppy discs • Mouse movement • Payroll
Hence,
structure is a very useful construct. 10.8
UNION Union is a variable, which holds at a common assigned area different data types of varying sizes at different points in time. Assume that a program, at different points in time of execution uses a double, a float, an integer and a string. In the normal course we would have to allocate memory space for each data type. Assuming that we want to use only one of them at any time and if we do not mind losing the values, we can save a lot of memory space by declaring a common area for storing them. If
we
use dedicated memory for each variable the space would remain unutilized most of the time during program execution. This common storage area can be declared as a union as shown
below: Check Your Progress 4. Why is an entire structure passed to the function? 5. Can a structure be created within a structure? Self-Instructional Material 241 Structures and Union NOTES
union uname { double d; float f; char s[ ]; int i ; } un1 ; See the resemblance between structure declaration and union declaration. The common properties are: (
a) They can have a name optionally, such as uname. (b)
They can contain members of varying data types. (c) The declarations end with a semicolon. (d) Union members can be accessed in the same way as structure members, as
given below: union_name. member or union_pointer -&lt; member (e)
A union can be assigned to another union such as, un2 = un1; Where the structure of un1 along with its members are copied to un2. However, the difference is: (
a)
The memory size of the structure variable is the sum of the sizes of its members, whereas in union it is the largest size of its members. (b)
It is the programmer's responsibility to keep track of which type is currently in
use unlike in structure where no member is lost. (c) In structures all members can be initialized whereas
a union can only be initialized with a value of the type of its first member.
A program using union to store either an int value or float value is given below: * Example 10.9– Demonstrate union */ #include &gt;stdio.h&lt; #include &gt;conio.h&lt; union sel { int n; float f; }; void main() { union sel m1; void printval(union sel *m1,char type); char type ='p'; char cont ='y'; while(cont=='y') { printf("\nWant to enter Integer Or Float (
i/f):"); type=getche();
242
Self-Instructional Material Structures and Union NOTES
if(type=='i') { printf("\nEnter Integer Value :"); scanf("%d",&m1.n); } else { printf("\nEnter Float Value :"); scanf("%f",&m1.f); } printval(&m1,type); printf("\nWant to continue- enter (y/n):"); cont=getche(); } } void printval(union sel *m1,char type) { if(type=='i') printf("Integer Value Is %d\n",m1-&lt;n); else printf("Float Value Is %f\n",m1-&lt;f); } Result of the program Want to enter Integer Or Float (i/f):i Enter Integer Value :456 Integer Value Is 456 Want to continue- enter (y/n):y Want to enter Integer Or Float (i/f):3 Enter Float Value :300.30 Float Value Is 300.299988 Want to continue- enter (y/n):n Union sel is declared before main(), with two members int n and float f. A function printval is also declared. Depending upon whether the user wants an integer value or float value, type is set to i or f. If type is i, integer value is received and if it is f, a float value is received. The value received is printed in the function printval. Note carefully how the pointer to union is declared in the function prototype and header. You have to declare union whenever you pass a union to a function. It is declared as union sel * m1. As we would have declared int * i, the type here is union sel. If you have perused the result, you would find that the program asks for float value, although 3 has been typed instead of f. It is because of the design of the program. It will ask for float value when a character other than i is typed. You can take this as an exercise to correct the program to ask for float value only when "f" is typed.
Let us consider another example.
Self-Instructional Material 243 Structures and Union NOTES

93% MATCHING BLOCK 77/126 W

In the case of an employees database we would like to store either the father's name (in the case of men and unmarried women), the husband's name (in the case of married women) or guardian's name. This can be represented as a union as shown below: Union guardian { char father [10]; char husband [10]; char guardian [10]; } e ;

We
have defined a union guardian of employees. By using this you can save memory space. The memory required will be 10 bytes. Had you considered this as a structure you would have used 30 bytes and all three variables will be used. This is not required since only one value will be present at any time, and Union is useful in such cases. You can now define a structure for an employee database with union embedded. You have to declare union above structure since union will be one of the members of the structure. You can define it as follows : union guardian { char father [10]; char husband [10]; char guardian [10]; } u; struct employee { char name [15]; float basic; char birthdate [10]; union guardian u; } emp [2] ; You know how to refer to members of structures. For example, you can refer to the name of the employee's father as: emp[0]. u . father. Now look at Example 10.10. /*Example 10.10 union within structure*/ #include &gt;stdio.h&lt; main() { union guardian { char *father; char *husband; char *guardian; }u; struct employee {
Check Your Progress 6. Define union. 7. State three common properties of structure declaration and union declaration. 8. Write three differences between structure and union.
244
Self-Instructional Material Structures
and Union NOTES char *name; float basic; char *birthdate; union guardian u; }emp[2]; int i; emp[0].name="RAM"; emp[0].basic= 20000.00; emp[0].birthdate= "19/11/1948"; emp[0].u.father= "SWAMY"; emp[1].name="SITA"; emp[1].basic= 12000.00; emp[1].birthdate= "19/11/1958"; emp[1].u.husband="RAM"; for(i=0;i&gt;2;i++) { if( emp[i].basic ==12000)
printf("Name:%s\nbirthdate:%s\nguardian:%s\n",
emp[i].name, emp[i]. birthdate,
emp[i].u.husband); } } Result of the program Name:SITA birthdate:19/11/1958 guardian:RAM The employee details are given in the form of struct emp which also contains union for guardian. All the string variables have been given as pointers to char. The struct is defined as an array of size 2, and the initial values of emp[0] and emp[1] are assigned for each member. Next the data of employees whose basic equals 12000 are printed. Note how the union is handled. You will note that in such cases when
only one
of the three will be entered, there is no need to reserve memory for storing 3 variables. It is enough to store only one of them. Union provides a methodology for achieving this. 10.9
SUMMARY
In this unit, you have learned about structures and union. A
structure
is a user-defined data type like an array. While an array contains elements of the same data type, a structure
contains members of varying data types.
Thus, it is useful to represent real-life examples. The structure declaration ends with a semicolon. As in function prototype, structure declaration does not lead to allocation of memory, and memory is allocated only when structure variables are declared. Structure variables can be declared either just before closing structure declaration or afterwards. The tag for structure is optional. The elements of a structure can be referenced or copied individually through the dot operator. A structure can also be straightaway assigned to another structure. An array
Self-Instructional Material 245 Structures and Union NOTES of structures is a replication of structures like an array of any other variables. An array of structures contains the same structure with varying data. A structure can be passed to a function either entirely or element-wise. Pointers to structures are similar to other pointers, and use -&lt; operator instead of dot operator when individual members are accessed. Structures can be passed by reference, and in such cases, should be declared before main(). Dynamic memory allocation using malloc and calloc can be carried out when pointers to structures are used. Structures can be nested. In such cases, the most basic structure should be on top. It can be used in the structure declared just below it. This, in turn, can be used in a structure declared thereafter and so on. You learnt that a structure is useful in database management and graphics, and is the forerunner for classes in C++. Thus, structures are clearly defined for manipulating complex data elements. Finally, in this unit, you have learned that
Unions help in conserving memory. As only one of the many variables is used at a time,
it is not necessary to allocate spaces, for storing all the variables, but enough to allocate space for storing only one of them. If variables of different data types are members of the union, the size equivalent to the largest of the members will be allocated. At any time, only one of the members, i.e., the last stored variable will exist. All other variables would have been lost. If not specific, the value assigned to the union will be stored on the first listed variable. Values can be assigned specifically to any of the members. The syntax of union is similar to that of a structure. 10.10 KEY TERMS • Structure: It represents collections of a number of items.
struct is a keyword or a reserved word of 'C'
programming. It has a structure tag or name.
The beginning of the structure is indicated by opening brace. • srtcmp(): This function is used to compare two defined strings in a program. •
union: It is a variable, which holds at a common assigned area different data types of varying sizes.

A lot of memory space is saved by declaring a common area for storing them. 10.11 ANSWERS TO 'CHECK YOUR PROGRESS' 1. Structures can represent items of varying data types pertaining to an item.
struct is a keyword or a reserved word of 'C'.
The structure consists of different types of data elements which is different from array. 2. (
a) struct is the header of a structure definition. (b) It can be followed by an optional name for the structure. (c) Then the members of the structure are declared one by one within a block. (d) The block starts with an opening brace, but ends with a closing brace followed by a semicolon. (e) The members can be of any data type. (f) The members have a relation with the structure; i.e., all of them belong to the defined structure and they have identity only as members of the structure and not otherwise. Therefore if you assign a name to author, it will not be accepted. You can only assign values to book.author.
246
Self-Instructional Material Structures and Union NOTES 3.
The structure variable declaration is of no use unless the variables are assigned values. Here each member has to be specifically accessed for each structure variable. 4.
The entire structure can be passed to the function making for easy handling. 5.
Yes, a structure can be created within a structure.
Here, a structure defined earlier can become a member of another structure. 6.
Union is a variable, which holds at a common assigned area different data types of varying sizes at different points in time. 7.
The common properties of structure and union declaration are: (a)
They can have a name optionally, such as uname. (b) They can contain members of varying data types. (c) The declarations end with a semicolon. 8. (
a)
The memory size of the structure variable is the sum of the sizes of its members, whereas in union it is the largest size of its members. (b)
It is the programmer's responsibility to keep track of which type is currently in
use unlike in structure where no member is lost. (c) In structures all members can be initialized whereas
a union can only be initialized with a value of the type of its first member. 10.12
QUESTIONS AND EXERCISES Short-Answer Questions 1. State whether True or False and give reasons: (a)
A structure consists of similar data types. (b) An array of structures consists of similar data types. (c) struct is the header of the structure declaration. (d) Initialization in a function has to be of storage class static. (e) A pointer to a structure has the same properties as other data types. (f) A structure is a derived data type. (g) The dot operator can be used with pointers to structures also. (h) Union should have a name tag. (i) A union can be assigned to another . (j) Assignment of a value to a member of a union cancels the previous assignment to another member. 2. Compute what the following program will do: #include &gt;stdio.h&lt; main() { union unnamed { char *ch ; long unsigned s;
Self-Instructional Material 247 Structures and Union NOTES }u1,u2; u1.s = 43210; printf(" %lu" , u1.s ); printf(" %s" , u1.ch); u2 . ch = "ab"; printf (" %s ", u2 . ch); printf (" %lu" , u2 . s); } Long-Answer Questions 1. Give descriptive answers with examples. (a) Structures vs arrays (b) Passing structures to functions (c) Pointers to structures and functions (d) Application of malloc and calloc to structure printers (e) Nested structures 2. Write programs for the following. (a) A menu based bank account management software. 1 - Opening account 2 - Deleting account 3 - Deposit 4 - Withdrawal 5 - Printing summary of accounts - total balance at the end of the day. (b) Design a library database which contains records of all books and issue status. (c) Develop a simple pay roll package with the following structure. Name of employee Designation Basic pay Total Allowances Total deductions Net pay The database should print a monthly summary employee-wise and as per head of account. The program should run till a specified key is pressed, i.e., the menu will appear till you want to quit. 10.13
FURTHER READING Gottfried, Byron S. Programming with C, 2nd edition. Schaum's Outline Series. New York: McGraw-Hill, 1996.
248
Self-Instructional Material Structures and Union NOTES Forouzan, Behrouz A. and Richard F. Gilberg. Computer Science: A Structured Programming Approach Using C, 2nd edition. California: Thomson, Brooks Cole, 2000. Dey, Pradip and Manas Ghosh. Computer Fundamentals and Programming in C. New Delhi: Oxford Higher Education, 2006. Bronson, Gary J. A First Book of ANSI C, 3rd edition. California: Thomson, Brooks Cole, 2000. Kanetkar, Yashwant. Understanding Pointers in C.
New Delhi: BPB Publication, 2001. Kanetkar, Yashwant. Let us C. New Delhi: BPB Publication, 1999.
Kernighan, Brian W. and Dennis Ritchie. C Programming Language, 2nd edition. New Jersey: Prentice Hall, 1988. Foster, W. D. and L. S. Foster. C by Discovery. Boston: Addison-Wesley, 2005. Kanetkar, Yashwant. Working with C. New Delhi: BPB Publication, 2003.
Horton, Ivor. Instant C Programming. New Jersey: Wrox Press (John Willey & Sons), 1995. Lawlor, Steven C. The Art of Programming Computer Science with 'C'. New Jersey: West Publishing Company, 1996.
MODULE – V
250 Self-Instructional Material Data Files NOTES
Self-Instructional Material 251 Data Files NOTES UNIT 11 DATA FILES Structure 11.0 Introduction 11.1 Unit Objectives 11.2 Why Files? 11.3 File Pointer 11.4 Opening

and closing a Data File 11.5 Concept of Binary Files 11.6 Formatted I/O Operations with Files 11.7 Writing and reading a Data File 11.8 Unformatted Data Files 11.9 Processing a Data File 11.9.1

File Copy 11.9.2 Line Input/Output 11.9.3 Use of the Command Line Argument 11.9.4 Personal File of an Employee 11.10 Summary 11.11 Key Terms 11.12
Answers to 'Check Your Progress' 11.13 Questions and Exercises 11.14 Further Reading 11.0 INTRODUCTION In this unit, you will learn about data files. In C programming, you include &gt;stdio.h&lt;— a file essential for any program to read from a standard input device or to write to a standard output device. It has declaration pointers to three files, namely stdin, stdout and stderr which means that the contents of these files are added to the program, when the program executes. In this unit, you will learn to use either the hard disk drive or the floppy disk drive as the input/output medium. In day-to-day usage of large applications, the standard input/output is neither convenient nor adequate to handle large volumes of data and hence, the disk drives only serve as Input/Output (I/O) devices. You will discuss the usage of files for storing data, popularly known as data files. Data files stored in the secondary or auxiliary storage devices, such as hard disk drives or floppy disks, are permanent unless deleted. In contrast, what is written to the monitor is only for immediate use. The data stored in disk drives can be accessed later and modified, if necessary. Further, in this unit, you will learn about file pointers, binary mode and text mode operations, reading a data file and processing a data file. 11.1 UNIT
OBJECTIVES After going through this unit, you will be able to: • Understand the
basic concept of
data files • Understand the
need for files • Explain file pointer
252 Self-Instructional Material Data Files NOTES •

**92%** **MATCHING BLOCK 79/126** W

Open and close a data file • Understand the importance of binary files • Understand formatted I/O operations with files • Write and read a data file • Identify unformatted data files • Process a data file 11.2

WHY FILES? For programming &gt;stdio.h&lt; is included
in every file. This file is essential for any program to read from standard input device or to write to the standard output device. The file &gt;stdio.h&lt; has declarations to the pointers to three files, namely stdin, stdout and stderr. It means that the contents of these files are added to the program, when the program executes. Each of the files performs an essential task as follows: (a) stdin facilitates usage of the standard input device for program execution and normally points to the keyboard, which is the default input device. (b) stdout facilitates the usage of a standard output device where program output is displayed and points to the video monitor. (c) stderr facilitates sending error messages to the standard device that is again the monitor. stdin, stdout and stderr are pointers or file pointers and are declared in &gt;stdio.h&lt;. So far
you have
been using stdin and stdout for input and output.
In
this unit, we will learn to use either the hard disk drive or the floppy disk drive as the input/output medium. In day-to-day usage of large applications, the standard input/output is neither convenient nor adequate to handle large volumes of data and hence, the disk drives only serve as Input/Output (I/O) devices. You will learn about the usage of files for storing data, popularly known as data files. Data files stored in the secondary or auxiliary storage devices, such as hard disk drives or floppy disks, are permanent unless deleted. In contrast, what is written to the monitor is only for
the immediate
use. The data stored in disk drives can be accessed later and modified, if necessary. In C, we come across two types of files: a. Stream-
oriented b. System-oriented System-oriented files or low-level files are more closely related to the operating system and hence, require more complex programming skills to use them. They may be found to be more efficient than the former in some cases, but we will not discuss them further because of their complexity. Instead, we will discuss stream- oriented files only in this
unit.
Stream-oriented files are also called standard files. Data can be stored in
the
standard files in two ways as given below: • Storing characters or numerals consecutively. Each character is interpreted as an individual data item. • The data items are arranged in blocks in an unformatted manner. Each block may be an array or a structure.
Self-Instructional Material 253 Data Files NOTES
Let us see how disk I/O is organized. If the file is stored in a floppy or hard disk drive, the following actions are involved in reading from the file: • Finding out where the data is. • Positioning the head over the correct location on the disk. • Reading the content. • Transmitting to the main memory. Similar activities are involved in writing to a disk as well. If the computer, or more specifically the operating system, which handles files in a computer, reads or writes one character at a time comprising the
four
steps listed above, then it will be uninteresting and the response will be too slow. It may cause wear out of the storage system quickly. Therefore, it would be better to receive large volumes of data or characters to a buffer in the computer system and then perform whatever actions are dictated by the program. Similarly, all characters to be written can be collected in a buffer and written on to the disk, either after the buffer is full or after the operation is completed. This will minimize the overheads required for the read or write operations. The buffer is also,
the

memory, which
is used to store data temporarily without the knowledge of the user. In fact, you created a buffer and stored values into it before printing them using the sprintf() function. The concept is similar here also. This is a good practice. Therefore, the characters are read or written through a buffer assigned by the system. The operations are essentially performed as depicted pictorially below:
FILE BUFFER SYSTEM 11.3 FILE POINTER

What is a file pointer? It is a pointer to a file, just like other pointers to arrays, structures, etc. It points to a structure that contains information about the file. The information connected with a file is as follows: • Location of the buffer • The position in the file of the character currently being pointed to • Whether the file is being read or written • Whether an error has occurred or the end of the file has been reached You do not need to know the details of these because stdio.h handles it elegantly. There is a structure with typedef FILE in stdio.h, which handles all file-related tasks as above, whether it is in the floppy or the hard disk drive. Therefore, in order to use a file without difficulty,

you
have to include stdio.h and declare a file pointer, which points to FILE as shown: FILE * fp; Therefore,
the file pointer points to a structure, which contains information about the file
management functions. When you open a file, and
when
the opening of the file is successful, the file pointer will point to the first character in the file. In other words, the file gets opened and loaded to the buffer. NULL is a macro defined in &gt;stdio.h&lt;, which indicates that file open has failed. Therefore, when file open is successful, the file pointer will point to the address of the buffer, which will be a non-zero integer value. If not, the file pointer will get a value of NULL, which is 0.
254
Self-Instructional Material Data Files NOTES
The file pointer will point to the next character after the first one is fetched or copied on to the system. The structure FILE keeps track of where the pointer remains at any point in time after opening the file. It keeps track of which files are being used. It also knows whether the file is empty, the end of the file has been reached or an error has occurred.
You do not have to worry about the
file management tasks once a file pointer has been declared in our program to point to FILE. Since FILE is known to &gt;stdio.h&lt;, you do not have to bother about it. This declaration of structure FILE has relieved the programmer from most of the mundane jobs.
11.4 OPENING AND CLOSING
A DATA FILE
Any file has to be opened for any further processing, such as reading, writing or appending, i.e., writing at the end of the file. The characters will be written or read one after another from the beginning to the end, unless otherwise specified. You have to open the file and assign the file pointer to take care of further operations. Hence, you can declare, FILE * fp; fp = fopen (“filename”, “r”);
The
filename is the name of the file, which you want to open. You must give the path name correctly so that the file can be opened. ‘r’ indicates that the file has to be opened for reading purposes. fp = fopen (“Ex1.C”, “r”); will enable opening file Ex1.C. Therefore, the arguments to fopen() are the name of the file and the mode character. Obviously w is for write, a for append, i.e., adding at the end of the file. If these characters are specified, the operations as indicated can be performed after opening the file. It is, therefore, essential to indicate the operations to be performed before opening the file. When the file is opened in the ‘w’ mode, the data will be written to the file from the beginning. This means that if the named file is already in existence, the previous contents will be lost due to overwriting. If the file does not exist, then a file with the assigned name will be opened. When the append mode is specified, the writing will start after the last entry, or in other words, previous contents of the file will be preserved. FILE provides the link between the operating system and the program currently being executed. FILE is a structure containing information about the corresponding files, including information such as: • The location of the file • The location of the buffer • The size of the file After the command is executed in the read mode, the file will be loaded into the buffer if it is present. If the file is absent, or the file specification is not correct, then the file will not be opened. If the opening of the file is successful, the pointer will point to the first character in the file, and if not, NULL is returned, meaning that the access is not successful. The fopen() function returns a pointer to the starting address of the buffer area associated with the file and assigns it to the file pointer, fp in this case.
Check Your Progress 1. Why is the &gt;stdio.h&lt; file essential? 2. Define the tasks performed by stdin, stdout and stderr. 3. Where are the pointers stdio, stdout and stderr declared? 4. Explain system-oriented and stream-oriented files. 5. Write the actions performed while reading a file stored in a floppy or hard disk.
Self-Instructional Material 255 Data Files NOTES

After the operations are completed, the file has to be closed. The syntax for closing file is given below: fclose(filepointer); fclose() also flushes or empties the buffer. The function fputc() performs putting one character into a file. If for every fputc(), the computer prints a character to a file, then it will get tired. Therefore, it collects all the characters to be written onto a file in the buffer. When the buffer is full or when fclose() is executed, the buffer is emptied by writing to the hard disc drive in the assigned file name. 11.5 CONCEPT OF BINARY FILES We can open files in the text mode or the binary mode. In the binary mode, everything will be stored in the binary form and the storage space will be equal to the number of bytes required for the storage of various data types. In the text mode, they will be stored as alphanumeric characters. If you require to use the file in the binary mode, you must use 'rb' for reading, 'wb' for writing, and 'ab' for appending. If you want to store data in the text mode, you have to append t to the mode character as 'rt', 'wt', 'at', etc. Since the default is in the text mode, t will be assumed if nothing is specified after the mode character. Therefore, mode 'w' means opening a text file for writing. The difference between opening files in the binary mode and the text mode are given below in Table 11.1: Table 11.1 Difference between Binary Mode and Text Mode Operations Text Mode Binary Mode New line character is converted (\
n)
to CR|LF combination while writing to file. No such conversion. While reading, CR|LF is converted back to .\n Does not arise. A special character is inserted at the end of the file. While reading the file, EOF is detected. There is no such arrangement. Text mode needs more than the 2 bytes for storing an integer, since it treats each digit as a character. e.g., 30,000 needs 5 bytes. In binary mode the numbers will be stored in the specified width. 30000 needs 2 bytes only. Therefore, binary files and text files are to be processed taking into account their properties as above, although the file could be opened in any mode. The file I/O functions, such as fgetc, fputc, fscanf, fprintf, fgets, fputs, are applicable to the operations in any of the modes. The files can be used to store employee records using structures in a payroll program. Book records can be stored in a file in a library database. Inventories can be stored in a file. However, storing all these in the text mode will consume more space on the file. Hence, the binary mode can be used to create the files. Some files cannot be stored in the text mode at all, such as executable files.
Check Your Progress 6. Define file pointer. 7. What does null macro define? 8. Explain FILE structure.
256 Self-Instructional Material Data Files NOTES 11.6 FORMATTED I/O OPERATIONS WITH FILES We are familiar with reading and writing. So far we
were reading from and writing to standard input/output. Therefore,
we
used functions for the formatted I/O with stdio such as scanf() and printf(). We also used unformatted I/O such as getch(), putch() and other statements. When dealing with files, there are similar functions for I/O. The functions getc(), fgetc(), fputc() and putc() are unformatted file I/O functions similar to getch() and putch(). We will consider the formatted file operations in this section. When it pertains to standard input or output,
we
use scanf() and printf(). To handle formatted I/O with files,
we
have to use fscanf() and fprintf(). We can write numbers, characters, etc. to the file using fprintf(). This helps in writing to the file neatly with a proper format. In fact, any output can be directed to a file instead of the monitor. However, we have to indicate which file we are writing to by giving the file pointer. The following syntax has to be followed for fprintf(): fprintf (filepointer, "format specifier", variable names);
We
are only adding the file pointer as one of the parameters before the format specifier. This is similar to sprintf(), which helps in writing to a buffer. In the case of sprintf(), buffer was a pointer to a string variable. Here, instead of a pointer to a string variable, a pointer to a file is given in the fprintf() statement. Like the string pointer in sprintf(), the file pointer should have been declared in the function and should be pointing to the file. Before writing to a file, the file must be opened in the write mode. You can declare the following: FILE * fp ; fp = fopen ("filename", "wb"); You have to write wb within double quotes for opening a file for writing in the binary mode. Therefore, fopen() searches the named file. If the file is found, it starts writing. Obviously the previous contents will be lost. If a file is not found, a new file will be created. If unable to open a file for writing, NULL will be returned. We can also append data to the file after the existing contents. In this manner, we will be able to preserve the contents of a file. However, when you open the file in the append mode, and the file is not present, a new file will be opened. If a file is present, then writing is carried out from the current end of the file. After writing is completed either in the write mode or the append mode, a special character will be automatically included at the end of the text in case of text files. In case of binary files, no special character will be appended. This can be read back as EOF. Usually it is – 1, but it is implementation-dependent. Hence, it is safer to use EOF to check the end of the text files. 11.7 WRITING AND READING A DATA FILE
Let us look at a program to write numbers to a binary file using fprintf() and then read from the file using fscanf().
It is given in Example 11.1. /*
Example 11.1 - writing digits to a binary file and then reading*/
Self-Instructional Material 257 Data Files NOTES #
include &gt;stdio.h&lt; int main() { int alpha,i; FILE *fp; fp=fopen("ss.doc", "wb"); if(fp==NULL) printf("

could not open file\n"); else { for (i=0; i&gt;=99; i++) fprintf(fp," %d", i); fclose(fp); /*now read the contents*/ fp=fopen("ss.doc", "rb"); for (i=0; i&gt;100; i++) { fscanf(fp,"%d", &alpha); printf(" %d", alpha); } fclose (fp); } } Result of the program 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 What does the program do? (a) The file ss.doc is opened in the binary mode for writing. If the opening of the file was not successful, the message will be displayed and program execution will stop. If successful, the program will enter the else block. Numbers 0 to 99 are generated one after another and written then and there to the file using the fprintf() function. There should be space before %d as shown in fprintf(), otherwise the program may not work. (b) The file is closed using fclose(). (c) Now the same file is opened for reading in the binary mode. (d) Next the text is scanned using fscanf(), one at a time, and written on the monitor using simple printf(). The difference between scanf() and fscanf() is the specification of the file pointer before the format specifier. (e) After reading, the file is closed. The result of the program is read from the file ss.
doc and
printed on the monitor. In all the programs involving files, a similar check to see that file opening was successful should be made. For the sake of improved readability, this statement has been skipped in the rest of the programs.
258
Self-Instructional Material Data Files NOTES
Let us look at one more example of writing, appending and then reading one integer at a time with the help of the for loop. Look at the program below: /* Example 11.2 - writing, then appending digits to a file and then reading*/ #include &gt;stdio.h&lt; int main() { int alpha,i; FILE *fp; fp=fopen("ss.doc", "wb"); for (i=0; i&gt;20; i++) fprintf(fp," %d", i); fclose(fp); fp=fopen("ss.doc", "ab"); for (i=20; i&gt;100; i++) fprintf(fp," %d", i); fclose(fp); /*now read the contents*/ fp=fopen("ss.doc", "rb"); for (i=0; i&gt;100; i++) { fscanf(fp,"%d", &alpha); printf(" %d", alpha); } fclose (fp); } Result of the program The result will be same as Example 11.1
A binary
file is opened in the write mode, and digits from 0 to 19 are written on to the file. The file is then closed using fclose(). The same file is opened in the append mode again, and numbers from 20 to 99 are appended to the file. After the file is closed, the file is opened in the read mode. The contents of the file are read using fscanf() and written to the monitor. Remember to leave a space before %d in fprintf() as otherwise you may have a problem. The file is closed again. We have used the same file pointer, since at any time only one file is in use. If more than one file is to be kept open simultaneously, it may call for multiple pointers. 11.8 UNFORMATTED DATA FILES After having worked with the formatted I/O, let us now look at the unformatted I/O. If you want to read a character from the file, you can use the getc() or fgetc() functions. If alpha is the name of the character variable, you can write, alpha = fgetc (fp); This means the character pointed to by fp is read and assigned to alpha.
A summary of header files and functions are given in Annexure 3. You can also go to the help Check Your Progress 9. How is the data stored in text mode or binary mode? 10. What does mode w mean? 11. Which files cannot be stored in the text mode? 12. What are functions getc(), fgetc(), fputc() and putc()?
Self-Instructional Material 259 Data Files NOTES
screen of the 'C' language system to get more details as well as search for help on any of the library functions. The help screen gives the syntax of the functions and also provides examples in which the function or command is used. Even after reading this book or any other book on 'C', you will not be able to use all the functions. Hence, the best way is to take the help from the help screen whenever other functions are to be used. fgetc() reads the character pointed to by fp. It then increments fp so that fp points to the next character. We can keep on incrementing fp till the end of file, i.e., end of data is reached. When a file is created in the text mode, the system inserts a special character at the end of the text. Therefore, while reading a file, when the last character has been read and the end of the file is reached, EOF is returned by the file pointer. The following program reads one character at a time till EOF is reached from an already created text file, ss.doc. The program is implemented using the do...while statement. /* Example 11.3 - reading characters from a file */ #include &gt;stdio.h&lt; int main() { int alpha; FILE *fp; fp=fopen("ss.doc", "r"); do { alpha=fgetc(fp); putchar(alpha); } while(alpha!=EOF); fclose (fp); } Result of the program Since file ss.doc is read, the output will be same as Example 11.1, if no change has been made in the file. If we were to read from a binary file, EOF may not be recognized. Therefore, a counter can be set up to read a predefined number of characters as given in the previous examples. 11.9 PROCESSING A DATA FILE 11.9.1 File Copy File
copy can be achieved by reading
one character at a time and writing to another file either in the write mode or the append mode. Here it is proposed to read from a file and write to two different files, one in the write mode and another in the append mode. This means we have to open three files in the following manner: FILE * fr, *fw; *fa;
260
Self-Instructional Material Data Files NOTES
You can assign three file pointers as given above. Three files are then opened. You can use any name for the file pointers and there may be as many file pointers as the number of files to be used. The program is given below: /*Example 11.4 - reading from file ss, writing to file ws and appending to file as, all at a time*/ #include &gt;stdio.h&lt; int main() { int alpha; FILE *fr,*fw, *fa; fr=fopen("ss.doc", "r"); fw=fopen("ws.doc", "w"); fa=fopen("as.doc", "a"); do { alpha=fgetc(fr); fputc(alpha, fw); fputc(alpha, fa); putchar(alpha); }while(alpha!=EOF); fclose (fr); fclose (fw); fclose (fa); } After opening the three files, alpha() gets the character, which is written to both the files using fputc(), and the character is also displayed on the screen. This is continued till EOF is received in alpha from ss.doc, the source file. Finally, the files are closed. Verify that our program has worked alright. Since,
we

| 98% | MATCHING BLOCK 81/126 | W |
|---|---|---|

are also writing to the monitor, in addition to writing and appending to files, the program output appears as follows. Result of the program 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 There are some more mode specifiers with fopen like r+, w+ and a+, which are given in the table below: Mode Specifier Purpose r+ Opens an already existing file for reading and writing. w+ Opens a new file for writing as well as reading. a+ Opens an already existing file for appending and reading.

Self-Instructional Material 261 Data Files NOTES 11.9.2

| 97% | MATCHING BLOCK 83/126 | W |
|---|---|---|

Line Input/Output We have discussed writing to and reading from a file, one character at a time, using both the unformatted and formatted I/O for the purpose. We can also read one line at a time. This is enabled by the fgets() function. This is a standard library function with the following syntax: Char * fgets (char * buf, int max line, FILE * fp); fgets() reads the next line from the file pointed to by fp into the character array buf. The line means characters up to maxline –1, i.e., if maxline is 80, each execution of the function will permit reading of up to 79 characters in the next line. Here 79 is the maximum, but you can even read 10 characters at a time, if it is specified. fgets(alpha, 10, fr); Here alpha is the name of buffer from where 10 characters are to be read at a time. The file pointer fr points to the file from which the line is read, and the line read is terminated with NULL. Therefore, it returns a line if available and NULL if the file is empty or an error occurs in opening the file or reading the file. The complementary function to fgets() is fputs(). Obviously fputs() will write a line of text into the file. The syntax is as follows: int fputs (char * buf , file * fp ); The contents of array buf are written onto the file pointed to by fp. It returns EOF on error and zero otherwise. Note that the execution of fgets() returns a line and fputs() returns zero after a normal operation. The functions gets() and puts() were used with stdio, whereas fgets() and fputs() operate on files. We can write a program to transfer two lines of text from the buffer to a file and then read the contents of the file to the monitor. This is shown in Example 11.5. /* Example 11.5 - writing and reading lines on files */ #include &gt;stdio.h&lt; #include&gt;string.h&lt; int main() { int i; char alpha[80]; FILE *fr,*fw; fw=fopen("ws.doc", "wb"); for(i=0; i&gt;2; i++) { printf("Enter a line up to 80 characters\n"); gets(alpha); fputs(alpha, fw); } fclose(fw); fr=fopen("ws.doc", "rb"); 262

Self-Instructional Material Data Files NOTES
while ( fgets(alpha,20, fr)!=NULL) puts(alpha); fclose (fr); } Note carefully the fgets() statement. Here alpha is the buffer with a width of 80 characters. Each line can be up to 80 characters and two lines are entered through alpha to ws.doc. Later on, 20 characters are read into alpha at a time from same file till NULL is returned. Result of the program Enter a line upto 80 characters aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa Enter a line upto 80 characters bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb aaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaabbbb bbbbbbbbbbbbbbbbbb bbbbbbbbbbbbbbbbbb bb More than 20 numbers of a & b were written on to the file. However, since we have specified reading 20 characters at a time. The output appears in 6 lines. Had we specified reading more characters at a time, the number of reads would have reduced. You can try this yourself. Thus you can read and write one line at a time. 11.9.3
Use of the
Command Line Argument
This can be used to copy a file to another file. Assume that the first named file is to be copied to the second named file. We may write a program and convert it into an executable file, specifying the argument in the DOS command line. We may specify as follows at the C&lt; prompt: C &lt; prgname . exe f1.cpp f2.cpp. This means that we want to copy the contents of f1.cpp to f2.cpp. Here the number of arguments are 3, and therefore argc will contain 3. *agrv[0] = prgname.exe *agrv[1] = f1.cpp - source to copy from *agrv[2] = f2.cpp - file where to be copied A character at a time is to be fetched from f1.cpp and put into f2.cpp. 11.9.4 Personal File of an Employee A menu-based program to create employee records on file and calculate the age of any employee on date is given below: /* Example 11.6 Create a Personal File for Employees &
Self-Instructional Material 263 Data Files NOTES

| 97% | MATCHING BLOCK 84/126 | W |
|---|---|---|

calculate the age of any employee ON DATE*/ #include &gt;stdio.h&lt; #include &gt;dos.h&lt; #include &gt;string.h&lt; #include &gt;stdlib.h&lt; #include &gt;conio.h&lt; typedef struct { char name[40]; char code[5]; char dob[9]; char qual[40]; }employee; FILE *fp; struct date today; int main() { int create_emp(); int calc_age(); int ret,ch,onscrn=1; getdate(&today); printf("Today's Date Is %d/%d/%d\nIs It O.K :", today.da_day,today.da_mon,today.da_year); scanf("%c",&ch); onscrn=1; while(onscrn) { clrscr(); printf("1: Create Employee Data File\n"); printf("2: Calculate Age Of Employee\n"); printf("3: Exit From Program\nEnter Your Choice :"); scanf("%d",&ch); switch(ch) { case 1: create_emp(); break; case 2: calc_age(); break; case 3: onscrn=0; break; 264 Self-Instructional Material

Data Files NOTES } } fclose(fp); }

98% MATCHING BLOCK 85/126 W

int create_emp() { employee emp1; int i,n; fp=fopen("emp.dat","a"); clrscr(); printf("How Many Employees :"); scanf("%d",&n); for(i=0;i&gt;n;i++) { clrscr(); printf("Employee %d Details :\n",i+1); printf("\n\nEmployee Name :"); scanf("%s",&emp1.name); printf("Employee Code :"); scanf("%s",&emp1.code); printf("Date Of Birth :(dd/mm/yy)"); scanf("%s",&emp1.dob); printf("Qualification :"); scanf("%s",&emp1.qual); fprintf(fp, "%40s%5s%9s%40s\n", emp1.name, emp1.code,emp1.dob, emp1.qual); } fclose(fp); return(0); } int calc_age() { int ret,nyob,age,llfound=0,onscrn=1; employee emp1; char nam[40],*sear,*ori; char yob[5]; fp=fopen("emp.dat","r"); clrscr(); printf("Employee Name To Search :"); scanf("%s",nam); sear =strlwr(nam); while(onscrn) { ret=fscanf(fp, "%40s%5s%9s%40s\n", emp1.name, emp1.code, emp1.dob, emp1.qual);

Check Your Progress 13. Which are the functions used to read a character from a file? 14. How is file copy achieved? 15. Why are command line arguments used?
Self-Instructional Material 265 Data Files NOTES
if(ret==EOF) { onscrn=0; continue; } ori=strlwr(emp1.name); if(strcmp(sear,ori)==0) { clrscr(); printf("Employee Name :%s\n",emp1.name); printf("Employee Code :%s\n",emp1.code); printf(" Date of Birth :%s\n",emp1.dob); printf("Qualification :%s\n",emp1.qual); strcpy(yob,"19"); strncat(yob,emp1.dob+6,2); yob[4]=0; nyob=atoi(yob); age = today.da_year - nyob; printf("Age of Employee :%d\n",age); getch(); onscrn=0; llfound=1; } } fclose(fp); if (!llfound) { printf("%s Not found in emp.dat\n",nam); getch(); } return(0); } Result of the program Employee Name : saravanan Employee Code : 06 Date of Birth : 02/06/63 Qualification : MBA Age of Employee : 36 You should be able to understand the program by reading the following: The function &gt;dos.h&lt; is included. Look at the online help and see what it does. You will find that it defines various constants and declarations needed for DOS and 8086 specific calls.
You
can use it to get the system date to calculate the age of the employee. After the system date is confirmed, the menu appears. If you choose 1, it calls
266
Self-Instructional Material Data Files NOTES
create_emp and asks for the number of employees. Then it accepts the records of a specified number of employees. The employee record is a structure. After creating the records, you can opt to calculate the employee's ages by entering 2. This invokes the function calc_age. The function asks for the name of employee it must search for. If the name matches, the age will be calculated and displayed. The records are written in the append mode, so you will not lose the records. Note that the structures are written to the file using fprintf() and read from the file using fscanf(). Note also that date is a structure with three members da_day, da_mon, da_year. This example has demonstrated that structures can be written to a file. 11.10
SUMMARY In this unit, you have learned the concept of data files. In real life, the data has to be stored in files, because a large volume of data will be handled. Therefore, it is important to learn how files are handled. C provides an easy way of accessing
files. A pointer to the FILE has to be assigned for each file, which is to be opened, to facilitate their handling.
The
files are to be opened in one of the modes such as read, write or append specifically. After the operation is over, the file has to be closed using fclose().
The file opening for reading or writing
cannot be assumed to take place correctly at all times. Therefore, suitable error statements are to be included in the program and the file can be read one character at a time using the fgetc() and fscanf() commands.
Similarly, one character at a time can be written to a file using the fputc(), fprintf() commands. Lines can be read using fgets() and written using fputs(). The file copy can be implemented using a command line function. This will help in copying files at the command prompt. The first argument will be the executable file name followed by the source file and target file. There are vital differences between writing or reading in the default text mode and the binary mode. You also learnt that the binary mode is preferred over the text mode for large applications involving digits. Two utilities were developed, one for searching for a given string in a document file and another for creating a personal file of employees. 11.11 KEY TERMS • stdin: It
facilitates usage of the standard input device for program execution and normally points to the keyboard, which is the default input device. • stdout:
It facilitates the usage of a standard output device where the program output is to be displayed,
i.e., the monitor. •
stderr: It sends error messages to the standard output device,
i.e., the monitor. • System-oriented files or low-level files: These
are more closely related to the operating system and hence, require more complex programming skills to use them. •
Stream-oriented files: These are also called standard files and store data in two ways. Either each character is interpreted as an individual data item or
the data items are arranged in blocks in an unformatted manner where each block may be an array or a structure.
Self-Instructional Material 267 Data Files NOTES 11.12 ANSWERS TO 'CHECK YOUR PROGRESS' 1.
This file is essential for any program to read from standard input device or to write to the standard output device. The file &gt;stdio.h&lt; has declarations to the pointers to three files, namely stdin, stdout and stderr. It means that the contents of these files are added to the program, when the program executes. 2. (

a) stdin facilitates usage of the standard input device for program execution and normally points to the keyboard, which is the default input device. (b) stdout facilitates the usage of a standard output device where program output is displayed and points to the video monitor. (c) stderr facilitates sending error messages to the standard device that is again the monitor. 3. stdin, stdout and stderr are pointers or file pointers and are declared in &gt;stdio.h&lt;. 4.
System-oriented files or low-level files are more closely related to the operating system and hence, require more complex programming skills to use them.
Stream-oriented files are also called standard files. Data can be stored in
the
standard files in two ways as given below: • Storing characters or numerals consecutively. Each character is interpreted as an individual data item. • The data items are arranged in blocks in an unformatted manner. Each block may be an array or a structure. 5. If the file is stored in a floppy or hard disk drive, the following actions are involved in reading from the file: • Finding out where the data is. • Positioning the head over the correct location on the disk. • Reading the content. • Transmitting to the main memory. 6.
It is a pointer to a file, just like other pointers to arrays, structures, etc. It
points to a structure that contains information about the file.
The information connected with a file is as given below: • Location of the buffer • The position in the file of the character currently being pointed to • Whether the file is being read or written • Whether an error has occurred or the end of the file has been reached 7. NULL is a macro defined in &gt;stdio.h&lt;, which indicates that file open has failed. 8.
FILE provides the link between the operating system and the program currently being executed. FILE is a structure containing information about the corresponding files, including information such as: • The location of the file • The location of the buffer • The size of the file 9. We can open files in the text mode or the
binary mode. In the binary mode, everything is
stored in the binary form and the storage space is equal to the
268
Self-Instructional Material Data Files NOTES
number of bytes required for the storage of various data types. In the text mode, they are stored as alphanumeric characters. If you require to use the file in the binary mode, you must use 'rb' for reading, 'wb' for writing, and 'ab' for appending. If you want to store data in the text mode, you have to append t to the mode character as 'rt', 'wt', 'at', etc. Since the default is in the text mode, t will be assumed if nothing is specified after the mode character. 10. Therefore, mode 'w' means opening a text file for writing. 11.
Some files cannot be stored in the text mode at all, such as executable files. 12.
The functions getc(), fgetc(), fputc() and putc() are unformatted file I/O functions similar to getch() and putch(). 13.
To read a character from the file, you can use the getc() or fgetc() functions. If alpha is the name of the character variable, you can write, alpha = fgetc (fp); 14.
File copy can be achieved by reading one character at a time and writing to another file either in the write mode or the append mode. 15.
They
can be used to copy a file to another file. 11.13
QUESTIONS AND EXERCISES Short-Answer Questions 1. Name the two types of files used in C. 2. What is a file pointer? 3. What is the function of a file buffer? 4. Explain how data is stored in binary format. Long-Answer Questions 1. Write and modify programs to extend the personal file of employees for the following: (a) To modify employee record (b) To delete employee records (c) To sort employee records on their name (d) To maintain a leave record for each employee (e) To calculate the superannuation of each employee 2. Explain system-oriented and stream-oriented files. 3. What
do you mean by formatted I/O with files? 4. Write a program to write and read a data file. 5.

**77%** **MATCHING BLOCK 86/126** **W**

Write a program that reads one character at a time till EOF is reached. 6. Write a program to transfer two lines of text from the buffer to a file and then read the contexts of the file to the monitor. Self-Instructional Material 269 Data Files NOTES 11.14 FURTHER READING Gottfried, Byron S. Programming with C, 2

nd edition. Schaum's Outline Series. New York: McGraw-Hill, 1996.
Forouzan, Behrouz A. and Richard F. Gilberg. Computer Science: A Structured Programming Approach Using C, 2nd edition. California: Thomson, Brooks Cole, 2000. Dey, Pradip and Manas Ghosh. Computer Fundamentals and Programming in C. New Delhi: Oxford Higher Education, 2006. Bronson, Gary J. A First Book of ANSI C, 3rd edition. California: Thomson, Brooks Cole, 2000. Kanetkar, Yashwant. Understanding Pointers in C.
New Delhi: BPB Publication, 2001. Kanetkar, Yashwant. Let us C. New Delhi: BPB Publication, 1999.
Kernighan, Brian W. and Dennis Ritchie. C Programming Language, 2nd edition. New Jersey: Prentice Hall, 1988. Foster, W. D. and L. S. Foster. C by Discovery. Boston: Addison-Wesley, 2005. Kanetkar, Yashwant. Working with C. New Delhi: BPB Publication, 2003. Horton, Ivor. Instant C Programming. New Jersey: Wrox Press (John Willey & Sons), 1995. Lawlor, Steven C. The Art of Programming Computer Science with 'C'. New Jersey: West Publishing Company, 1996.

Self-Instructional Material 271 Low-level Programming NOTES UNIT12 LOW-LEVEL PROGRAMMING Structure 12.0 Introduction 12.1 Unit Objectives 12.2 Register Variables 12.3 Bit-Wise Operations 12.3.1 Conversion of Decimal to Binary 12.3.2 Hexadecimal and Octal Representation 12.3.3 Bit-Wise Operators 12.3.4 Encrypting Selected Bits 12.4 Bit-Wise Assignment Operators 12.5 Summary 12.6 Key Terms 12.7
Answers to 'Check Your Progress' 12.8 Questions and Exercises 12.9 Further Reading 12.0 INTRODUCTION In this unit, you will learn about
low-level programming concept and its components. Register variables have similar characteristics as auto variables. The only difference between them is that while auto variables are stored in the memory, register variables are stored in the register of the CPU. The initial value will be an unpredictable or garbage value. The variables are local to the block and they will be available as long as the blocks are active. The CPU registers respond much faster than the memory, so that the computing time is reduced. Therefore, those variables, which are used frequently, can be declared as register variables. You will learn that binary operators operate on 2 operands, i.e., a+b,a%b, etc., and the unary operators operate on single operand, such as a++, a−−, etc. In C, you can represent numbers as decimal, octal and hexadecimal numbers. They are all stored as integers. You will learn that in software, you organize the digits byte-wise, but the hardware handles it bit-wise. Therefore, bit-wise operators are very useful for directly interacting with the hardware. However, when you do programming, you will be using decimal numbers which are processed by bit-wise operators to carry out the task. You will also learn about Right shift, Left shift, AND, OR, Exclusive OR and Bit-wise assignment operators. 12.1 UNIT OBJECTIVES
After going through this unit, you will be able to: • Understand the basic concept of
low-level programming • Define register variables • Explain bit-wise operations • Convert decimal to binary • Understand bit-wise assignment operators
272 Self-Instructional Material Low-level Programming NOTES 12.2 REGISTER VARIABLES Register variables have similar characteristics as auto variables. The only difference between them is that while auto variables are stored in the memory, register variables are stored in the register of the CPU. The initial value will be an unpredictable or garbage value. The variables are local to the block and they will be available as long as the blocks are active. Why then do you need to declare one more storage class? The CPU registers respond much faster than the memory. After all, you want to access, store and retrieve the stored variables faster, so that the computing time is reduced. Registers are faster than the memory. Therefore, those variables, which are used frequently, can be declared as register variables. They are declared as, register int i ; A memory's basic unit may be 1 byte, but depending on the size of the variable, even 10 contiguous bytes of memory can be used to store a long double variable. Such an extension of size is not possible in the case of registers. The registers are of fixed length like 2 bytes or 4 bytes and therefore, only integer or char type variables can be stored as register variables. Since registers have many other tasks to do, register variables may be defined sparingly. If a register variable is declared and if it is not possible to accept it as a register variable for whatever reasons, the computer will treat it as an auto variable. Therefore, the programmer may specify a frequently used variable in a program as a register variable in order to speed up the execution of the program. 12.3 BIT-WISE OPERATIONS Binary operators are usually assumed to be operators, which operate on binary digits, i.e., bits. However, this is not the convention in C. Binary operators refer to those operators, which operate on 2 operands, i.e., a + b, a % b, etc. In the same way, unary operators are those, which operate on single operand, such as a++, a− − , etc. Bit-wise operators access the internal representation of the numbers, which are bits 0 or 1. These operators apply only to the integer family operands including char. There are six operators for bit-wise operation or manipulation. The operators and their symbols are as follows: & bit-wise AND | bit-wise OR ^ bit-wise exclusive OR &gt;&gt; left shift &lt;&lt; right shift ~ one's complement 12.3.1 Conversion of Decimal to Binary In C, you can represent numbers as decimal, octal and hexadecimal numbers. They are all stored as integers. Since the decimal number is an integer, it is stored in 1 word or 2 bytes. If you had a mechanism for storing a bit, it would have been more economical. As
Self-Instructional Material 273 Low-level Programming NOTES there are no exclusive representations for binary numbers, they have to be stored using the existing storage mechanism. Therefore, each bit of a binary number will also be considered as an integer and stored one after another. If a short integer is available, that is enough, since you have to store only a 0 or 1. Each bit has to be stored as a word of 1 byte or 2 bytes, and the 8 bits of a number have to be stored as an array of 8 integers. Any problem involving bit-wise operation involves conversion of the numbers into binary numbers. Now, consider an algorithm to convert decimal into binary. You repetitively divide the number by 2 and the remainder is assembled to form the binary number. The following is the example to convert 19 into binary. 2 19 2 9 – 1 2 4 – 1 2 2 – 0 2 1 – 0 0 – 1 The equivalent of 19 in binary is 10011. Therefore, when you divide by 2, the first remainder is the least significant bit (LSB) and the last remainder is the most significant bit (MSB). For simplicity, you assume that you convert them into 8-bit numbers. Algorithm 1 gives the method of converting a decimal number into an array of bits. ALGORITHM 1 Step 1: Store the decimal number in integer variable num. Step 2: Declare an array of size 8 to store 8 bits. Step 3: for (i = 0 ; i &gt;= 7; i + + ) { b [ i ] = num % 2 ; num = num / 2 ; } Let us convert 19 using the algorithm to confirm. Step 1 b [ 0 ] = 19 % 2 = 1 Step 2 b [ 1 ] = 9 % 2 = 1 Step 3 b [ 2 ] = 4 % 2 = 0 Step 4 b [ 3 ] = 2 % 2 = 0 Step 5 b [ 4 ] = 1 % 2 = 1 Step 6 b [ 5 ] = 0 % 2 = 0 Step 7 b [ 6 ] = 0 % 2 = 0 Therefore, 19 10 = 0010011 b [ 0 ] = LSB b [ 7 ] = MSB Now you can understand all the bit-wise operations easily. In software, you organize the digits byte-wise, but the hardware handles it bit- wise. Therefore, bit-wise operators are very useful for directly interacting with the

274 Self-Instructional Material Low-level Programming NOTES hardware. However, when you do programming you will be using decimal numbers. The bit-wise operator recognizes the number, carries out the bit-wise operation and gives the result in decimal numbers. To check whether the operation is correct, you have to convert the operand and the result into bits using Algorithm 1 and then see whether it is correct. This is only required initially till you gain confidence, and may not be necessary later on. Although bit-wise operators manipulate the bits, they understand the decimal, octal and hexadecimal numbers and carry out the operation at one go. 12.3.2 Hexadecimal and Octal Representation If you use decimal numbers, you have to convert them into binary format using Algorithm 1. Let us use hexadecimal or octal numbers for understanding the use of the bit-wise operator, since it will be easy for us to convert them into binary form and vice versa. You have to specify the operand in the hex or octal number system, and the result will also be in the same system. Since integers are represented as 16-bit numbers, you can consider integer as 4 hexadecimal numbers each of width 4 bits. Take a decimal number 6666 and write a program to convert it into binary form and print out its equivalent octal and hexadecimal numbers. You have to use Algorithm 1 for conversion from decimal to binary. Hex and octal numbers can be directly printed. The program is as follows: /*Example 12.1*/ /* to convert decimal to binary, octal and hexadecimal*/ #include &gt;stdio.h&lt; int b[16], i; /*global variables*/ int main() { int num=6666; void convert (int num); convert(num); printf("\n16 bit binary number equal to %d\n", num); for (i=15; i&lt;=0; i––) { printf("%d", b[i]); } printf("\nequivalent hexadecimal number\n%x", num); printf("\nequivalent octal number\n%o", num); } void convert(int a) { for (i=0; i&gt;=15; i++) { b[i]=a%2; a= a/2; } }
Self-Instructional Material 275 Low-level Programming NOTES Result of the program 16 bit binary number equal to 6666 0001101000001010 equivalent hexadecimal number 1a0a equivalent octal number 15012 What do you notice? The hexadecimal number groups 4 bits each from the left and gives an equivalent hexadecimal of the 4 bits before the number is printed. For example, the first four bits (from LSB) are 1010 whose equivalent hex number is A or a. Similarly, the other three groups of 4 bits are converted to hex. Thus, you have 4 digits of hexadecimal numbers for the 16-bit binary number. Similarly, the 16-bit binary number is grouped into 3 bits each from the LSB. Therefore, 6 octal numbers will be printed. The last group contains only the MSB. It can only be 0 or 1. The other numbers could be from 0 to 7. Therefore, it is easy to convert octal numbers into binary form. In this case, since MSB is zero, it is omitted and there are only 5 digits in the octal representation. Now, analyse the program. This program uses new concepts. An integer array of size 16 is declared as a global array b[16]. The function main() passes num to convert. In convert, the binary conversion is completed and stored in array b[0] to b[15]. If you print it straightaway, you will get b[0] on the left. To print b[15] on the leftmost side, you print b[15] first in the for loop in main. Although the function convert does not pass any value, the array is known because it is a global array. You have indirectly passed the array through global variable. 12.3.3 Bit-Wise Operators Now using examples, see the operation of bit-wise operators in detail: One's complement operator It changes 1 to 0 and 0 to 1, if x = 1100, ~x = 0011. The usage is as follows: Let us declare i, j as integers; i = 6666; j = ~ i; A table for one's complement of octal numbers is as follows: Octal Number Binary Complement in Binary Complement in Octal 0 000 111 7 1 001 110 6 2 010 101 5 3 011 100 4 4 100 011 3 5 101 010 2 6 110 001 1 7 111 000 0 Check Your Progress 1. Which variables are declared as register variables? 2. Define binary operators. 3. How do bit-wise operators interact?
276 Self-Instructional Material Low-level Programming NOTES Therefore, you can write the one's complements from this table. j = ~ i i = 015012 j = 762765 This can be used for encryption of information for security purposes. Right shift operator unsigned int i , j ; j = i &lt;&lt; 2 ; Here, the bits in i will be shifted 2 places to the right and stored in j. If you write i &lt;&lt; 6, the bits will be shifted right by 6 places. What happens to the leftmost bits? They will all be filled with zeros. Now, i = 6666 in hex will be 1A0A Let, j = i &lt;&lt; 4 ; The bits will be shifted by 4 places. Therefore, j = 01A0 i = 015012 in octal if j = i &lt;&lt; 3 then j = 001501; Left shift operator j = i &gt;&gt; p; This expression will shift i by p bits and will store it in j. What happens to the rightmost bits shifted? Zeros will be inserted whenever a bit is shifted. i = 015012 / * octal number * / j = i &gt;&gt; 3; Their result will be 150120. i = 1A0A; / * hexadecimal number * / j = i &gt;&gt; 4 ; The shifted number will be, A0A0 Shifting again by 4 bits will give 0A00. Check by shifting the octal number 150120 to the left by 3 bits. The result will be 501200. Find out the reason yourself. AND operator It compares two bits, and if both are 1, the output is 1, otherwise zero. This can be used to compare the sets of bits. Assume that you want to check only the 16th bit in the 16-bit word of a number, you can carry out the task using AND operator on the number and word with the 16th bit 1 and all other 15 bits 0. When you compare using AND, the 15 bits will be 0 and 16th bit will be a 0 or 1, depending on the number.
Self-Instructional Material 277 Low-level Programming NOTES Remember to use a single &. You know && stands for logical AND. This will operate on 2 operands. Let us have, a = 015012 octal b = 177777 c = a & b will provide an output of c = 015012 octal. Verify by converting into bits. A = 1A0A hexadecimal b = 0000 c = a & b will produce c = 0000 hexadecimal because b is all zeros. OR operator This is also a binary operator. The output of OR operator will be 0, if both the inputs are 0, and 1 otherwise. Let, a = 015012 octal b = 000000 c = a | b will produce c = 015012 octal Let, a = 015012 octal b = 177777 octal c = a | b will produce c = 177777 octal. This is because b is all ones and hence a | b will automatically produce all 1s, even without looking at a. When b is all zeros, the output will be 1 wherever a is 1. Therefore, the output will be same as a. Exclusive OR operator When one of the 2 operands is 1, we get the output of exclusive OR as 1; otherwise, the output will be 0. a = 1A0A hex = 0001101000001010 Let, b = 1111111111111111 = FFFF hex a ^ b = 1110010111110101 = E5F5 hex Let, c = 0000000000000000 = 0000 hex a ^ c = 0001101000001010 = 1A0A hex b ^ c = FFFF hex 12.3.4 Encrypting Selected Bits Write a program that will convert selected bits of a number into 1 and leave the rest unchanged.

278 Self-Instructional Material Low-level Programming NOTES ALGORITHM 2 Step 1: Get the number num Step 2: Get the bitfrom & bit_to, which will be changed to 1 . Note: LSB = 1 and MSB = 16. Step 3: Convert the number with MSB = b [15], LSB = b[0] Step 4: Change the bits to 1 at the chosen places and leave it as it is, at other places. The program is given below: /*Example 12.2 */ /* to convert selected bits of a given number to 1 and leave the rest as they are*/ #include &gt;stdio.h&lt; int b[16], c[16], i; /*global variables*/ int main() { int num, bitfrom, bit_to, temp; void convert (int num); void build(int bitfrom, int bit_to); printf("Enter the number to be manipulated\n"); scanf("%d", &num); printf("enter bit from and then bit_to for change to 1\n"); printf("MSB=16th bit, LSB 1st bit\n"); scanf("%d%d", &bitfrom, &bit_to); /*Still some will give wrongly as from =4, to=8; This error can be corrected as follows:*/ if (bit_to &lt; bitfrom) /*exchange*/ {temp=bit_to; bit_to=bitfrom; bitfrom=temp;} convert(num); printf("\n binary number equal to number given by you\n"); for (i=15; i&lt;=0; i−−) { printf("%d", b[i]); } build (bitfrom, bit_to); printf("\n binary number with changed bits\n"); for (i=15; i&lt;=0; i−−) { printf("%d", b[i]); } }
Self-Instructional Material 279 Low-level Programming NOTES void convert(int a) { for (i=0; i&gt;=15; i++) { b[i]=a%2; a= a/2; } } void build(int bitfrom, int bit_to) { for (i=0; i&gt;=15; i++) { if (i &lt;= (bit_to-1) && i &gt;= (bitfrom-1)) b[i]=1; } } Result of the program Enter the number to be manipulated 123 enter bit from and then bit_to for change to 1 MSB=16th bit, LSB 1st bit 3 12 binary number equal to number given by you 0000000001111011 binary number with changed bits 0000111111111111 You can see how it works using a simple example. Let, b = 1001 Assume that you want to convert bit 2 into bit 1 as 1. Therefore, from bit b[bit_to−1] to b[bitfrom −1] as 1. b[0]=1, b[1]=1, rest unchanged. The changed number = 1011 Note here that you called LSB as the 1st bit and MSB as the 16th bit and given the bitfrom and bit_to, they will be 1 more than the actual element number in the array b. That is the reason, 1 has been subtracted in the program. The program takes care of operator errors. If you study the result, you will notice that bitfrom and bit_to have been interchanged, but the program was executed without interruption, since it was designed to take care of such mistakes. 12.4 BIT-WISE ASSIGNMENT OPERATORS These operators combine bit-wise operators and assignment operators. The assignment operator is always to the right of the bit-wise operator.
280 Self-Instructional Material Low-level Programming NOTES You can see their usage with examples, assuming 6-bit numbers for simplicity. Let x be 010110, i.e., x = 026 octal Usage of OR AND = operator x | = 014 is equivalent to, x = x | 014 014 = 001100 The OR of x AND 014 = 011110 Therefore, x|=014 is equal to 036 octal. Let us see the usage of other bit-wise assignment operators as well. Usage of & and = operator Let, x = 066 = 110 110 x & = 044 means x = x & 044, 044 = 100100 Therefore, the AND of x and 044 = 100100 = 044 Let,

**27%** **MATCHING BLOCK 87/126** W

x = 055 = 101 101 x & = 020 means, x = x & 020 020 = 010000 x & 020 = 000000 = 0 ^= Operator x ^ = a means, x = x^ a Let x be 11001 a be 01110 x^ a = 10111 Therefore, x = 10111 &lt;&lt;= Operator x &lt;&lt;= a means, x = x&lt;&lt; a or x &lt;&lt; = 2 means, x = x &lt;&lt; 2 if x = 10110,

it will be shifted right twice Therefore, x = 00101. 12.5 SUMMARY In this unit, you have learned about bit-wise operators, which operate on bit representation of numbers. These are very useful for interacting with hardware, which understands Check Your Progress 4. Define AND operator. 5. What is OR operator? 6. What are bit-wise assignment operators?
Self-Instructional Material 281 Low-level Programming NOTES only bits, whereas the software handles numbers in terms of bytes. These features allow programmers to write low-level programming in C. There are six bit-wise operators as follows: & bit-wise AND | bit-wise OR ~ bit-wise one's complement ^ bit-wise exclusive OR &gt;&gt; left shift &lt;&lt; right shift As in + =, you can also have bit-wise assignment operators. All unary bit-wise operators can be written in this form. In this unit, you have learned about register variables which have similar characteristics as auto variables. The only difference between the two is that while auto variables are stored in the memory, register variables are stored in the register of the CPU. The initial value will be an unpredictable or garbage value. The variables are local to the block and they will be available as long as the blocks are active. The CPU registers respond much faster than the memory, so that the computing time is reduced. Therefore, those variables, which are used frequently, can be declared as register variables. You also learnt that in software, you organize the digits byte-wise, but the hardware handles it bit-wise. Therefore, bit-wise operators are very useful for directly interacting with the hardware. 12.6 KEY TERMS • Binary operators: In C programming it refers to those operators, which operate on 2 operands, i.e., a+b, a%b, etc. • AND operator: This is a binary operator and compares two bits. If both are 1, the output is 1, otherwise zero. • OR operator: This is also a binary operator. The output of OR operator will be 0, if both the inputs are 0, and 1 otherwise. • Bit-wise assignment operators: These operators combine bit-wise operators and assignment operators. The assignment operator is always to the right of the bit-wise operator. 12.7 ANSWERS TO 'CHECK YOUR PROGRESS' 1. Those variables, which are used frequently, can be declared as register variables. 2. Binary operators refer to those operators, which operate on 2 operands, i.e., a + b, a % b, etc. 3. Bit-wise operators are very useful for directly interacting with the hardware. 4. It compares two bits, and if both are 1, the output is 1, otherwise zero. This can be used to compare the sets of bits.

282 Self-Instructional Material Low-level Programming NOTES 5. This is also a binary operator. The output of OR operator will be 0, if both the inputs are 0. 6. These operators combine bit-wise operators and assignment operators. The assignment operator is always to the right of the bit-wise operator. 12.8 QUESTIONS AND EXERCISES Short-Answer Questions 1. Say True or False: (a) && and & have the same meaning in C. (b) The right shift operator has to be followed by an integer. (c) A left shift operator shifts the value to the left of it by the number of times specified on its right. (d) Register variables in different functions can use the same name. 2. What is a register variable? 3. What are the six bit-wise operators? 4. Describe bit-wise assignment operators. Long-Answer Questions 1. Write a program to multiply a number by 2 or 4 depending on the user's choice. 2. Write a program to divide a number by 4 or 8 depending on the user' s choice. 3. Write a program to shift a number by 4 bits by storing the shifted bits to another number. The program should print out the values of the number and the number formed by shifted bits in decimal and binary forms. 4. Write a program to complement the bits starting from r and ending at s of a given number and print the value of the manipulated number in binary, octal and hexadecimal form. 5. Write a program for binary addition with carry of 8-bit numbers. Hint: sum = a ^ b carry = a & b At every bit position, add the corresponding bits of a, b and carry from the addition of previous bit. 6. Explain bit-wise operations with the help of examples. 12.9

FURTHER READING Gottfried, Byron S. Programming with C, 2nd edition. Schaum's Outline Series. New York: McGraw-Hill, 1996. Forouzan, Behrouz A. and Richard F. Gilberg. Computer Science: A Structured Programming Approach Using C, 2nd edition. California: Thomson, Brooks Cole, 2000. Dey, Pradip and Manas Ghosh. Computer Fundamentals and Programming in C. New Delhi: Oxford Higher Education, 2006.

Self-Instructional Material 283 Low-level Programming NOTES Bronson, Gary J. A First Book of ANSI C, 3rd edition. California: Thomson, Brooks Cole, 2000. Kanetkar, Yashwant. Understanding Pointers in C.

New Delhi: BPB Publication, 2001. Kanetkar, Yashwant. Let us C. New Delhi: BPB Publication, 1999.

Kernighan, Brian W. and Dennis Ritchie. C Programming Language, 2nd edition. New Jersey: Prentice Hall, 1988. Foster, W. D. and L. S. Foster. C by Discovery. Boston: Addison-Wesley, 2005. Kanetkar, Yashwant. Working with C. New Delhi: BPB Publication, 2003. Horton, Ivor. Instant C Programming. New Jersey: Wrox Press (John Willey & Sons), 1995. Lawlor, Steven C. The Art of Programming Computer Science with 'C'. New Jersey: West Publishing Company, 1996.

Self-Instructional Material 285 Additional Features of C NOTES UNIT 13 ADDITIONAL FEATURES OF C Structure 13.0 Introduction 13.1 Unit Objectives 13.2 Enumeration 13.3 Command Line Parameters: argc and argv 13.3.1 Creation of a Utility to Search for a given String in a File 13.4 Typedef 13.5 C Preprocessor 13.6 Defining-MACRO 13.7 Conditional Compilation Directives 13.8 Summary 13.9 Key Terms 13.10

Answers to 'Check Your Progress' 13.11 Questions and Exercises 13.12 Further Reading 13.0 INTRODUCTION In this unit, you will learn about

additional features of C programming. Enumeration is one of the basic data types. The keyword for this data type is enum. This is similar to structures and unions. It is always used in conjunction with other data types. You will learn about argc and argv which are called command line arguments. If the program is executed under a DOS environment, i.e., at C&lt; prompt, then you can run the executable file of the program and pass the arguments. The keyword typedef is used to create a new name for the existing data type name. You will learn that the headers used for library functions are the additional facilities provided by 'C' for ease of programming. The library functions are included in the program during 'preprocessing'. The word 'preprocessor' means that processing is carried out before the compilation of the program written by us. Using macros, you can simplify the procedure, as a change carried at the top would be reflected throughout the program. Macros are faster than functions, as they do not require to be called with arguments and return values, unlike functions. You will also learn that a program gets converted into executable code. The program statements you write are the source code, which is made complete during preprocessing, and then compiled. The linker in the system links all files and gives you an executable file. The code in turn is called executable code and stored with file name.EXE extension. 13.1 UNIT OBJECTIVES After going through

this unit, you will be able to: •

Understand the

additional features of C •

Define enumeration • Explain command line parameters • Understand typedef

286 Self-Instructional Material Additional Features of C NOTES • Explain C preprocessor • Define macros • Understand conditional compilation directives 13.2 ENUMERATION This is one of the basic

**97%** **MATCHING BLOCK 88/126** **W**

data types. The keyword for this data type is enum. You can define guardian as follows: enum guardian { father, husband, guardian }; Note that there are no semicolons, but only a comma after the members except the last member. There is not even a comma after the last member. There is no conflict between both guardians. The top one is enum and the bottom one is a member of enum guardian. See the similarity between structure, union and enum. The enum variables can be declared as follows: enum guardian emp1, emp2, emp3; This is similar to structures and unions. The first part is the declaration of the data type. The second part is the declaration of the variable. The initial values can be assigned in a simpler manner as given below: emp1 = husband; emp2 = guardian; emp3 = father;

You

have to assign only those declared as part of the enum declaration. Assigning constants not declared will cause errors. The compiler treats the enumerators given within the braces as constants. The first enumerator father will be treated as 0, the husband as 1 and the guardian as 2. Therefore, it is strictly as per the natural order starting from 0. The enumerated data type is never used alone. It is used in conjunction with other data types. We can write a program using enum and struct.
It is given below: /*
Example 13.1 enum within structure*/ #include &gt;stdio.h&lt; int main() { enum guardian { father, husband, relative }; struct employee
Self-Instructional Material 287
Additional Features of C NOTES {
char *
name; float basic; char *birthdate; enum guardian guard; }emp[2]; int i; emp[0].name="RAM"; emp[0].basic= 20000.00; emp[0].birthdate= "19/11/1948"; emp[0].guard= father; emp[1].name="SITA"; emp[1].basic= 12000.00; emp[1].birthdate= "19/11/1958"; emp[1].guard=husband; for(i=0;i&gt;2;i++) { if( emp[i].basic ==12000) { printf("Name:%s\nbirthdate:%s\nguardian: %d\n", emp[i].name, emp[i]. birthdate,
emp[
i].guard); } } } Result of the program Name : SITA birthdate : 19/11/1958 guardian : 1 The program clearly assigns the relationships between the employee and the guardian; enum guardian is the data type, and guard is a variable of this type. However, when you are printing emp [i]. guard, you are printing an integer. Hence 0, 1 or 2 will only be printed for the status, and this is a limitation.
However,
this can be overcome by modifying the program. The program modified with a switch statement is given below: /*Example 13.2 expanding enum*/ #include &gt;stdio.h&lt; int main() { enum guardian { father, husband, relative }; struct employee { char *name;
288
Self-Instructional Material Additional Features of C NOTES
float basic; char *birthdate; enum guardian guard; }emp[2]; int i; emp[0].name="RAM"; emp[0].basic= 20000.00; emp[0].birthdate= "19/11/1948"; emp[0].guard= father; emp[1].name="SITA"; emp[1].basic= 12000.00; emp[1].birthdate= "19/11/1958"; emp[1].guard=husband; for(i=0;i&gt;2;i++) { if( emp[i].basic ==12000) {printf("Name:%s\nbirthdate:%s\nguardian:", emp[i].name, emp[i]. birthdate); switch(emp[i].guard) { case 0:printf("father\n"); break; case 1:printf("husband\n"); break; case 2:printf("relative\n"); break; } } } } Result of the program Name : SITA birthdate : 19/11/1958 guardian : husband Even though the conversion of an integer to actual name is additional work, enum is an useful construct, since it improves readability in addition to number of other advantages. For example, we can define boolean as follows: enum boolean { false, true }; Here "false" will contain an integer value "0" and true "1". This can be used to assign values for "found" in our sorting programs. We can define found as follows after declaring boolean. enum boolean found;
Self-Instructional Material 289 Additional Features of C NOTES
We have allowed the system to assign integer values to the members of enum, but we can also assign specific values to the various members of enum. When values are assigned, then it takes precedence over what the system assigns. We can define boolean as, enum boolean { yes = 1, no = 0 }; This is similar to defining using #define. The #define equivalent for this will be, #define yes 1 #define no 0 Although #define and enum provide a way to associate symbolic names such as boolean with constants, there are difference between them. The differences are: (
i) enum can
generate values itself unlike #define where you have to specify the replacement constant. (ii) The compilers need not check the validity of what is stored in the enum variable, but the #define replacement constant will be checked for valildity. (iii) It is possible to print out the values of enum variables in symbolic form, but this is not possible with # define. Anyway, either of them can be used depending on the context. 13.3
COMMAND LINE PARAMETERS: argc AND argv These are called command line arguments. We have so far been passing arguments to functions other than the main function. We can also pass arguments to main function, but since this is the first function to be executed, the arguments are to be supplied before calling it. When do we call main? We call main when we execute the program itself. If the program is executed under a DOS environment, i.e., at C&lt; prompt, then we can run the executable file of the program and pass the arguments. For example, we can create prg. exe. When we execute this prg, the main function of prg will be executed. If we want to pass arguments then we can pass it to main along with the command itself. For example, C &lt; prg. exe 100, 200, 300 You type the above mentioned command at the C prompt in DOS shell. The name of file containing the program is to be followed by the arguments or values to be passed to the program, i.e., the main function. In this way, you can pass arguments to the main function in the command line. The main function is always called with two arguments, neither more nor less. The first is called argc and the second argument is called argv. It is the convention to call the command line arguments such as these as argc and argv. The argc stands for argument count. It is an integer type argument and contains the total number of arguments passed. The second is called the argument vector, conventionally called argv, and

| 95% | MATCHING BLOCK 89/126 | SA | The C Programming Language.pdf (D44958811) |
|---|---|---|---|

is a pointer to an array of strings that contain the arguments.

Each argument will be a string. If you want to pass 4 arguments then argc will contain 4 and argv will point to the array of 4 strings, i.e., the address of the first string. argv[0] will always point to the name of the program. In the above example,

290 Self-Instructional Material Additional Features of C NOTES prg. exe will be stored in argv[0]. 100, 200 and 300 will be stored as strings argv [1], argv [2] and argv [3] respectively. If argc = 1, there is only one string, i.e., the program file. Therefore there are no other command line arguments. The following program prints the contents of the argv. For this to happen the program has to be executed in DOS mode and arguments are to be supplied in the command line. An example of
the program is given below: /*Example 13.3*/ /* to demonstrate argc, argv*/ #

| 92% | MATCHING BLOCK 92/126 | SA | 002 CHAPTERS.docx (D19089854) |
|---|---|---|---|

include &gt;stdio.h&lt; main(int argc, char *argv[]) { int j; for (j=0; j&gt;

argc; j++) printf("%s\n", argv[j]); } The names argc and argv are conventional, but any other name could be used as well. Compile the program. When successful, go to DOS Shell. Type the program name, followed by whatever strings you want. It will reproduce whatever was typed in the C prompt. Output of the Program when the file name followed by the arguments. Result of the program D:\CPROG1\BX62.EXE 100 200 300 400 You did not give argc explicitly, and the compiler counted the number of arguments and assigned the value for argc. Do not look for the same type of result, since this will depend on the arguments and file name actually given. This demonstrates the command line arguments passed to the main function at run time.
This can be used to copy a file to another file. Assume that the first named file is to be copied to the second named file. You may write a program and convert it into an executable file, specifying the argument in the DOS command line. You may specify as follows at the C&lt; prompt: C &lt; prgname . exe f1.cpp f2.cpp This means that you want to copy the contents of f1.cpp to f2.cpp. Here, the number of arguments are 3, and therefore argc will contain 3. *argv[0] = prgname.exe *argv[1] = f1.cpp – source to copy from *argv[2] = f2.cpp – file where to be copied A character at a time is to be fetched from f1.cpp and put into f2.cpp.
Self-Instructional Material 291 Additional Features of C NOTES 13.3.1 Creation of a Utility to Search for a given String in a File Let us write a program sfind. The program has to be stored in this name after compilation, and executed in the command line. The argument argv[0] is the name of the file. The string to be searched in the file is given as argv[1]. The file where to be searched is given as argv[2]. Thus there will be 3 arguments as given below: sfind exe swamy ws.doc When this is typed the program should find whether swamy is in ws.doc; if so it should return the string, and if not, say so. The program sfind.exe is given below: /* Example 13.4 - finding given string in file */ #include &gt;stdio.h&lt; #include&gt;string.h&lt; main(int argc, char *argv[]) { char alpha[80]; FILE *fr; fr=fopen(argv[2], "rb"); while ( fgets(alpha,79, fr)!=NULL) { if(strstr(alpha, argv[1])!=NULL) { printf("String %s found", argv[1]); } else puts("string not found"); } fclose (fr); } Result of the program C:\BORLANDC\BIN&lt;sfind.exe Delhi ws.doc String Delhi found The main() is passed the arguments. The file ws.doc is opened. The contents are brought, line after line to the buffer and the matching of strings checked by using the library function strstr. If the result is NULL, the string is not found. If it is not NULL, then the string supplied as argv[1] is found in the file. When this program is available we need not specify the actual names of the string and file to be scanned in the program file. The file name and string name can be typed from the keyboard at run time. Thus this program has become a general purpose program. 13.4 TYPEDEF The keyword typedef is used to create

| 47% | MATCHING BLOCK 90/126 | W |
|---|---|---|

a new data type name. It does not really create a new type, but only gives a new name to an existing type.

For example, typedef long double LD; LD x , y; Check Your Progress 1. What is enumeration? 2. How does the compiler treat the enumerators? 3. Differentiate between enum and #define. 4. Define argc and argv.
292 Self-Instructional Material Additional Features of C NOTES Here, long double is given a new name LD. Wherever LD appears, the compiler will take it as a long double. Also consider the following example, typedef void (FN) (int , int) ; Here FN represents a function with two integer arguments and returns nothing. This can be used for any other function having the same property, i.e., it passes two integer arguments and nothing is returned from the function. Later on in the same program we may declare, FN f1 , f2; This serves as a declarator for f1 and f2, which are of the same type. The program below explains the concept. /*
Example 13.5*/ /*program to demonstrate calling muliple functions*/ #include&gt;stdio.h&lt; int main() { long nummul=0; long num=0, rev=0, add_digit=0; /*good practice to inialize all variables*/ typedef long FN(long ); FN reverse; /*see simplification*/ FN mult; FN sum_digit; printf("enter unsigned number\n"); scanf("%lu", &num); if (num%2) /*remainder 1*/ { rev =reverse(num); printf("number is odd\n"); printf("number entered=%lu\n number reversed=%lu\n", num, rev); } else { nummul=mult(num); printf("number is even\n"); printf("number=%lu\n its multiple=%lu\n", num, nummul); } if (num%3 ==0) { add_digit= sum_digit(num); printf("number evenly divisible by 3\n"); printf("sum of digits =%lu", add_digit); } }
Self-Instructional Material 293 Additional Features of C NOTES

| 64% | MATCHING BLOCK 91/126 | W |
|---|---|---|

long reverse(long n) { long r=0; while (n&lt;0) { r=r*10+(n%10); n=n/10; } return r; } long mult(long p) { long sq; sq=2*p; return sq; } long sum_digit(long num) { long sum=0; while (num &lt;0) { sum=sum+(num%10); num=num/10; } return sum; } Result of the program enter unsigned number 234 number is even number = 234

its multiple = 468 number evenly divisible by 3 sum of digits = 9 In the union guardian declaration we can add the following : typedef union guardian UN; UN u1, u2; Here, UN will be substituted by union guardian at the time of compilation. Thus the usage of typedef helps in reducing typing and considered to improve readability. This can be used to make short hand notations for the existing data types as well as user defined variables such as structures.
294 Self-Instructional Material Additional Features of C NOTES 13.5 C PREPROCESSOR In all the program examples seen so far, certain files were included before the main function. These are the headers for library functions such as scanf, printf, gets, strlen, etc., and are additional facilities provided by 'C' for ease of programming. The library functions are included in the program during 'preprocessing'. The word 'Preprocessor' means that processing is carried out before the compilation of the program written by us. You have quite often used #include and #define. The former includes the contents of files such as &gt;stdio.h&lt; during compilation, while #define replaces the symbolic name with actual values before actual compilation. Thus preprocessing is nothing but the operations or processing carried out before actually processing. Now, see in detail the various preprocessor directives used in 'C'. #Include The files following this statement are to be included in the program. For example, when you use standard library function printf() you include &gt;stdio.h&lt;. The file &gt;stdio.h&lt; contains the declarations or the prototype for the printf() function. Now that you know function and function prototypes it is easy for you to visualize. The library functions may be elsewhere. The header file contains prototypes for one or more library functions. For example, stdio contains prototypes for scanf(), printf(), get char, putchar, etc., with one function corresponding to each one of them. But for the include statement, the compiler would not have recognized printf(). Therefore, the contents of the printf() function are to be included in our program file before actual compilation. You can write this in the following forms: #include &gt;filename&lt; #include "filename" Such lines will be replaced by the contents of the file before compilation. The file may be in the user area or may be in the header file. If the right specifications for the filename are given the system will find the file and include it. If you write the include statement as #include &gt;stdio.h&lt; or #include &gt;prg.cpp&lt;, the system looks for the file only in the specified directories, namely h and cpp respectively. On the other hand, if you specify #include "stdio.h"or #include "prg.cpp", the system

would look for the file in the current directory as well as the specified list of directories.

The statement #include is not only applicable to library functions and their prototypes, but also applicable to programmer developed files. If there is a large program file, it can be divided function-wise and stored in different files. All these files are to be included in the file containing main() by using #include. #include is a directive or preprocessor directive to the compiler. Programmers can also combine all function declarations, definitions, variable declaration, etc. in case of a large program file in a separate file, which can be included in every file containing part of the program. 13.6 DEFINING-MACRO You have used this kind of statement a number of times to increase the flexibility of programming. The syntax is as follows: #define symbolic_name replacement_constant
Self-Instructional Material 295 Additional Features of C NOTES For example, #define MAX 10 Wherever MAX is found, the compiler will replace it as 10. This may also happen before the program is actually compiled, and such statements are also called macros or macro definitions. MAX is called a macro template and 10, its corresponding macro expansion. The rules of macro definition are as follows: (a) No semicolon at the end of the statement. (b) A macro template is usually written in capital letters for ease of identification. (c) No commas in between. The advantages of such macro definitions are clear: (a) No accidental change of constants. (b) If a constant occurs at a number of places like the size of an array, the rate of interest, etc, and if you want to increase the size of the array or in the other case, the rate of interest later, then you would need to makes change at a number of places in the program code. Using macros, simplifies the procedure, as a change carried at the top would be reflected throughout the program. You can use macros for substituting complex statements such as: # define INPUT(a) scanf("%f" , &a); # define OUTPUT printf("Enter a value"); # define AREACYL(r, h) (2*3.14 * r *( r+h)) # define volsp(r) ( 3.14 * r * r * r * 4.0/3.0 ) Note the last two statements carefully. We are passing arguments. For example, AREACYL refers to the area of the cylinder, r is the radius and h is the height. Whenever you want to calculate the area of the cylinder, you can specify the symbolic constant AREACYL which is the macro with the arguments namely radius and height. For example, area = AREACYL(3.0, 2,0); Now AREACYL will be replaced by 2.0 * 3.14 * 3.0 * (3.0 + 2.0). Of course you should have defined area as a float in the function. Thus you have used the macro like a function. Similarly the #define volsp calculates the volume of a sphere of radius r. You can give any radius. It will substitute the r with the given radius. For example, you can call, x = volsp(4.0); Remember that there should not be spaces between the macro name and the argument. Spaces left there, if any, will be treated as the argument and hence as replacement text. Whenever you define a macro in the form of a function, the entire macro should be enclosed within parenthesis. Now look at the example 13.6 given below: /*Example 13.6 - to demonstrate macros*/ #include&gt;stdio.h&lt; #define INPUT(a) scanf("%f" , &a); #define OUTPUT printf("Enter radius of sphere\n"); #define volsp(r) ( 3.14 * r * r * r * 4.0/3.0 ) #define AREACYL(r,h) (2.0*3.14*r*(r+h)) int main() {

296 Self-Instructional Material Additional Features of C NOTES float a, v, r, h; OUTPUT INPUT(a) v=volsp(a); printf("voume of sphere of radius %f = %f\n", a,v); printf("Enter radius and after a space height of cylinder\n"); INPUT(r) INPUT(h) printf("area of cylinder=%f\n", AREACYL(r,h)); } Result of the program Enter radius of sphere 2.0 voume of sphere of radius 2.000000 = 33.493332 Enter radius and after a space height of cylinder 4 5 area of cylinder=226.080000 Four macros have been defined in the program. In the main(), OUTPUT will be replaced by printf("Enter radius of sphere\n"); wherever it is found, at the time of preprocessing, but before compilation. Similarly, the other 3 macros will also be substituted at the time of preprocessing. Parameters are passed in the macros AREACYL(r,h) and volsp(r). This is similar to passing values in a function. The program prints the volume of the sphere and then the area of the cylinder for the dimensions given at run-time. There are, however, differences between macros and functions although a macro resembles a function in the above examples. As you have guessed right, the executable code of a macro based program will be larger than a function-based program if the macro is used more than once. This is due to the reason that macro will be substituted wherever it appears, whereas actual code of the corresponding function will be at one place and not substituted whenever called. Macros are faster than functions, as they do not require to be called with arguments and return values, unlike functions. This calling and return does not arise with macros, since macros are substituted at every place of occurrence. Therefore, depending on the context, a macro or a function could be used, i.e., if speed is the criteria, a macro could to be used, and if program size is the criteria, a function could be used. By now you know how a program gets converted into executable code. The program gets converted to executable code in the following manner. The program statements you write are the source code, which is made complete during preprocessing, and then compiled. These two operations take place when you say compile, and the object code is produced after compilation with a file name.OBJ extension. The linker in the system links all files and gives you an executable file. The code in turn is called executable code and stored with file name.EXE extension. When you say run, the .EXE file is executed. Thus you may have 4 files in the same name with different extensions as follows:

Self-Instructional Material 297 Additional Features of C NOTES name.c name.bak /*back up file*/ name.OBJ name.EXE Whenever you want to execute the program, use the executable file. #undef MACROS 'C' is a flexible language. We have defined certain macros. We can always undefine them at some point later in the program. The syntax is given below: #undef INPUT #undef AREACYL Whenever the compiler comes across #undef directives it will cease to recognize the corresponding macro definition from that point onwards. 13.7 CONDITIONAL COMPILATION DIRECTIVES #if You have read about the execution of statements at run time depending on certain conditions. 'C' allows even conditional compilation. In 'C', a set of statements will be included for compilation, depending on certain conditions. In the former case, conditions were checked at the time of program execution. In the latter case, the conditions are checked at the time of compilation itself. Therefore, the check is carried out during preprocessing. For example, # if (constant integer expression) { s1 } # endif or #elif or # else The group of statements s1 will be included if the constant integer expression is true, i.e., non zero. The statements are those following #if and upto #endif or #elif or #else. Let us assume that the I/O bus of the computer varies and that it could be 1 byte, 2 bytes, 4 bytes and 8 bytes. The following code segment defines the size of short integer and integer depending on the I/O bus. This is a hypothetical example to illustrate the concept. int main() { #if (DATA == 1) SHRT-MAX = + 127; SHRT-MIN = - 128; INT-MAX = + 127; INT-MIN = - 128; #else #if (DATA == 2) SHRT-MAX = + 127; SHRT-MIN = - 128; Check Your Progress 5. Why is the keyword typedef used? 6. How does typedef improve readability? 7. Why are scanf(), printf(), gets() and strlen used? 8. Define preprocessor. 9. What happens when you define #include in your program? 10. Differentiate between #include&gt;stdio.h&lt; and #include "stdio.h"; #include&gt;prg.cpp&lt;and #include"prg.cpp".

298 Self-Instructional Material Additional Features of C NOTES INT-MAX = + 32767; INT-MIN = - 32768; #elif (data == 4)/* Similar to else if */ SHRT-MAX = + 32767; SHRT-MIN = - 32768; INT-MAX = + 2147483647; INT-MIN = - 247483648; #elif ....... #endif } The first group of statements is clearly marked by #if and #else. These will be compiled if DATA == 1. Similarly, the other groups will be compiled depending on the value of DATA. Although in practice, this can be achieved using the general branch constructs, DATA will be only one of the various types for a particular system. Therefore all the blocks need not be compiled. If the irrelevant statements are not included, you can reduce the size of the code, then program execution will be faster. If you had used general branching, the entire code will be compiled, although the other conditions will never be used. Conditional compilation thus helps in improving performance. Since the compiler has to work with varying configurations, conditional compilation provides the flexibility to compile the appropriate statements depending upon the type of hardware. Therefore, a single program will suffice, as otherwise different programs would be required for different configurations. #ifdef There are other types of conditional compilation statements. You define macros with #define and make them inactive using #undef. You can execute certain statements if the macro has been defined using the #ifdef statement. For example, #ifdef INPUT s1; s2; #endif When the compiler goes over to the #ifdef INPUT statement, it will check whether INPUT remains defined. If so, s1 and s2 will be compiled. If not, the compiler will skip the statements. This statement gives flexibility to the programmer when the requirements of the program change frequently. It is also useful when the software is to be supplied for different hardware configurations and therefore software suitable for all configurations must be written. Only one of them will be true in any situation and hence one #ifdef will be true. The corresponding code segment will be compiled. If the same thing is executed through switch case, then the whole code, applicable for all conditions will have to be compiled, whereas here only one segment of code will be compiled. This is the essential difference. The opposite of #ifdef is #ifndef. Here, a set of statements will be compiled if a macro has not been defined. For example, main() # ifndef volsp execute statements for AREACYL Check Your Progress 11. Describe the rules for defining macro. 12. Differentiate between macros and function. 13. What happens when the compiler comes across #undef?

Self-Instructional Material 299 Additional Features of C NOTES #endif #ifndef AREACYL calculate volsp #endif 13.8 SUMMARY In this unit, you have learned that the enumerated data type is a declaration of a new data type with real life name. A group of names can be declared as enumerated data type, as in the case of other basic data types. The names are given within brackets and can be used in assignment statements. Basically, integer constants are associated with the members of the enum. Whenever you assign the actual names, the system assigns the integer constants. Different integer constants can also be specifically assigned by the programmer. Whenever, the program prints out the values, it will print only integers. The integers can in turn be converted to the corresponding names. The enum improves readability of the program. The main function can have arguments passed to it through command line arguments represented by argc and argv. The identifier argc contains the count of the number of command line arguments, while argv[ ] is a pointer to an array of strings. Each string contains the command line argument, which is passed to the main function. argv[0] contains the first argument, argv [1] contains the second argument and so on. If argc = 1, then there are no command line arguments since argv[0] contains the name of the program file to be executed. Since the arguments are to be passed to main from the C&lt; prompt in the DOS mode, the first argument has to be the name of the file with .exe extension. This means the program will be executed first so as to call main and then pass on the other arguments. A program to print command line arguments was developed. typedef creates new names for existing data types. Once a name has been defined using typedef, it can be used throughout the rest of the program. No new data type can be created with typedef. It is only a shorthand notation which helps in reducing the writing or typing but certainly improves understanding of the program and hence improves readability. Once the typedef statement has been understood, then there will be no need to examine the type-defined statements for accuracy elsewhere in the program. You have learned that 'C' programming has a number of features which make it interesting. The various library functions can be included in the programs so as to reduce the coding effort as well as to improve the quality of programs. However, the compiler should know the library functions used, and therefore, the declarations of the functions are made known so that through the declarations in the included file the entire function can be called at the time of compiling the program. The header files are included through the #include statement. When the file name is indicated like &gt;stdio.h&lt;, the system looks for the file in the current directory. On the other hand, if indicated within quotes like 'stdio.h', the system looks for the file in the current directory, as well as in the other specified directories. Further, you have learned that whenever some segments of code or variable names are repeatedly used in a program, these could be defined as macros. The macro name (instead of the whole code) can be given at locations when these are required. 300 Self-Instructional Material Additional Features of C NOTES This eliminates errors due to wrong typing and improves readability. Even multi-line programs can be treated as macros and arguments passed as in a function. The essential difference between a macro and a function are: • A macro will be substituted at all places increasing the code size at the time of execution. • A function will be included only once, keeping the code size to the minimum. • A macro will facilitate faster execution whereas a function can be slow. 'C' also provides for compilation of specific code segments depending on conditions. This is achieved through the #if, #endif, #elif and #else statements. Conditional compilation can also be carried out depending on whether a particular macro has been defined or not. Thus, in 'C' programming, there is a stage in between coding and compilation, which is known as preprocessing. At this stage, the macros have to be substituted, the included files actually brought to the source code file, and then the entire code presented to the 'C' language compiler. Even in compilation, you can have flexibility through the use of conditional compilation directives. Thus, the same version of the software can be executed in varying hardware configurations through the use of conditional compilation statements. 13.9 KEY TERMS • Enumeration: This is one of the basic data types. The keyword for this data type is enum. It is similar to structures and unions. It is always used in conjunction with other data types. • argc: It stands for argument count. It is an integer type argument and contains the total number of arguments passed. • argv: It is called argument vector and

| 95% | MATCHING BLOCK 94/126 | SA | The C Programming Language.pdf (D44958811) |
|---|---|---|---|

is a pointer to an array of strings that contain the arguments.

Each argument will be a string. • typedef: It is used to create a new name for the existing data type. • Macros: These simplify the programming procedure, as a change carried at the top would be reflected throughout the program. Macros are faster than functions, as they do not require to be called with arguments and return values. 13.10 ANSWERS TO 'CHECK YOUR PROGRESS' 1. This is one of the basic data types. The keyword for this data type is enum. 2.
The compiler treats the enumerators given within the braces as constants. 3. (
i)
enum can
generate values itself unlike #define where you have to specify the replacement constant. (ii) The compilers need not check the validity of what is stored in the enum variable, but the #define replacement constant will be checked for valildity. (iii) It is possible to print out the values of enum variables in symbolic form, but this is not possible with # define. Anyway, either of them can be used depending on the context.
Self-Instructional Material 301 Additional Features of C NOTES 4. These are called command line arguments. The first is called argc and the second argument is called argv. It is the convention to call the command line arguments such as these as argc and argv. The argc stands for argument count. It is an integer type argument and contains the total number of arguments passed. The second is called the argument vector, conventionally called argv, and

| 95% | MATCHING BLOCK 96/126 | SA | The C Programming Language.pdf (D44958811) |
|---|---|---|---|

is a pointer to an array of strings that contain the arguments.

Each argument will be a string. 5. The keyword typedef is used to create

| 47% | MATCHING BLOCK 95/126 | W | |
|---|---|---|---|

a new data type name. It does not really create a new type, but only gives a new name to an existing type. 6.

Thus, the usage of typedef helps in reducing typing and is considered to improve readability. This can be used to make short hand notations for the existing data types as well as user-defined variables such as structures. 7. These are the headers for library functions such as scanf, printf, gets, strlen, etc., and are additional facilities provided by 'C' for ease of programming. The library functions are included in the program during 'preprocessing'. 8. 'Preprocessor' means that processing is carried out before the compilation of the program written by us. We have quite often used #include and #define. The former includes the contents of files such as &gt;stdio.h&lt; during compilation, while #define replaces the symbolic name with actual values before actual compilation. Thus preprocessing is nothing but the operations or processing carried out before actually processing. 9. The files following this statement are to be included in the program. For example, when we use standard library function printf() we include &gt;stdio.h&lt;. The file &gt;stdio.h&lt; contains the declarations or the prototype for the printf() function. 10. If we write the include statement as #include &gt;stdio.h&lt; or #include &gt;prg.cpp&lt;, the system looks for the file only in the specified directories, namely h and cpp respectively. On the other hand, if we specify #include "stdio.h"or #include "prg.cpp", the system

| 94% | MATCHING BLOCK 97/126 | SA | C_book_AmrutaJog_ShrutiJathar.pdf (D54111295) |
|---|---|---|---|

would look for the file in the current directory as well as the specified list of directories. 11. (

a) No semicolon at the end of the statement. (b) A macro template is usually written in capital letters for ease of identification. (c) No commas in between. 12. Macros are faster than functions, as they do not require to be called with arguments and return values, unlike functions. This calling and return does not arise with macros, since macros are substituted at every place of occurrence. Therefore, depending on the context, a macro or a function could be used, i.e., if speed is the criteria, a macro could to be used, and if program size is the criteria, a function could be used. 13. Whenever the compiler comes across #undef directives it will cease to recognize the corresponding macro definition from that point onwards.
302 Self-Instructional Material Additional Features of C NOTES 13.11
QUESTIONS AND EXERCISES Short-Answer Questions 1. State whether True or False, giving reasons: (a)
A union can be assigned to another. (b) Assignment of a value to a member of a union cancels the previous assignment to another member. (c) enum is another basic data type. (d) typedef defines new data types like enum. (e) Macros delay execution as compared to functions. (f) Conditional compilation means conditional execution. (g) The &gt;*.h&lt; files are searched in all directories. (h) #endif can be used alongwith if. (i) #if can only act on macros. (j) #inlcude is applicable only to library functions. Long-Answer Questions 1. Enumerate the difference between: (a) enum and #define (b) enum and typedef 2. Describe the following with the help of example programs: (a) Preprocessing (b) Conditional compilation (c) Macros vs Functions 3. Write a program for the following: (a) A macro to find the roots of the quadratic equation? (b) A macro to find the smallest of three given numbers. (c) A macro for finding out the area of a triangle. (d) Conditional compilation for finding out the area of a rectangle or the area of a square. (e) To provide conditional compilation for one of the following when called: (i) Divisible by 3 (ii) Divisible by 5 (f) The main function depending upon the request should find whether a given number is divisible by 2 or 3 or 5. (g) Rewrite the above program with functions. Compare the number of executable statements between the two versions. 13.12
FURTHER READING Gottfried, Byron S. Programming with C, 2nd edition. Schaum's Outline Series. New York: McGraw-Hill, 1996. Forouzan, Behrouz A. and Richard F. Gilberg. Computer Science: A Structured
Self-Instructional Material 303 Additional Features of C NOTES Programming Approach Using C, 2nd edition. California: Thomson, Brooks Cole, 2000. Dey, Pradip and Manas Ghosh. Computer Fundamentals and Programming in C. New Delhi: Oxford Higher Education, 2006. Bronson, Gary J. A First Book of ANSI C, 3rd edition. California: Thomson, Brooks Cole, 2000. Kanetkar, Yashwant. Understanding Pointers in C.
New Delhi: BPB Publication, 2001. Kanetkar, Yashwant. Let us C. New Delhi: BPB Publication, 1999.
Kernighan, Brian W. and Dennis Ritchie. C Programming Language, 2nd edition. New Jersey: Prentice Hall, 1988. Foster, W. D. and L. S. Foster. C by Discovery. Boston: Addison-Wesley, 2005. Kanetkar, Yashwant. Working with C. New Delhi: BPB Publication, 2003. Horton, Ivor. Instant C Programming. New Jersey: Wrox Press (John Willey & Sons), 1995. Lawlor, Steven C. The Art of Programming Computer Science with 'C'. New Jersey: West Publishing Company, 1996.

Self-Instructional Material 305 Annexures NOTES ANNEXURES Annexure 1 ASCII Values of Characters ASCII Value Character ASCII Value Character ASCII Value Character ASCII Value Character 000 NUL 032 Blank 064 @ 096 , 001 SOH 033 ! 065 A 097 a 002 STX 034 " 066 B 098 b 003 ETX 035 # 067 C 099 c 004 EOT 036 $ 068 D 100 d 005 ENQ 037 % 069 E 101 e 006 ACK 038 & 070 F 102 f 007 BEL 039 ' 071 G 103 g 008 BS 040 ( 072 H 104 h 009 HT 041 ) 073 I 105 i 010 LF 042 * 074 J 106 j 011 VT 043 + 075 K 107 k 012 FF 044 , 076 L 108 l 013 CR 045 _ 077 M 109 m 014 SO 046 . 078 N 110 n 015 SI 047 / 079 O 111 o 016 DLE 048 0 080 P 112 p 017 DC1 049 1 081 Q 113 q 018 DC2 050 2 082 R 114 r 019 DC3 051 3 083 S 115 s 020 DC4 052 4 084 T 116 t 021 NAK 053 5 085 U 117 u 022 SYN 054 6 086 V 118 v 023 ETB 055 7 087 W 119 w 024 CAN 056 8 088 X 120 x 025 EM 57 9 089 Y 121 y 026 SUB 058 : 090 Z 122 z 027 ESC 059 ; 091 [ 123 { 028 FS 060 &gt; 092 \ 124 | 029 GS 061 = 093 ] 125 } 030 RS 062 &lt; 094 ^ 126 ~ 031 US 063 ? 095 - 127 DEL Note: The first 32 characters and the last character are control characters; they cannot be printed.
306 Self-Instructional Material Annexures NOTES Annexure 2
Operator Precedence
Operator Associativity ( ) [ ] –&lt; . (dot)

| 85% | MATCHING BLOCK 98/126 | SA | C_book_AmrutaJog_ShrutiJathar.pdf (D54111295) |
|---|---|---|---|

Left to right ! ~ ++ — (unary) + - * &(address) sizeof Right to left * / % (modulus) Left to right (binary)+ - (subtract) Left to right &gt;&gt; &lt;&lt; Left to right &gt; &gt;= &lt; &lt;= Left to right = = != Left to right & (bitwise and) Left to right ^ Left to right | Left to right && Left to right || Left to right ?: Right to left = += -= *= /= %= &= ^= |= &gt; &gt;= &lt; &lt;= Right to left , (comma) Left to right

Self-Instructional Material 307
Annexures NOTES Annexure 3 Summary of Library Functions

| 100% | MATCHING BLOCK 99/126 | SA | The C Programming Language.pdf (D44958811) |
|---|---|---|---|

int fflush(FILE *stream) On an output stream, fflush causes any buffered but unwritten data to be written;

| 84% | MATCHING BLOCK 100/126 | SA | The C Programming Language.pdf (D44958811) |
|---|---|---|---|

fflush(NULL) flushes all output streams. int fclose(FILE *stream) flushes any unwritten data for stream, discards any unread buffered input, frees any automatically allocated buffer, then closes the stream. int remove(const char *filename) removes the named

filel.

| 98% | MATCHING BLOCK 101/126 | SA | The C Programming Language.pdf (D44958811) |
|---|---|---|---|

It returns non-zero if the attempt fails. int rename(const char *oldname, const char *newname) changes the name of a file; it returns non-zero if the attempt fails.

| 98% | MATCHING BLOCK 102/126 | SA | The C Programming Language.pdf (D44958811) |
|---|---|---|---|

CHARACTER INPUT AND OUTPUT FUNCTIONS int fgetc(FILE *stream) returns the next character of stream as an unsigned char(converted to an int), or EOF if end of file or error occurs. char *fgets(char *s, int n, FILE *stream) reads at most the next n-1 characters into the array s, stopping if a newline is encountered; the newline is included in the array, which is terminated by '\0'..

| 89% | MATCHING BLOCK 103/126 | SA | The C Programming Language.pdf (D44958811) |
|---|---|---|---|

int fputc(int c, FILE *stream) writes the character c on stream. It returns the character written, or EOF for error. int fputs(const char *s, FILE *stream) writes the string s

on stream

| 98% | MATCHING BLOCK 104/126 | SA | The C Programming Language.pdf (D44958811) |
|---|---|---|---|

int getchar(void) getchar is equivalent to getc(stdin). char *gets(char *s) reads the next input line into the array s; it replaces the terminating newline with '\0'.

| 71% | MATCHING BLOCK 105/126 | SA | The C Programming Language.pdf (D44958811) |
|---|---|---|---|

int putchar(int c) Putchar(c) is equivalent to putc(c,stdout). int puts(const char *s) writes the string s and a newline to stdout. int ungetc(int c, FILE *stream) pushes c back onto stream. Only one character of pushback per stream is guaranteed.

| 100% | MATCHING BLOCK 106/126 | SA | The C Programming Language.pdf (D44958811) |
|---|---|---|---|

FILE POSITIONING FUNCTIONS int fseek(FILE *stream, long offset, int origin) fseek sets the file position for stream;

void rewind(FILE *stream) moves the file pointer to the first character in the file.
308 Self-Instructional Material Annexures NOTES

| 100% | MATCHING BLOCK 107/126 | SA | The C Programming Language.pdf (D44958811) |
|---|---|---|---|

CHARACTER CLASS TESTS: &gt; CTYPE.H&lt; The header &gt;ctype.h&lt; declares functions for testing characters.

| 84% | MATCHING BLOCK 108/126 | SA | The C Programming Language.pdf (D44958811) |
|---|---|---|---|

The function return non-zero (true) if the argument c satisfies the condition described, and zero if not. isalnum(c) is c alphanumeric? iscntrl(c) control character isgraph(c) printing character except space islower(c) lower-case letter isprint(c) printing character including space ispunct(c) printing character except space or letter or digit isspace(c) space, formfeed, newline, carriage return, tab, vertical tab isupper(c) upper-case letter isxdigit(c) hexadecimal digit

isalpha(c) alphabets isdigit(c) decimal digit

| 51% | MATCHING BLOCK 109/126 | SA | The C Programming Language.pdf (D44958811) |
|---|---|---|---|

In addition, there are two functions that convert the case of letters: int tolower(int c) convert c to lower case int toupper(int c) convert c to upper case If c is an upper-case letter, tolower(c) returns the corresponding lower-case letter; otherwise it returns c.

If c is a

| 82% | MATCHING BLOCK 110/126 | SA | The C Programming Language.pdf (D44958811) |
|---|---|---|---|

lower-case letter, toupper(c) returns the corresponding upper-case letter; oth- erwise it returns c. STRING FUNCTIONS: &gt;STRING.H&lt;

| 97% | MATCHING BLOCK 111/126 | SA | The C Programming Language.pdf (D44958811) |
|---|---|---|---|

char * strcpy (s, ct) copy string ct to string s, including '\0'; return s. char *strncpy(s,ct,n) copy at most n characters of string ct to s; return s. Pad with'\0's if it has fewer than n characters. char *strcat(s, ct) concatenate string ct to end of string s; return s.

| 100% | MATCHING BLOCK 112/126 | SA | The C Programming Language.pdf (D44958811) |
|---|---|---|---|

int strcmp(cs, ct) compare string cs to string ct; return &gt; 0 if cs &gt; ct, 0 if cs==ct, or &lt;0 if cs&lt;ct.

char *strchr(cs, c)
return
pointer to the

| 100% | MATCHING BLOCK 113/126 | SA | The C Programming Language.pdf (D44958811) |
|---|---|---|---|

first occurrence of c in cs or NULL if not present. char *strrchr(cs,c) return pointer to

the last

| 82% | MATCHING BLOCK 114/126 | SA | The C Programming Language.pdf (D44958811) |
|---|---|---|---|

occurrence of c in cs or NULL if not present. char *strstr(cs, ct) return pointer to

the

| 100% | MATCHING BLOCK 115/126 | SA | The C Programming Language.pdf (D44958811) |
|---|---|---|---|

first occurrence of string ct in cs, or NULL if not present. size_t strlen(cs) return length of cs.

| 100% | MATCHING BLOCK 116/126 | SA | The C Programming Language.pdf (D44958811) |
|---|---|---|---|

MATHEMATICAL FUNCTIONS: &gt;MATH.H&lt; The header &gt;math.h&lt; declares mathematical functions and macros.

| 66% | MATCHING BLOCK 117/126 | SA | The C Programming Language.pdf (D44958811) |
|---|---|---|---|

Angles for trigonometric functions are expressed in radians. sin(x) cos(x) tan(x) asin(x) acos(x)

atan(x)
Self-Instructional Material 309 Annexures NOTES sinh(x) cosh(x) tanh(

| 100% | MATCHING BLOCK 118/126 | SA | The C Programming Language.pdf (D44958811) |
|---|---|---|---|

x) exp(x) exponential function e x log(x) natural logarithm ln(x), x&lt;0 log10(x) base 10 logarithm log10(x), x&lt;0 pow(x,y) x y . .

sqrt(x) x , fabs(x) absolute value |x|

| 100% | MATCHING BLOCK 119/126 | SA | The C Programming Language.pdf (D44958811) |
|---|---|---|---|

UTILITY FUNCTIONS: &gt;STDLIB.H&lt; The header &gt;stdlib.h&lt; declares functions for number conversion, storage allocation, and similar tasks.

void *calloc(size_t nobj, size_t size) void *malloc(size_t size)

| 66% | MATCHING BLOCK 120/126 | SA | The C Programming Language.pdf (D44958811) |
|---|---|---|---|

void free(void *p) free deallocates the space to p; p is a pointer to

the space allocated by calloc, malloc. void exit(int status) exit causes normal program termination.

| 100% | MATCHING BLOCK 121/126 | SA | The C Programming Language.pdf (D44958811) |
|---|---|---|---|

How status is returned to the environment is implementation-dependent, but zero is taken as successful termination.

| 100% | MATCHING BLOCK 122/126 | SA | The C Programming Language.pdf (D44958811) |
|---|---|---|---|

int abs(int n) abs returns the absolute value of its int argument. long labs(long n) labs returns the absolute value of its long argument.

| 100% | MATCHING BLOCK 123/126 | SA | The C Programming Language.pdf (D44958811) |
|---|---|---|---|

DATE AND TIME FUNCTIONS: &gt;TIME.H&lt; The header &gt;time.h&lt; declares types and functions for manipulating date and time.

| 100% | MATCHING BLOCK 124/126 | SA | The C Programming Language.pdf (D44958811) |
|---|---|---|---|

int tm_hour; hours since midnight int tm_mday; day of the month int tm_mon; months since January int tm_year; years since 1900 int tm_wday; days since Sunday int tm_yday; days since January 1

| 100% | MATCHING BLOCK 125/126 | SA | The C Programming Language.pdf (D44958811) |
|---|---|---|---|

clock_t clock(void) Clock returns the processor time used by the program since the beginning of execution,

| 76% | MATCHING BLOCK 126/126 | SA | The C Programming Language.pdf (D44958811) |
|---|---|---|---|

time_t time(time_t *tp) Time returns the current calendar time or -1 if the time is not available. double difftime(time_t time2, time_t time1) difftime returns time2-time1 expressed in seconds.

## Hit and source - focused comparison, Side by Side

| | |
|---|---|
| **Submitted text** | As student entered the text in the submitted document. |
| **Matching text** | As the text appears in the source. |

| 1/126 | SUBMITTED TEXT 15 WORDS | 96% MATCHING TEXT 15 WORDS |
|---|---|---|
| | flow lines have arrows to indicate the direction of flow of control between the boxes. The | Flow lines have arrows to indicate the direction of the flow of control between the boxes. ?The |

W http://referenceglobe.com/kpsslp/support/upload_videos/programming%20for%20problem%20solving_1585 ...

| 2/126 | SUBMITTED TEXT 11 WORDS | 83% MATCHING TEXT 11 WORDS |
|---|---|---|
| | Programming languages are classified into machine language, assembly language, high-level language, | |

SA BCSDSC_1-BASIC CONCEPTS OF COMPUTER & C PROGRAMMING_plagarisum check-FINAL.pdf (D139109702)

| 3/126 | SUBMITTED TEXT 27 WORDS | 89% MATCHING TEXT 27 WORDS |
|---|---|---|
| | C Keywords auto break case char const continue default do double else enum extern float for goto if int long register return short signed sizeof static struct switch typedef union unsigned void volatile while 2.5.4 | C keywords is given auto break case char const 22 continue default do double else enum extern float for goto if int long register return short signed size of static struct switch typedef union unsigned void volatile while |

W https://www.iare.ac.in/sites/default/files/AERO_PROGRAMMING_FOR_PROBLEM_SOLVING_LECTURE_NOTES.pdf

| 4/126 | SUBMITTED TEXT 95 WORDS | 100% MATCHING TEXT 95 WORDS |
|---|---|---|
| | The following are the examples of valid and invalid integers: Valid integers +345 /* integer */ 345 /* integer */ −345 /* integer */ 729u /* unsigned integer */ 729U /* unsigned integer */ −112345L /* Long integer */ 112345UL /* Unsigned Long integer */ +112345l /* Long integer */ 112345l /* Long integer - if no sign precedes, it is a positive number */ Invalid integers 345.0 /* decimal point not allowed */ 112, 345L /* no comma allowed */ 112 345UL /* = blank not allowed */ 112890345L /* exceeds the maximum */ +112 345UL /* unsigned cannot have + */ (345l /* ( not allowed */ −345s /* illegal characters */ | The following are the examples of valid and invalid integers: Valid integers +345 /* integer */ 345 /* integer */ −345 /* integer */ 729u /* unsigned integer */ 729U /* unsigned integer */ −112345L /* Long integer */ 112345UL /* Unsigned Long integer */ +112345l /* Long integer */ 112345l /* Long integer - if no sign precedes, it is a positive number */ Invalid integers 345.0 /* decimal point not allowed */ 112, 345L /* no comma allowed */ 112 345UL /* = blank not allowed */ 112890345L /* exceeds the maximum */ +112 345UL /* unsigned cannot have + */ (345l /* ( not allowed */ −345s /* illegal characters */ |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 5/126 | SUBMITTED TEXT 56 WORDS | 99% MATCHING TEXT 56 WORDS |
|---|---|---|
| | fractional form. It must have a decimal point. It could be either positive or negative. As usual, the default sign is positive. No commas or blanks or special characters are allowed in between. The following are the examples of valid and invalid float types: Valid floats 144.00 226.012 Invalid floats +144 /* no decimal point */ 1,44.0 /* comma not allowed */ 42 Self-Instructional Material | fractional form. It must have a decimal point. It could be either positive or negative. As usual the default sign is positive. No commas or blanks or special characters are allowed in between. The following are the examples of valid and invalid float types: Valid floats 144.00 226.012 Invalid floats +144 /* no decimal point */ 1,44.0 /* comma not allowed */ NOTES Self-Material 25 |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 6/126 | SUBMITTED TEXT 70 WORDS | 89% MATCHING TEXT 70 WORDS |
|---|---|---|
| | a subscript of n – 1, where n is the size of the array. 2.10.1 Array Declaration Assuming that there are 40 employees in an office and you want to store their ages, you would have to create 40 variables of type integer or float and store their ages. While such a definition seems alright, it would cause complications while programming. Instead, this can be declared as an array of size 40. For example, int emp_age [40]; Here, | a subscript of n–1, where n is the size of the array. Self-Instructional 96 Material 6.2.1 Array Declaration Assuming that there are 40 employees in an office and we want to store their ages, we would have to create 40 variables of type integer or float and store their ages. While such a definition seems alright, it would cause complications while programming. Instead, this can be declared as an array of size 40. For instance, short emp_age [40]; Here, |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 7/126 | SUBMITTED TEXT 13 WORDS | 89% MATCHING TEXT 13 WORDS |
|---|---|---|
| | can be arrays of integers, arrays of characters and arrays of floating-point numbers. | can arrays of integers, arrays of characters and arrays of floating point numbers, |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 8/126 | SUBMITTED TEXT 8 WORDS | 100% MATCHING TEXT 8 WORDS |
|---|---|---|
| | Operators and Expressions NOTES Self-Instructional Material 57 Operators and Expressions NOTES | Operators and Expressions NOTES Self-Material 39 Operators and Expressions NOTES ( |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 9/126 | SUBMITTED TEXT 7 WORDS | 100% MATCHING TEXT 7 WORDS |
|---|---|---|
| | precedence and associativity? 60 Self-Instructional Material Operators and Expressions NOTES | precedence and associativity Self-Material 35 Operators and Expressions NOTES |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 10/126 | SUBMITTED TEXT 141 WORDS | 96% MATCHING TEXT 141 WORDS |
|---|---|---|
| | OPERATOR EXAMPLE READ AS &gt; less than a &gt; b Is a &gt; b &lt; greater than a &lt; b Is a &lt; b &gt;= less than or equal to a &gt;= b Is a &gt; or = b &lt;= greater than or equal to a &lt;= b Is a &lt; or = b == equal to a == b Is a equal to b != not equal to a != b Is a not equal to b Notice that for checking equality the double equal sign is used, which is different from other programming languages. The statement a = b assigns the value of b to a. For example, if b = 5, then a is also assigned the value of 5. The statement a == b checks whether a is equal to b or not. If they are equal, the output will be true; otherwise, it will be false. | Operator Example Read as &gt; less than a &gt; b Is a &gt; b &lt; greater than a &lt; b Is a &lt; b &gt;= less than or equal to a &gt;= b Is a &gt; or = b &lt;= greater than or equal to a &lt;= b Is a &lt; or = b == equal to a == b Is a equal to b != not equal to a != b Is a not equal to b Self-Instructional Material 37 Operators and Expressions NOTES Note that for checking equality the double equal sign is used, which is different from other programming languages. The statement a = b assigns the value of b to a. For example, if b = 5, then a is also assigned the value of 5. The statement a == b checks whether a is equal to b or not. If they are equal, the output will be true; otherwise, it will be false. |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 11/126 | SUBMITTED TEXT 19 WORDS | 65% MATCHING TEXT 19 WORDS |
|---|---|---|
| | char signed char 1 unsigned char 1 int short &gt; = 2 unsigned int 2 long 4 long unsigned 4 float double 8 long double 10 | char signed char int unsigned int (unsigned ) long int (long) unsigned long int (unsigned long ) float double long double |

W https://cpp4rstarters.wordpress.com/2012/01/05/4/

| 12/126 | SUBMITTED TEXT 256 WORDS | 98% MATCHING TEXT 256 WORDS |
|---|---|---|
| | Now look at their precedence from Annexure 2. &lt; &lt;= &gt; &gt;= have precedence over == != Note that arithmetic operators + – * / have precedence over the relational and logical operators. Therefore, in the following statement: (x – 3 &lt; 5) x – 3 will be evaluated first and only then the relation will be checked. Therefore, there is no need to enclose x – 3 within parenthesis as in ((x – 3) &lt; 5). 3.4.2 Logical Operators You can use logical operators in programs. These logical operators are: && denoting logical And \|\| denoting logical Or ! denoting logical Negation The relational and logical operators are evaluated to check whether they are true or false. 'True' is represented as 1 and 'False' is represented as 0. It is also by convention that any non-zero value is considered as 1 (true) and zero value is considered as 0 (false). For example, if (a – 3) {s1} else {s2} If a is 5, then s1 will be executed. If a = 3, then s2 will be executed. If a = –5, s1 will still be executed. To summarize, the relational and logical operators are used to take decisions based on the value of an expression. 3.4.3 Assignment Operators Assignment operators are written as follows: identifier = expression; For example, i = 3; Note: 3 is an expression const A = 3; 'C' allows multiple assignments in the following form: identifier 1 = identifier 2 = .....= expression. For example, a = b = z = 25; However, you should know the difference between an assignment operator and an equality operator. In other languages, both are represented by =. In 'C' the equality operator is expressed as = = and assignment as =. 62 | Now look at their precedence from Table 3.3. &lt; &lt;= &gt; &gt;= have precedence over == != Note that arithmetic operators + – * / have precedence over the relational and logical operators. Therefore, in the following statement: (x – 3 &lt; 5) x – 3 will be evaluated first and only then the relation will be checked. Therefore, there is no need to enclose x – 3 within parenthesis as in ((x – 3) &lt; 5. 3.2.3 Logical Operators You can use logical operators in programs. These logical operators are: && denoting logical And \|\| denoting logical Or ! denoting logical Negation The relational and logical operators are evaluated to check whether they are true or false. 'True' is represented as 1 and 'False' is represented as 0. It is also by convention that any non-zero value is considered as 1 (true) and zero value is considered as 0 (false). For example, if (a – 3) {s1} else {s2} If a is 5, then s1 will be executed. If a = 3, then s2 will be executed. If a = –5, s1 will still be executed. To summarize, the relational and logical operators are used to take decisions based on the value of an expression. 3.2.4 Assignment Operators Assignment operators are written as follows: identifier = expression; Self-Instructional 38 Material For example, i = 3; Note: 3 is an expression const A = 3; 'C' allows multiple assignments in the following form: identifier 1 = identifier 2 = ..... = expression. For example, a = b = z = 25; However, you should know the difference between an assignment operator and an equality operator. In other languages, both are represented by =. In 'C' the equality operator is expressed as = = and assignment as =. |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 13/126 | SUBMITTED TEXT | 15 WORDS | 89% | MATCHING TEXT | 15 WORDS |
|---|---|---|---|---|---|

If one of the operands is a long double, the other will be converted

SA C book final copy (1).doc (D31388791)

| 14/126 | SUBMITTED TEXT | 15 WORDS | 90% | MATCHING TEXT | 15 WORDS |
|---|---|---|---|---|---|

long double. • else If one of the operands is a double, the other will be converted

SA C book final copy (1).doc (D31388791)

| 15/126 | SUBMITTED TEXT | 15 WORDS | 90% | MATCHING TEXT | 15 WORDS |
|---|---|---|---|---|---|

double. • else If one of the operands is a float, the other will be converted

SA C book final copy (1).doc (D31388791)

| 16/126 | SUBMITTED TEXT | 22 WORDS | 92% | MATCHING TEXT | 22 WORDS |
|---|---|---|---|---|---|

| Submitted | Matching |
|---|---|
| In addition, more than one operator may have the same precedence. For instance * and / have the same precedence. To avoid | In addition, more than one operator may have the same precedence. For example, * and / have the same precedence. To avoid |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 17/126 | SUBMITTED TEXT 200 WORDS | 96% MATCHING TEXT 200 WORDS |
|---|---|---|
| | are learning ‘C’ through the PC, you will get the output through the video monitor. Just as there are library functions for input, there are standard library functions for output as well. They are as follows: • printf() • putchar() • putch() • puts() Giving input and getting output are achieved by using the standard library functions. Note that all the function names are followed by parentheses. The parentheses are meant for passing arguments or getting arguments. The arguments may be absent in certain functions as in the case of main(). However, function names must be followed by parentheses to indicate to the compiler that they are functions. Arguments can be either variables or constants. 4.2.1 Use of printf() Initially, formatted input/output statements will be discussed. Here, you must categorically specify the data type of variables to be read or written and their formats. The printf() and scanf() are formatted I/O statements. The printf() statement can be programmed to give the output in the desired manner. Example 4.1 is given below to illustrate the use of the printf() function in printing the required values. /*Example 4.1*/ /* to demonstrate the print function*/ #include &gt;stdio.h&lt; main() { printf(“welcome to more serious programming\n”); } Here, the first statement after the comment line directs the compiler to include the standard input/output header file. Note that in the printf() statement whatever | are learning ‘C’ through the PC, we will get the output through the video monitor. Just as there are library functions for input, there are standard library functions for output as well. They are given as follows: • printf() • putchar() • putch() • puts() Giving input and getting output are achieved by using the standard library functions. Note that all the function names are followed by parentheses. The parentheses are meant for passing arguments or getting arguments. The arguments may be absent in certain functions as in the case of main(). However, function names must be followed by parentheses to indicate to the compiler that they are functions. Arguments can be either variables or constants. 4.2.1 Use of printf() Initially, formatted input/output statements will be discussed. Here, one must categorically specify the data type of variables to be read or written and their formats. The printf() and scanf() are formatted input and output statements. The printf() statement can be programmed to give the output in the desired manner. Example 4.1 is given below to illustrate the use of the printf() function in printing the required values. /*Example 4.1*/ /* To demonstrate the print function*/ #include &gt;stdio.h&lt; main() { printf(“welcome to more serious programming\n”); } Self-Instructional 50 Material Here the first statement after the comment line directs the compiler to include the standard input/output header file. Note that in the printf() statement whatever |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 18/126 | SUBMITTED TEXT 11 WORDS | 75% MATCHING TEXT 11 WORDS |
|---|---|---|
| | include &gt;stdio.h&lt; int main() { int a, b, sum; a=10; b=20; sum=a+b; printf(“a=%d b=%d sum=%d\n”, a, b, | include&gt;stdio.h&lt; int main() { x,a,b,c; a = 2; b = 4; c = 5; a-- + b++ - ++printf("a=%d, b=%d, c=%d\n",a,b, |

W http://referenceglobe.com/kpsslp/support/upload_videos/programming%20for%20problem%20solving_1585 ...

| 19/126 | SUBMITTED TEXT 18 WORDS | 43% MATCHING TEXT 18 WORDS |
|---|---|---|
| | c’) ; } (b) #include &gt;stdio.h&lt; int main() { unsigned a,b,c; a=10; b=5; c= (a*a)+(b*b)+(2*a*b); printf(“Square of %d + %d is equal to %d ”,a,b,c); } 86 | c=6 x: 0 #include&gt;stdio.h&lt; int main() { int a,b,c; a = 2; b = 4; c = 5; a-- + b++ - ++printf("a=%b=% d, c=%d\n",a,b, c); |

W http://referenceglobe.com/kpsslp/support/upload_videos/programming%20for%20problem%20solving_1585 ...

| 20/126 | SUBMITTED TEXT 288 WORDS | 92% MATCHING TEXT 288 WORDS |
|---|---|---|
| | did not get any message when the numbers were unequal and this can be avoided by using the else statement. The usage of if .. else is shown below. if (condition true) { statements s1 } else { statements s2 } statements s3; The statement else is always associated with an if. If the condition is true, then statements s1 will be executed. After executing them, the program will skip the else block and control goes to statement s3 that follows the else block. If the condition is false, then the statements in the else block, i.e., s2 will be executed followed by statement s3. Statements s1 will not be executed at all when the condition becomes false. The usage of braces clearly brings out which statements belong to the if block and which to the else block. Example 5.2 brings out the usage of if... else. /*Example 5.2 This program demonstrates use of if.. else*/ #include &gt;stdio.h&lt; main() { 90 Self-Instructional Material Control Statements NOTES unsigned int a,b; printf ("enter two integers\n"); scanf("%u%u", &a, &b); if (a==b) { printf("you typed equal numbers\n"); } else { printf("numbers not equal\n"); } } The output of the program when unequal numbers were keyed in is as follows. Result of the program enter two integers 17 13 numbers not equal 5.2.3 Nesting of the if...else Statements You witnessed the usage of a single if statement in Example 5.1. You saw if followed by else in Example 5.2. There is no restriction to the number of if, which can be used in a program. This applies to else as well, but else can only follow an if statement. You can have the following in a program: { if (condition1) { if (condition2) {statements–s1} else if (condition3) {statements–s2,} } else {statements–s4} statements–s5 } This is called a nested if and else statement. As the level of nesting increases, it will be difficult to analyse and logical mistakes will be made more easily. In the above example, when condition1 is false, statements–s4 will be executed. If condition1 is true and condition2 is also true, then statements– s1 will be executed. | did not get any message when the numbers were unequal and this can be avoided by using the else statement. The usage of if .. else is shown below. if (condition true) { Decision Making and Branching NOTES statements s1 } else { statements s2 } statements s3; The statement else is always associated with an if. If the condition is true, then statements s1 will be executed. After executing them, the program will skip the else block and control goes to statement s3 that follows the else block. If the condition is false, then the statements in the else block, i.e., s2 will be executed followed by statement s3. Statements s1 will not be executed at all when the condition becomes false. The usage of braces clearly brings out which statements belong to the if block and which to the else block. Example 5.2 brings out the usage of if... else. /*Example 5.2 To demonstrate the use of if.. else*/ #include &gt;stdio.h&lt; main() { int a,Self-Instructional Material 59 and Branching NOTES printf ("enter two integers\n"); scanf("%u%u", &a, &b); if (a==b) { printf("you typed equal numbers\n"); } else { printf("numbers not equal\n"); } } The output of the program when unequal numbers were keyed in is as follows. Result of the program enter two integers 17 13 numbers not equal 5.2.3 Nesting of the if...else Statements We witnessed the usage of a single if statement in Example 5.1. We saw if followed by else in Example 5.2. There is no restriction to the number of if, which can be used in a program. This applies to else as well, but else can only follow an if statement. We can have the following in a program: { if (condition1) { } else if (condition2) {statements–s1} else if (condition3) {statements–s2,} { statements–s4} statements–s5 } This is called a nested if and else statement. As the level of nesting increases, it will be difficult to analyse and logical mistakes will be made more easily. Self-Instructional 60 Material In the above example, when condition1 is false, statements– s4 will be executed. If condition1 is true and condition2 is also true, then statements–s1 will be executed. |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 21/126 | SUBMITTED TEXT 80 WORDS | 92% MATCHING TEXT 80 WORDS |
|---|---|---|
| | The while Loop The while loop is a subset of the for loop. The syntax for the while loop is given below: while (expression) {statements;} This means that the statement(s), which is a single statement or multiple statements, will be executed while the expression is true. When it becomes false, the execution will stop. The while is similar to the for loop without exp1 and exp3. The for loop can be simulated or replaced with the while loop as given below. exp1; while (exp2) { statements exp3; } The programmer can use while or for | The while Loop The while loop is a subset of the for loop. The syntax for the while loop is given as follows: while (expression) {statements;} This means that the statement(s), which is a single statement or multiple statements, will be executed while the expression is true. When it becomes false, the execution will stop. The while loop is similar to the for loop without exp1 and exp3. The for loop can be simulated or replaced with the while loop as follows. exp1; while (exp2) { statements exp3; } Self-Instructional 84 Material The programmer can use while or for |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 22/126 | SUBMITTED TEXT 19 WORDS | 40% MATCHING TEXT 19 WORDS |
|---|---|---|
| | The following program confirms the same: /*Example 3.3*/ /* to demonstrate typecasting */ #include &gt;stdio.h&lt; int main() { int x; float y = 10.67; x=(int)y; printf("x = %d | |

SA C book final copy (1).doc (D31388791)

| 23/126 | SUBMITTED TEXT 165 WORDS | 99% MATCHING TEXT 165 WORDS |
|---|---|---|
| | alpha=(alpha+32); putch(alpha); } else { if(alpha=='1') printf("End of Session"); else printf("\ninvalid entry; retry"); } }while(alpha!='1'); } Result of the program enter upper case alphabet- enter 1 to quit Gg enter upper case alphabet- enter 1 to quit o invalid entry; retry enter upper case alphabet- enter 1 to quit Dd enter upper case alphabet- enter 1 to quit 1End of Session How does it differ? Here too, the program will attempt to convert one character before it can be terminated. Assuming that the first character was 1, the program will still attempt to convert it and print the message "End of Session" before it quits. Suppose the first character is a valid one and a number of characters are converted in succession; when you want to terminate the program, 1 has to be pressed and even then the program will not stop immediately. It will stop only after the statements are executed. Since the problem is the same, a detailed look at both the examples will bring out the similarity in operation between both the constructs. However, there are occasions when it is quite suitable as given in the next section. 5.4 | alpha=(alpha+32); putch(alpha); } else { if(alpha=='1') printf("End of Session"); else printf("\ninvalid entry; retry"); } }while(alpha!='1'); } Self-Instructional 90 Material Result of the program enter upper case alphabet- enter 1 to quit Gg enter upper case alphabet- enter 1 to quit o invalid entry; retry enter upper case alphabet- enter 1 to quit Dd enter upper case alphabet- enter 1 to quit 1End of Session How does it differ? Here too, the program will attempt to convert one character before it can be terminated. Assuming that the first character was 1, the program will still attempt to convert it and print the message "End of Session" before it quits. Suppose the first character is a valid one and a number of characters are converted in succession; when you want to terminate the program, 1 has to be pressed and even then the program will not stop immediately. It will stop only after the statements are executed. Since the problem is the same, a detailed look at both the examples will bring out the similarity in operation between both the constructs. However, there are occasions when it is quite suitable as given in the next section. |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 24/126 SUBMITTED TEXT 298 WORDS | 91% MATCHING TEXT 298 WORDS |
|---|---|
| want to get a one digit number up to 9 and print its value in words, it would become a more complex task. The switch statement comes in handy in such situations. The syntax of the switch statement is as follows: switch (expression) { case constant or expression : statements case constant or expression : statements .. default : statements } When the switch keyword is encountered, the associated expression is evaluated. The program now looks for the case, which matches with the value of the expression. Execution then starts from the statement corresponding to the case which matches. Each case has to be accompanied by integer expressions, which must be unique, as otherwise the program will not know where to start. For example, if the first two cases are as follows: case 10 : s1; case 10 : s2; In this case, the program would not know whether to execute s1 or s2 when the expression of switch evaluates to 10. Therefore, the constant expressions following the case keyword should all be unique. There may be occasions when none of the constant expressions matches the switch expression in which case the default statements will be executed. Thus, switch allows branching of the program execution to an appropriate place. A program to print the values of the digits in words is given below: /*Example 5.16 converts the digits 0-9 in words*/ #include &gt;stdio.h&lt; #include &gt;conio.h&lt; Self-Instructional Material 111 Control Statements NOTES main() { int a; char ch ='c'; while (ch=='c') { printf("\nEnter a digit 0 t0 9\n"); scanf("%d",&a); switch(a) { case 0:printf("Zero\n"); break; case 1:printf("One\n"); break; case 2:printf("Two\n"); break; case 3:printf("Three\n"); break; case 4:printf("Four\n"); break; case 5:printf("Five\n"); break; case 6:printf("Six\n"); break; case 7:printf("Seven\n"); break; case 8:printf("Eight\n"); break; case 9:printf("Nine\n"); break; default:printf("Illegal character\n"); } printf("enter 'c' if you want to continue\n"); printf("or any other character to end\n"); ch=getche(); if (ch!='c') printf("End of Session"); } } Result of the program Enter a digit 0 t0 9 6 Six enter 'c' if you want to continue 112 Self-Instructional Material Control Statements NOTES or any other character to end c Enter a digit 0 t0 9 7 Seven enter 'c' if you want to continue or any other character to end nEnd of Session | want to get a one digit number up to 9 and print its value in words, it would become a more complex task. The switch statement comes in handy in such situations. The syntax of the switch statement is as follows: switch (expression) { case constant or expression : statements case constant or expression : statements .. default : statements } When the switch keyword is encountered, the associated expression is evaluated. The program now looks for the case, which matches with the value of the expression. Execution then starts from the statement corresponding to the case which matches. Each case has to be accompanied by integer expressions, which must be unique, as otherwise the program will not know where to start. For example, if the first two cases are as given below: case 10 : s1; case 10 : s2; In this case the program would not know whether to execute s1 or s2 when the expression of switch evaluates to 10. Therefore, the constant expressions following the case keyword should all be unique. There may be occasions when none of the constant expressions matches the switchexpression in which case the default statements will be executed. Thus switch allows branching of the program execution to appropriate place. A program to print the values of the digits in words is given below: /*Example 5.8 converts the digits 0-9 in words*/ #include &gt;stdio.h&lt; #include &gt;conio.h&lt; Self-Instructional 68 Material int a; char ch ='c'; while (ch=='c') { printf("\nEnter a digit 0 t0 9\n"); scanf("%d",&a); switch(a) { case 0:printf("Zero\n"); break; case 1:printf("One\n"); break; case 2:printf("Two\n"); break; case 3:printf("Three\n"); break; case 4:printf("Four\n"); break; case 5:printf("Five\n"); break; case 6:printf("Six\n"); break; case 7:printf("Seven\n"); break; case 8:printf("Eight\n"); break; case 9:printf("Nine\n"); break; default:printf("Illegal character\n"); Decision Making and Branching NOTES } enter 'c' if you want to continue\n"); printf("or any other character to end\n"); ch=getche(); if (ch!='c') printf("End of Session"); } } Result of the program Enter a digit 0 t0 9 6 Six enter 'c' if you want to continue any other to end Self-Instructional Material 69 Making and Branching NOTES Self-Instructional 70 Material Enter a digit 0 t0 9 7 Seven enter 'c' if you want to continue or any other character to end nEnd of Session |

**W** https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 25/126 SUBMITTED TEXT 12 WORDS | 84% MATCHING TEXT 12 WORDS |
|---|---|
| include&gt;stdio.h&lt; main() { int a, b; scanf ("%d %d", &a, &b); if ( (a+b) &gt;100) printf( " | include&gt;stdio.h&lt; void int a, b, c; scanf("%d%d",&a,&b); 10 c=a+b; printf("% |

**W** https://www.iare.ac.in/sites/default/files/AERO_PROGRAMMING_FOR_PROBLEM_SOLVING_LECTURE_NOTES.pdf

| 26/126 SUBMITTED TEXT 5 WORDS | 61% MATCHING TEXT 5 WORDS |
|---|---|
| include&gt;stdio.h&lt; #include&gt;conio.h&lt; main() { int ch; ch=getch(); if(ch&lt;='A' && ch&gt;='Z') { ch= | |

**SA** C_book_AmrutaJog_ShrutiJathar.pdf (D54111295)

| 27/126 | SUBMITTED TEXT | 17 WORDS | 68% MATCHING TEXT | 17 WORDS |
|---|---|---|---|---|

Write a program to find the factorial of a given number using while. 4. Write a program,

SA Block 3 and 4.pdf (D129191893)

| 28/126 | SUBMITTED TEXT | 31 WORDS | 47% MATCHING TEXT | 31 WORDS |
|---|---|---|---|---|

Self-Instructional Material Functions NOTES Self-Instructional Material 127 Functions NOTES UNIT 6 FUNCTIONS Structure 6.0 Introduction 6.1 Unit Objectives 6.2 Modular Programming Overview 6.3 Function Prototypes 6.4 Function Call – Passing Arguments to a Function 6.4.1 Function Arguments 6.5 Function Definition 6.6 Scope Rules for

Self-Instructional Material 133 BLOCK - III USER DEFINED FUNCTIONS UNIT 8 FUNCTIONS BASICS Structure 8.0 Introduction 8.1 Objectives 8.2 General Forms of Function Prototype 8.2.2 Function Call – Passing Arguments to a Function 8.2.3 Function Definition 8.2.4 Function Arguments 8.2.5 Scope: Rules for

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 29/126 | SUBMITTED TEXT | 24 WORDS | 52% MATCHING TEXT | 24 WORDS |
|---|---|---|---|---|

Write a program to find the factorial of 10. 9. Write a program to find the numbers in any range, which are divisible by

SA Dr. Baldev SIngh LKC_Programming in C.pdf (D137918641)

| 30/126 | SUBMITTED TEXT | 171 WORDS | 94% MATCHING TEXT | 171 WORDS |
|---|---|---|---|---|

function definition consists of two parts, namely function declarator or heading and function functions. The function heading is similar to function declaration, but will not terminate with a semicolon. The use of functions will be demonstrated with simple programs in this unit. Assume that you wish to get two integers. Pass them to a function add. Add them in the add function. Return the value to the main function and print it. The algorithm for solving the problem is as follows: Main Function Step 1: define function add Step 2: get 2 integers Step 3: call add and pass the 2 values Step 4: get the sum Step 5: print the value function add Step 1: get the value Step 2: add them Step 3: return the value to main Thus you have divided the problem. The program is given below: /*Example 6.1*/ /* use of function*/ #include &gt;stdio.h&lt; int main() { int a=0, b=0, sum=0; int add(int a, int b); /*function declaration*/ printf("enter 2 integers\n"); scanf("%d%d", &a, &b); sum =add(a, b); /*function call*/ printf("sum of %d and %d =%d", a, b, sum); } /*function definition*/ int add (int c, int d) /*function declarator*/ { int e; e= c+d; return e; }

Function definition consists of two parts, namely function declarator or heading and function declarations. The function heading is similar to function declaration but will not terminate with a semicolon. The use of functions will be demonstrated with simple programs in this unit. Suppose you wish to get two integers. Pass them to a function add. Add them in the add function. Return the value to the main function and print it. The algorithm for solving the problem will be as follows: Main Function Step 1: Define function add Step 2: Get 2 integers Step 3: Call add and pass the 2 values Step 4: Get the sum Step 5: Print the value function add Step 1: Get the value Step 2: Add them Step 3: Return the value to main Thus, you have divided the problem. The program is as follows: /*Example 8.1* /* use of function*/ #include &gt;stdio.h&lt; int main() { int a=0, b=0, sum=0; int add(int a, int b); /*function declaration*/ printf("enter 2 integers\n"); scanf("%d%d", &a, &b); sum =add(a, b); /*function call*/ printf("sum of %d and %d =%d", a, b, sum); } Functions Basics NOTES /*function definition*/ int add (int c, int d) /*function declarator*/ { int e; e= c+d; return e; }

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 31/126 | SUBMITTED TEXT 242 WORDS | 99% MATCHING TEXT 242 WORDS |
|---|---|---|
| | the declaration in the calling program. They must also appear in the same order. • At the time of execution, when the function encounters the closing brace }, it returns control to the calling program and returns to the same place at which the function was called. In this program, you have a specific statement return (e) before the closing brace. Therefore, the program will go back to the main function with value of e. This value will be substituted as, sum = (returned value) Therefore, sum gets the value, which is printed in the next statement. This is how the function works. Assume now that the program gets a and b values, gets their sum1, gets c and d and gets their sum2 and then both the sums are passed to the function to get their total. The program for doing this is as follows: /*Example 6.2*/ /* A function called many times */ #include &gt;stdio.h&lt; int main() { float a, b, c, d, sum1, sum2, sum3; float add(float a, float b); /*function declaration*/ printf("enter 2 float numbers\n"); scanf("%f%f", &a, &b); sum1 =add(a, b); /*function call*/ printf("enter 2 more float numbers\n"); scanf("%f%f", &c, &d); sum2 =add(c, d); /*function call*/ sum3 =add(sum1, sum2); /*function call*/ printf("sum of %f and %f =%f\n", a, b, sum1); printf("sum of %f and %f =%f\n", c, d, sum2); printf("sum of %f and %f =%f\n", sum1,sum2, sum3); } /*function definition*/ float add (float c, float d) /*function declarator*/ { float e; e = c+d; return e; } Result of the program enter 2 float numbers 1.5 3.7 enter 2 more float numbers 5.6 8.9 sum of 1.500000 and 3.700000 =5.200000 | the declaration in the calling program. They must also appear in the same order. (c) At the time of execution, when the function encounters the closing brace }, it returns control to the calling program and returns to the same place at which the function was called. In this program, you have a specific statement return (e) before the closing brace. Therefore, the program will go back to the main function with the value of e. This value will be substituted as sum = (returned value) Therefore, sum gets the value which is printed in the next statement. This is how the function works. Assume now that the program gets a and b values, gets their sum1, gets c and d and gets their sum2 and then both the sums are passed to the function to get their total. The program for doing this is as follows: /*Example 8.2* /* A function called many times */ #include &gt;stdio.h&lt; int main() { float a, b, c, d, sum1, sum2, sum3; float add(float a, float b); /*function declaration*/ printf("enter 2 float numbers\n"); scanf("%f%f", &a, &b); sum1 =add(a, b); /*function call*/ printf("enter 2 more float numbers\n"); scanf("%f%f", &c, &d); sum2 =add(c, d); /*function call*/ sum3 =add(sum1, sum2); /*function call*/ printf("sum of %f and %f =%f\n", a, b, sum1); printf("sum of %f and %f =%f\n", c, d, sum2); printf("sum of %f and %f =%f\n", sum1,sum2, sum3); } Functions Basics NOTES /*function definition*/ float add (float c, float d) /*function declarator*/ { float e; e= c+d; return e; } Result of the program enter 2 float numbers 1.5 3.7 enter 2 more float numbers 5.6 8.9 sum of 1.500000 and 3.700000 =5.200000 |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 32/126 | SUBMITTED TEXT 32 WORDS | 89% MATCHING TEXT 32 WORDS |
|---|---|---|
| | ADD digits function Step 1: sum = 0 Step 2: while number &lt; 0 sum = sum + (number % 10) number = number / 10 Step 3: return (sum) See how the above algorithm adds digits | add–digits function Step 1: sum = 0 Step 2: while number &lt; 0 sum = sum + (number % 10) number = number / 10 Step 3: return (sum) Let us see how the above algorithm add–digits |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 33/126 | SUBMITTED TEXT 261 WORDS | 95% MATCHING TEXT 261 WORDS |
|---|---|---|
| | Give 4321 as the number. Step 1: sum = 0 Step 2: Iteration 1 sum = 0 + modulus of (4321/10) = 0 + 1 = 1 number = 4321/10 = 432 Iteration 2 sum = 1 + modulus of (432/10) = 1 + 2 After 4 iterations, sum =1 + 2 + 3 + 4 Step 3: Sum is returned. The program is given below: /*Example 6.3*/ /*program to demonstrate calling multiple functions*/ #include&gt;stdio.h&lt; int main() { long nummul=0; long num=0, rev=0, add_digit=0; /*good practice to initialize all variables*/ long reverse(long num); long mult(long num); int sum_digit(long num); printf("enter unsigned number\n"); scanf("%lu", &num); if (num%2) /*remainder 1*/ { rev = reverse(num); printf("number is odd\n"); printf("number entered=%lu\n number reversed=%lu\n", num, rev); } else { Self-Instructional Material 137 Functions NOTES nummul=mult(num); printf("number is even\n"); printf("number=%lu\n its multiple=%lu\n", num, nummul); } if (num%3 ==0) { add_digit= sum_digit(num); printf("number evenly divisible by 3\n"); printf("sum of digits =%lu", add_digit); } } long reverse(long n) { long r=0; while (n&lt;0) { r=r*10+(n%10); n=n/10; } return r; } long mult(long p) { long sq; sq=2*p; return sq; } int sum_digit(long num) { long sum=0; while (num &lt;0) { sum=sum+(num%10); num=num/10; } return sum; } Result of the program enter unsigned number 4321 number is odd number entered=4321 number reversed=1234 138 Self-Instructional Material Functions NOTES Look at the program. After getting the number from the user, it evaluates if remainder of (num/2) = true; i.e., if the remainder is 1, then it is true. If remainder = 1, then the number is odd and hence, the reverse function is called. The returned value is assigned to rev and printed. If the number is even, the number is doubled. Since the doubled value may exceed the maximum of the unsigned integer, we have declared it as a long integer. Next we check if the number is evenly divisible by 3. If it is so then we add the digits. Thus, the main function of Example 6.3 calls three functions for carrying out specific tasks. All the three functions are supplied with the same arguments, but return different values. 6.6 | give 4321 as the number. Step 1: sum = 0 Step 2: Iteration 1 sum = 0 + modulus of (4321/10) = 0 + 1 = 1 number = 4321/10 = 432 Iteration 2 sum = 1 + modulus of (432/10) = 1 + 2 Functions Basics NOTES After 4 iterations: sum =1+2+3+4 Step 3: sum is returned. The program is given below: /*Example 8.3*/ /*program to demonstrate calling multiple functions*/ #include&gt;stdio.h&lt; int main() { long nummul=0; long num=0, rev=0, add_digit=0; /*good practice to inialize all variables*/ long reverse(long num); long mult(long num); int sum_digit(long num); printf("enter unsigned number\n"); scanf("%lu", &num); if (num%2) /*remainder 1*/ { rev = reverse(num); printf("number is odd\n"); printf("number entered=%lu\n number reversed=%lu\n", num, rev); } else { Self-Material 141 Functions Basics nummul=mult(num); printf("number is even\n"); printf("number=%lu\n its multiple=%lu\n", num, nummul); } if (num%3 ==0) { add_digit= sum_digit(num); printf("number evenly divisible by 3\n"); printf("sum of digits =%lu", add_digit); } } long reverse(long n) { long r=0; while (n&lt;0) { r=r*10+(n%10); n=n/10; } return r; } long mult(long p) { long sq; sq=2*p; return sq; } int sum_digit(long num) { long sum=0; while (num &lt;0) { sum=sum+(num%10); num=num/10; } return sum; } Self-Instructional 142 Material Result of the program enter unsigned number 4321 number is odd entered=4321 number reversed=1234 Look at the program. After getting the number from the user, it evaluates if the remainder of (num/2) = true; i.e., if the remainder is 1, then it is true. If the remainder = 1, then the number is odd and hence the reverse function is called. The returned value is assigned to rev and printed. If the number is even, the number is doubled. Since the doubled value may exceed the maximum of the unsigned integer, we have declared it as a long integer. Next, we check if the number is evenly divisible by 3. If it is so, then we add the digits. Thus, the main function of Example 8.3 calls three functions for carrying out specific tasks. All the three functions are supplied with the same arguments but return different values. 8.2.4 |

**W** https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 34/126 | SUBMITTED TEXT 12 WORDS | 95% MATCHING TEXT 12 WORDS |
|---|---|---|
| | f 1 f 2 f 3 f 4 f 11 f 12 f 21 f 31 f 32 f 41 f 42 | |

**SA** C book final copy (1).doc (D31388791)

| 35/126 | SUBMITTED TEXT | 164 WORDS | 98% MATCHING TEXT | 164 WORDS |
|---|---|---|---|---|

The scope of the variable is local to the function unless it is a global variable. For example, int function1(int I ) { int j=100; double function2 (int j) ; function2 (j) ; } double function2 (int p) { double m; return m; } The variable j in function1 is not known to function2. You pass it to function2 through the argument j. This will be assigned as equal to int p. Similarly, m in function2 is not known to function1. It can be made known to function1 through the return statement. This makes the scope rules of variables in function quite clear. The scope of variables is local to the function where defined. However, global variables are accessible by all functions in the program if they are defined above all functions. /*Example 6.4*/ /* to demonstrate that the scope of a variable is local to the function*/ #include &gt;stdio.h&lt; int main() { float a=100.250, b=200.50; void change (float a, float b); change(a, b); printf("a= %f b= %f\n ", a, b); printf("these are the original values"); } /*function definition*/ void change (float a, float b) /*function declarator*/

The scope of the variable is local to the function, unless it is a global variable. For instance, int function1(int I ) { int j=100; double function2 (int j) ; function2 (j) ; } double function2 (int p) { double m; return m; } The variable j in function1 is not known to function2. You pass it to function2 through the argument j. This will be assigned as equal to int p. Similarly, m in function2 is not known to function1. It can be made known to function1 through the return statement. This makes the scope rules of variables in function quite clear. The scope of variables is local to the function where defined. However, global variables are accessible by all the functions in the program if they are defined above all functions. /*Example 8.4* /* To demonstrate that the scope of a variable is local to the function*/ #include &gt;stdio.h&lt; int main() { float a=100.250, b=200.50; void change (float a, float b); change(a, b); printf("a= %f b= %f\n ", a, b); these are the original values"); } Self-Instructional 144 Material /*function definition*/ void change (float a, float b) /*function declarator*/ {

**W** https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 36/126 | SUBMITTED TEXT | 117 WORDS | 95% MATCHING TEXT | 117 WORDS |
|---|---|---|---|---|

int main() { int m, n; int gcd(int m, int n); printf("Enter 2 integers\n"); scanf("%d %d", &m,&n); printf("GCD of %d and %d=%d", m, n, gcd(m,n)); } int gcd(int m, int n) { if (n==0) return m; else return (gcd(n, m%n)); } The program was executed twice and the result of the program is given below. Result of the program First Time Enter 2 integers 12 256 GCD of 12 and 256 = 4 Second Time Enter 2 integers 1225 625 GCD of 1225 and 625 = 25 We can even enter the first number to be lower than the second number as executed during the first time. The program works all right because in one iteration, the numbers get reversed, namely the first number is larger than the second number after iteration. Thus, recursion is quite suitable for solving this problem. 6.11

int main() { int m, n; int gcd(int m, int n); printf("Enter 2 integers\n"); scanf("%d %d", &m,&n); printf("GCD of %d and %d=%d", m, n, gcd(m,n)); } int gcd(int m, int n) { if (n==0) return m; else return (gcd(n, m%n)); } Functions Basics The program was executed twice and the result of the program is as follows: Result of the program First Time Enter 2 integers 12 256 GCD of 12 and 256=4 Second Time Enter 2 integers 1225 625 GCD of 1225 and 625=25 You can even enter the first number to be lower than the second number as executed during the first time. The program works all right because in one iteration, the numbers get reversed namely the first number is larger than the second number after iteration. Thus, recursion is quite suitable for solving this problem. 8.5

**W** https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 37/126 | SUBMITTED TEXT | 8 WORDS | 100% MATCHING TEXT | 8 WORDS |
|---|---|---|---|---|

Describe what the following programs do: (a) #include &gt;stdio.h&lt;

Describe what the following programs do: (a) #include &gt;stdio.h&lt;

**W** https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 38/126 | SUBMITTED TEXT | 12 WORDS | 95% MATCHING TEXT | 12 WORDS |
|---|---|---|---|---|

variables are declared within a function and are local to the function.

**SA** BCADSC-1-FUNDAMENTALS OF COMPUTERS AND PROGRAMMING IN C-Block-1-4 and 9-16-2-266.pdf (D138636232)

| 39/126 | SUBMITTED TEXT | 15 WORDS | 90% | MATCHING TEXT | 15 WORDS |
|---|---|---|---|---|---|

Any variable declared within a function is interpreted as an auto variable unless a different

SA BCADSC-1-FUNDAMENTALS OF COMPUTERS AND PROGRAMMING IN C-Block-1-4 and 9-16-2-266.pdf (D138636232)

| 40/126 | SUBMITTED TEXT | 17 WORDS | 76% | MATCHING TEXT | 17 WORDS |
|---|---|---|---|---|---|

static) having the same names. In such cases, the local variables take precedence over the external variables.

SA BCADSC-1-FUNDAMENTALS OF COMPUTERS AND PROGRAMMING IN C-Block-1-4 and 9-16-2-266.pdf (D138636232)

| 41/126 | SUBMITTED TEXT | 12 WORDS | 95% | MATCHING TEXT | 12 WORDS |
|---|---|---|---|---|---|

variables are declared within a function and are local to the function.

SA BCADSC-1-FUNDAMENTALS OF COMPUTERS AND PROGRAMMING IN C-Block-1-4 and 9-16-2-266.pdf (D138636232)

| 42/126 | SUBMITTED TEXT | 15 WORDS | 90% | MATCHING TEXT | 15 WORDS |
|---|---|---|---|---|---|

Any variable declared within a function is interpreted as an auto variable unless a different

SA BCADSC-1-FUNDAMENTALS OF COMPUTERS AND PROGRAMMING IN C-Block-1-4 and 9-16-2-266.pdf (D138636232)

| 43/126 | SUBMITTED TEXT | 17 WORDS | 85% | MATCHING TEXT | 17 WORDS |
|---|---|---|---|---|---|

| | |
|---|---|
| include &gt;stdio.h&lt; main() { auto int i = 10; { auto int i = 5; printf (“%d”, i); } printf(“%d”, i + 5); printf(“%d”, | include &gt;stdio.h&lt; int main( ) { auto int auto int i = 2; { auto int i = 3; printf ( "\ n%i); } printf ( "%d ", i); } printf( "%d\ |

W http://referenceglobe.com/kpsslp/support/upload_videos/programming%20for%20problem%20solving_1585 ...

| 44/126 | SUBMITTED TEXT | 15 WORDS | 90% | MATCHING TEXT | 15 WORDS |
|---|---|---|---|---|---|

Any variable declared within a function is interpreted as an auto variable unless a different

SA BCADSC-1-FUNDAMENTALS OF COMPUTERS AND PROGRAMMING IN C-Block-1-4 and 9-16-2-266.pdf (D138636232)

| 45/126 | SUBMITTED TEXT 202 WORDS | 88% MATCHING TEXT 202 WORDS |
|---|---|---|
| | How do you give a name to each element? You can use an index with the name of the array to indicate the elements. While in mathematics the first element can be called with an index [1], in "C" the first element is called with the index [0]. Index and subscript are used interchangeably to indicate the position of the element in the array. Thus, A [0] = 2 A [1] = 3 A [2] = 5 & A [3] = 7 Similarly, B [0] = Red B [1] = Green B [2] = Yellow Array A has four elements and hence the size of A is 4. Similarly, B has three elements and hence its size is 3. Therefore, the first element of any array will have a subscript of 0 and the final element a subscript of n–1, where n is the size of the array. Array Declaration Assuming that there are 40 employees in an office and you want to store their ages, you would have to create 40 variables of type integer or float and store their ages. While such a definition seems alright, it would cause complications while programming. Instead, this can be declared as an array of size 40. For example, int emp_age [40]; Here, | How do we give a name to each element? We can use an index with the name of the array to indicate the elements. While in mathematics, the first element is identified with an index [1], in "C" the first element is identified with the index [0]. Index or subscript is used to indicate the position of the element in the array. Thus, A [0] = 2 A [1] = 3 A [2] = 5 & A [3] = 7 Similarly, B [0] = Red B [1] = Green B [2] = Yellow Array A has four elements and hence the size of A is 4. Similarly, B has three elements and hence its size is 3. Therefore, the first element of any array will have a subscript of 0 and the final element a subscript of n–1, where n is the size of the array. Self-Instructional 96 Material 6.2.1 Array Declaration Assuming that there are 40 employees in an office and we want to store their ages, we would have to create 40 variables of type integer or float and store their ages. While such a definition seems alright, it would cause complications while programming. Instead, this can be declared as an array of size 40. For instance, short emp_age [40]; Here, |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 46/126 | SUBMITTED TEXT 161 WORDS | 89% MATCHING TEXT 161 WORDS |
|---|---|---|
| | the same name with a subscript corresponding to the position, i.e., the array name with the subscript written in square brackets. Consider another program to understand one-dimensional arrays more clearly. Assume the existence of an array of elements. You want to find out the greatest number in the array and its location. To do that we set up two other variables known as max and ind. You initialize them to zero. Then you compare each number with max. If it is greater than the max, then we note the location in ind and the associated value in max. When you have checked all the elements in the array, you will get the greatest number and its location. Since the first element has a subscript 0, you have to add 1 to the subscript to get the position. The program is given below: /*Example 8.2 to find the greatest number and its position in an array*/ #include&gt;stdio.h&lt; int main() { int a[5]= {1,5,2,6,3}; int max=0, i, ind=0; for(i=0; i&gt;=4; i++) { if (a[i] &lt; max) { max =a[i]; ind=i+1; } } printf("maximum number=%d location=%d\n", max, ind); } 170 Self-Instructional Material | the same name with a subscript corresponding to the position, i.e., the array name with the subscript written in square brackets. Consider another program to understand one-dimensional arrays more clearly. Assume the existence of an array of integers. We want to find the greatest number in the array and its location. To do that, we set up two other variables known as max and ind. We initialize them to zero. Then we compare each number Self-Instructional Material 99 Arrays NOTES with max. If it is greater than the max then we note the location in ind and the associated value in max. When we have checked all the elements in the array we would have got the greatest number and its location. Since the first element has a subscript 0, we have to add 1 to the subscript to get the position. The program is given below: to find the greatest number and its position in an array*/ #include&gt;stdio.h&lt; int main() { int a[5]= {1,5,2,6,3}; int max=0, i, ind=0; for(i=0; i&gt;=4; i++) { if (a[i] &lt; max) { max =a[i]; ind=i+1; } } printf("maximum number=%d location=%d\n", max, ind); } Self-Instructional 100 Material |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 47/126 | SUBMITTED TEXT 137 WORDS | 100% MATCHING TEXT 137 WORDS |
|---|---|---|
| | Result of the program maximum number=6 location=4 Let us see how the program works. Iteration 1 i = 0 max = 0 ind = 0 a[0] = 1 since a[0] &lt; max max = 1 ind = 1 Iteration 2 i = 1 max = 1 ind = 1 a[1] = 5 since a[1] &lt; max max = 5 ind = 2 Iteration 3 i = 2 max = 5 ind = 2 a[2] = 2 since a[2] &gt; max no change Iteration 4 i = 3 max = 5 ind = 2 a[3] = 6 Since a[3] &lt; max max = 6 ind = 4 Iteration 5 i = 4 Max = 6 ind = 4 a[4] = 3 since a [4] &gt; max no change The program prints: max = 6 location = 4 Thus, arrays are very useful for solving real problems encountered every day. 8.4 | Result of the program maximum number=6 location=4 Let us see how the program works. Iteration 1 i = 0 max = 0 ind = 0 a[0] = 1 since a[0] &lt; max max = 1 ind = 1 Iteration 2 i = 1 max = 1 ind = 1 a[1] = 5 since a[1] &lt; max max = 5 ind = 2 Iteration 3 i = 2 max = 5 ind = 2 a[2] = 2 since a[2] &gt; max no change Iteration 4 i = 3 max = 5 ind = 2 a[3] = 6 Since a[3] &lt; max max = 6 ind = 4 Iteration 5 i = 4 Max = 6 ind = 4 a[4] = 3 since a [4] &gt; max no change The program prints max = 6 location = 4 Thus, arrays are very useful for solving real problems encountered every day. 6.2.3 |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 48/126 | SUBMITTED TEXT 278 WORDS | 96% MATCHING TEXT 278 WORDS |
|---|---|---|
| | is a two-dimensional array with different subscripts. Here, there will be 50 different elements. The first element can be denoted as w [0][0]. The next element will be w [0][1]. The fifth element will be w [0][4]. The sixth element will be w [1][0]. The last element will be w [9][4]. This can be considered as a row and column representation. There are 10 rows and 5 columns in this example. When data is stored in the array, the second subscript will change from 0 to 4, one at a time, with the first subscript remaining constant at 0. Then the first subscript will become 1 and the second subscript will keep increasing from 0 to 4. This is repeated till the first subscript becomes 9 and the second 4. This array can be used to represent the names of 10 persons, with each name containing 5 characters. The first subscript refers to the name of the 0th person, 1st person, 2nd person and so on. The second subscript refers to the 1st character, 2nd character and so on of the name of a person. Thus, 10 such names can be stored in this array. The dimension of the array can be increased to 3 with 3 square brackets as given below: Marks [50][3][3]; The name of the first element will be Marks [0][0][0] The last element will be Marks [49][2][2]. It would be easy to add more dimensions to an array but it would also become more difficult to comprehend under normal circumstances. It may, therefore, be useful to solve complicated scientific applications. Now let us understand the concept of multidimensional arrays using a simple problem. Assume that you need to write a program to read two arrays (both two-dimensional) and multiply the corresponding elements and store them in another | is a two-dimensional array with different subscripts. Here, there will be 50 different elements. The first element can be denoted as w [0][0]. The next element will be w[0][1]. The fifth element will be w[0][4]. The sixth element will be w[1][0]. The last element will be w [9][4]. This can be considered as a row and column representation. There are 10 rows and 5 columns in this example. When data is stored in the array, the second subscript will change from 0 to 4, one at a time, with the first subscript remaining constant at 0. Then the first subscript will become 1 and the second subscript will keep increasing from 0 to 4. This is repeated till the first subscript becomes 9 and the second 4. This array can be used to represent the names of 10 persons, with each name containing 5 characters. The first subscript refers to the name of the 0th person, 1st person, 2nd person and so on. The second subscript refers to the 1st character, 2nd character and so on of the name of a person. Thus, 10 such names can be stored in this array. The dimension of the array can be increased to 3 with 3 square brackets as given below: Marks [ 50 ][ 3 ][3 ]; The name of the first element will be Marks [ 0 ] [ 0 ] [ 0 ] The last element will be Marks [ 49 ] [ 2 ] [2 ]. It would be easy to add more dimensions to an array but it would also become more difficult to comprehend under normal circumstances. It may, therefore, be useful to solve complicated scientific applications, however. Now let us focus on the concept of multi-dimensional arrays using a simple problem. Arrays NOTES Self-Instructional Material 101 Arrays NOTES Assume that we need to write a program to read two arrays (both 2 dimensional) and multiply the corresponding elements and store them in another 2- |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 49/126 | SUBMITTED TEXT 86 WORDS | 88% MATCHING TEXT 86 WORDS |
|---|---|---|
| | y[2] as follows: x = { 1 2 } y = { 5 6 } { 3 4 } { 7 8 } x [0][0] = 1 x [1][1] = 4 y [0][0] = 5 y [1][1] = 8 Therefore, after multiplication of the respective elements, we get, z = {5 12} {21 32} The program prints out the values of the products stored in array z. Result of the program z[0][0]=5 z[0][1]=12 z[1][0]=21 z[1][1]=32 Note that the elements are stored row by row | y[2] as follows: x = { 1 2 } y = { 5 6 } { 3 4 } { 7 8 } x [0][0] = 1 x [1][1] = 4 y [0][0] = 5 y [1][1] = 8 Therefore, after multiplication of the respective elements, we get z = { 5 12 } { 21 32 } The program prints the values of the products stored in array z. The output of the program appears as follows: z[0][0]=5 z[0][1]=12 z[1][0]=21 z[1][1]=32 Note that the elements are stored contiguously, row by row. |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 50/126 | SUBMITTED TEXT 26 WORDS | 88% MATCHING TEXT 26 WORDS |
|---|---|---|
| | second matrix] Assume that A[3][2] and B[2][4] The product matrix in this case will be P [3] [4]. This has to be understood clearly. The number of rows of | second matrix] Assume A [3] [2] & B [2] [4] The product matrix in this case will be P [3] [4]. This has to be understood clearly. The number of rows of |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 51/126 SUBMITTED TEXT 16 WORDS | 84% MATCHING TEXT 16 WORDS |
|---|---|
| OBJECTIVES After going through this unit, you will be able to: • Define array and multidimensional array • | |

SA Programming in C Block 2.pdf (D164968573)

| 52/126 SUBMITTED TEXT 27 WORDS | 95% MATCHING TEXT 27 WORDS |
|---|---|
| Let us now write a program to read a string with scanf() and write with printf(). The program is given below: /*Example 8.7 Reading and writing with formatted I/O functions*/ #include&gt;stdio.h&lt; | Let us now write a program to read a string with scanf() and write with printf(). Self-Instructional 122 Material The program is given below: Example 7.1 /* Reading and writing with formatted I/O functions*/ #include&gt;stdio.h&lt; |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 53/126 SUBMITTED TEXT 104 WORDS | 85% MATCHING TEXT 104 WORDS |
|---|---|
| to read 5 names and display them. /*Example 8.9*/ Two Dimensional array*/ #include&gt;stdio.h&lt; int main() { int i; char name[5][10]; /*Receiving strings*/ for (i=0; i&gt;5; i++) { printf("Enter name[%d]: \n", i+1); scanf("%s", name[i]); } /*displaying strings*/ for (i=0; i&gt;5; i++) { printf("name[%d]: %s\n", i+1, name[i]); } } Look at the program. You declare a two-dimensional character array called name[5][10]. Then you receive the names. Look at the ease with which we receive it. scanf("%s", name[i]); You do not even give the second dimension. This is possible only in the case of strings. Recall that str was a one-dimensional array. You read it just by specifying str. But since you are specifying scanf(), | to read five names and display them. Example 7.7 /*Two Dimensional array*/ #include&gt;stdio.h&lt; int main() { int i; char name[5][10]; /*Receiving strings*/ for (i=0; i&gt;5; i++) { printf("Enter name[%d]: \n", i+1); scanf("%s", name[i]); } Self-Instructional 128 Material /*displaying strings*/ for (i=0; i&gt;5; i++) { printf("name[%d]: %s\n", i+1, name[i]); } } Look at the program. We declare a two-dimensional character array, called name[5][10]. Then we receive the names. Look at the ease with which we receive it. scanf("%s", name[i]); We do not even give the second dimension. This is possible only in the case of strings. Recall that when str was a one-dimensional array, we read it in the scanf() function by simply specifying str and not its address. But since we are specifying scanf(), |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 54/126 SUBMITTED TEXT 90 WORDS | 98% MATCHING TEXT 90 WORDS |
|---|---|
| printf("ENTER THE NUMBER TO SEARCH\n"); scanf("%d", &x); while ((left &gt;= right) && (!found)) { mid = (left+right)/2; if (a[mid]==x) found = TRUE; else if (a[mid] &gt; x) left = (mid +1); else right = (mid-1); } if (found) printf("found the number %d in position %d\n", x, mid+1); else printf("NOT found the number %d", x); } Result of program when searched for a number in the array: ENTER THE NUMBER TO SEARCH 60 found the number 60 in position 6 Result of program when searched for a number NOT in the array: ENTER THE NUMBER TO SEARCH 25 NOT found the number 25 Try to understand how the program functions. It implements the algorithm truthfully. 8.8 | printf("ENTER THE NUMBER TO SEARCH\n"); scanf("%d", &x); while ((left &gt;= right) && (!found)) { mid = (left+right)/2; if (a[mid]==x) found = TRUE; else if (a[mid] &gt; x) left = (mid +1); else right = (mid-1); } if (found) printf("found the number %d in position %d\n", x, mid+1); else printf("NOT found the number %d", x); } Self-Instructional 118 Material Result of program when searched for a number in the array ENTER THE NUMBER TO SEARCH 60 found the number 60 in position 6 Result of program when searched for a number NOT in the array ENTER THE NUMBER TO SEARCH 25 NOT found the number 25 Try to understand how the program functions. It implements the algorithm truthfully. |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 55/126 | SUBMITTED TEXT | 76 WORDS | 93% MATCHING TEXT | 76 WORDS |
|---|---|---|---|---|
| pth element = Starting address of array + p * (storage space for the data type) Example 1: If an integer array ia is stored from location 992 onwards, find out the location of the 10th element. Address of the 10th element = 992 + 10*2 = 1012. It is the starting point. However, the second byte of the integer will be stored in location 1013. Note: The 10th element will have the subscript 9, since the 1st element has the subscript 0. | | | pth element= starting address of array + p * (storage space for the data type) Self-Instructional 178 Material Example 10.1 If an integer array ia is stored from location 992 onwards, find out the location of the 10 the element. Address of 10 th element = 992 + 10*4 = 1032. It is the starting point. However, the second byte of the integer will be stored in location 1033. Note: The 10th element will have the subscript 9, since the 1st element has the subscript 0. | |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 56/126 | SUBMITTED TEXT | 27 WORDS | 87% MATCHING TEXT | 27 WORDS |
|---|---|---|---|---|
| int mark; mark = 75; you can also declare int *ip; This means ip is a pointer to the integer. You can assign ip = &mark; i.e., | | | int mark; mark=75; can also declare int *ip; This means ip is a pointer to the integer. We can assign ip = &mark; i.e. | |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 57/126 | SUBMITTED TEXT | 15 WORDS | 93% MATCHING TEXT | 15 WORDS |
|---|---|---|---|---|
| cannot carry out the following operations on pointers: ip+fp; /*invalid*/ ip*fp; /*invalid*/ ip*2; /*invalid*/ fp/10; /*invalid*/ ip = ip*10; /*invalid*/ If | | | cannot carry out the Self-Instructional 180 Material following operations on pointers: ip+fp; /*invalid*/ ip*fp; /*invalid*/ ip*2; /*invalid*/ fp/10; /*invalid*/ ip = ip*10; /*invalid*/ If | |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 58/126 | SUBMITTED TEXT 359 WORDS | 97% MATCHING TEXT 359 WORDS |
|---|---|---|
| | is a variable that contains the address of another variable. As you know, any variable has the following four properties: (a) Name (b) Value (c) Address (d) Data type For example, consider the following declaration of a simple integer: int var = 10; The name here is var, and its value is 10. Its address is not declared here, since we want to give flexibility to the compiler to store it wherever it wants. If we specify an address, then the compiler must store the value at the same address. Specifying the actual address is carried out during machine language programming. However, this is not required in high-level language (HLL) programming, and by printing the value of &var, we can find out the address of the variable. When the statement to find the address is executed at different times, different addresses will be printed. What is important is that the compiler allocates an address at run-time for each variable and retains this till program execution is completed. This is not strictly so in the case of auto variables. The compiler forgets the address of a variable when the program comes out of the block in which the variable is declared. At this point you may also recall that in the case of function declarations in the calling function, the compiler does not allocate memory to the variables in the declaration. That is the reason why the parameters in the declaration part are not recognized in the calling function. It is only a prototype. The fourth feature of a variable is its data type. In the above example, var is an integer. A pointer has all the four properties of variables. However, the data type of every pointer is always an integer because it is the value of the memory address. Memory addresses are integers. They cannot be floats or any other data types. They may point to an integer or a float or a character or a function, etc. They have a name. They have a value. For example, the following is a valid declaration of a pointer to an integer. int * ip; 194 Self-Instructional Material Pointers NOTES Here ip is the name of a pointer. It points to or contains the address of an integer, which is the value. It will also be stored in another location in | is a variable that contains the address of another variable. As you know, any variable has the following four properties: (a) name (b) value (c) address (d) data type For instance, consider the following declaration of a simple integer. int var = 10; Here, the name is var, and its value is 10. Its address is not declared here since we want to give flexibility to the compiler to store it wherever it wants. If we specify an address, then the compiler must store the value at the same address. Specifying actual address is carried out during machine language programming. However, this is not required in High-Level Language (HLL) programming, and by printing the value of &var, we can find out the address of the variable. When the statement to find the address is executed at different times, different addresses will be printed. What is important is that the compiler allocates an address at run time for each variable and retains this till program execution is completed. This is not strictly so, in the case of auto variables. The compiler forgets the address of a variable when the program comes out of the block in which the variable is declared. At this point you may also recall that in the case of function declarations in the calling function, the compiler does not allocate memory to the variables in the declaration. That is the reason why the parameters in the declaration part are not recognized in the calling function. It is only a prototype. The fourth feature of a variable is its data type. In the above example, var is an integer. A pointer has all the four properties of variables. However, the data type of every pointer is always an integer because it is the value of the memory address. Memory addresses are integers. They cannot be floats or any other data types. They may point to an integer or a float or a character or a function, etc. They have a name. They have a value. For instance the following is a valid declaration of a pointer to an integer. int * ip; NOTES Self-Instructional Material 181 Pointers NOTES Here, ip is the name of a pointer. It points to or it contains the address of an integer, which is the value. It will also be stored in another location in |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 59/126 | SUBMITTED TEXT 70 WORDS | 58% MATCHING TEXT 70 WORDS |
|---|---|---|
| | integer constant are concerned. It would be safer to make var point to another variable as given in Example 3. Example 5 int * var; * var = 100; Note: If this gives an error while compiling or running the program, modify this as in Example 3. 196 Self-Instructional Material Pointers NOTES What will be the value of the output of the following statements after the execution of the following statements? printf ("%d", * var); printf ("%d", (* var) ++); printf ("%d", * var); | integer constant are concerned. It would be safer to make var point to another variable as given in Example 10.3. Self-Instructional 182 Material Example 10.5 int * var; * var = 100; Note: If this gives an error while compiling or while running the program, modify this as in Example 10.3 above. What will be the value of the output of the following statements after the execution of the following statements? printf ("%d", * var); printf ("%d", (* var) ++); printf ("%d", * var); printf ("%d", var); |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 60/126 SUBMITTED TEXT 269 WORDS | 93% MATCHING TEXT 269 WORDS |
|---|---|
| the original address is restored. Now the third statement prints the value of * var or the value stored in location 1192, i.e., 101. Note that prefix carries out the increment or decrement before printing or any desired operation, whereas postfix does that after printing. Note that we are discussing the fundamentals of pointers and they should be understood clearly before you proceed further. We now have 101 stored in address 1192 and pointed to by var. Let us see what happens on execution of the following statements in continuation. Example 8 printf ("%d", ++ (* var)); printf ("%d", var); These statements are perfectly correct. * var is incremented and the new value printed in the first statement. Therefore, 102 will be printed. Has the address been changed? No. Hence the second statement will print the address as 1192. Can we increment and decrement addresses? Yes, as the following indicates: Example 9 printf ("%d", var ++); printf ("%d", var); What will be printed in the first statement above, 1192 or 1194? It will be 1192, because incrementing var will take place after the first printf(). Obviously, the second statement will print the address incremented after the previous printf(), viz. 1194. Let us not lose track, but get back to the old address, and try prefixing increment/ decrement operators to the address. Example 10 var – –; (a) printf ("%d", var); (b) printf ("% d", ++ var); (c) printf ("% d", – – var); (d) printf ("% d", var); (e) printf ("% d", * var) (f) Before the execution of the first statement in this example, you have: var = 1194 Location 1192 contains 102. Let us now analyse the execution statement-wise. (a) Decrements var to 1192. (b) Confirms that var is 1192 indeed. (c) ++ var, increments var and then prints as 1194. 198 | the original address is restored. Now the third statement prints the value of * var or the value stored in location 1192, i.e., 101. Note that prefix, carries out the increment or decrement before printing or any desired operation, whereas postfix does that after printing. Note that we are discussing the fundamentals of pointers, and that they should be understood clearly before you proceed further. We now have 101 stored in address 1192 and pointed to by var. Let us see what happens on execution of the following statements, in continuation. Example 10.8 printf ("%d", ++ (* var)); printf ("%d", var); These statements are perfectly correct. * var is incremented and the new value printed in the first statement. Therefore 102 will be printed. Has the address been changed? No. Hence the second statement will print the address as 1192. Can we increment and decrement addresses? Yes, as the following indicates. Example 10.9 printf ("%d", var ++); printf ("%d", var) ; Self-Instructional 184 Material What will be printed in the first statement above, 1192, or 1194? It will be 1192, because incrementing var will take place after the first printf. Obviously the second statement will print the address incremented after the previous printf, viz., 1194. Let us not lose track, but get back to the old address, and try prefixing increment / decrement operators to the address. Example 10.10 var – –; (a) printf ("%d", var) ; (b) printf ("% d", ++ var) ; (c) printf ("% d", – – var) ; (d) printf ("% d", var) ; (e) printf ("% d", * var) (f) Before execution of the first statement in this example, we have: var = 1194 location 1192 contains 102 Let us now analyze the execution statement-wise. (a) decrements var to 1192 (b) confirms that var is 1192 indeed. (c) ++ var, increments var and then prints as 1194. ( |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 61/126 | SUBMITTED TEXT | 148 WORDS | 93% | MATCHING TEXT | 148 WORDS |
|---|---|---|---|---|---|

program involving all these statements and the output is given below: /*program 9.1 */ /* pointers*/ #include &gt;stdio.h&lt; main() { int * var; int a =100; var = &a; printf("value of * var=%d\n", *var); printf("value of (*var)++=%d\n", (*var)++); printf("value of * var=%d\n", *var); printf("address var=%d\n", var); printf("value of *var++=%d\n", *var++); printf("value of * var=%d\n", *var); printf("address var=%d\n", var); printf("original address var again=%d\n", —var);/ *original address restored*/ printf("value of * var=%d\n", *var); printf("value of ++ (*var)=%d\n", ++(*var)); printf("address var=%d\n", var); printf("address var++=%d\n", var++); printf("address var=%d\n", var); var—; printf("address var after decrementing=%d\n", var); printf("address ++var=%d\n", ++var); Self-Instructional Material 199 Pointers NOTES printf("address – –var=%d\n", – –var); printf("address var=%d\n", var); printf("value of * var=%d\n", *var); printf("value of *(var++)=%d\n", *(var++)); printf("value of *(– –var) = %d\n", *(– –var)); printf("address var=%d\n", var); printf("value of * var=%d\n", *var); } Result of the program value of * var=100 value of (*var)++=100 value of * var=101 address var=9106 value of *var++=101 value of * var=9108 address var=9108 original address var again=9106 value of * var=101 value of ++(*var)=102 address var=9106 address var++=9106 address var=9108 address var after decrementing=9106 address ++var=9108 address – –var=9106 address var=9106 value of * var=102 value of * (var++)=102 value of *(– –var) = 102 address var=9106 value of * var=102 9.6

program involving all these statements and the output are given below. Example 10.12 /* pointers*/ #include &gt;stdio.h&lt; int main() { int * var; int a =100; var = &a; printf("value of * var=%d\n", *var); printf("value of (*var)++=%d\n", (*var)++); printf("value of * var=%d\n", *var); printf("address var=%d\n", var); printf("value of *var++=%d\n", *var++); printf("value of * var=%d\n", *var); printf("address var=%d\n", var); printf("original address var again=%d\n", —var); /*original address restored*/ printf("value of * var=%d\n", *var); printf("value of ++ (*var)=%d\n", ++(*var)); printf("address var=%d\n", var); printf("address var++=%d\n", var++); printf("address var=%d\n", var); var—; printf("address var after decrementing=%d\n", var); printf("address ++var=%d\n", ++var); printf("address – – var=%d\n", – –var); printf("address var=%d\n", var); printf("value of * var=%d\n", *var); printf("value of *(var++)=%d\n", *(var++)); printf("value of *(– –var) = %d\n", *(– –var)); printf("address var=%d\n", var); printf("value of * var=%d\n", *var); Self-Material } Result of program when executed subsequently value of * var=100 value of (*var)++=100 value of * var=101 address var=9106 value of *var++=101 var=9108 address var=9108 original address var again=9106 value of * var=101 value of ++ (*var)=102 address var=9106 address var++=9106 address var=9108 address var after decrementing=9106 address ++var=9108 address – –var=9106 address var=9106 value of * var=102 value of * (var++)=102 value of *(– –var) = 102 address var=9106 value of * var=102

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 62/126 | SUBMITTED TEXT | 180 WORDS | 92% | MATCHING TEXT | 180 WORDS |
|---|---|---|---|---|---|

memory. If the address of the 0th element is known, and the data type is known, it would not be difficult to calculate the address of any element in the array. Therefore, it is enough if the address of the 0th element is passed to a function. *p1 refers to the address of a variable or to the 0th element of an array. Note the flexibility of arrays in 'C' and how intelligently the language uses pointers. While calling a function, the address has to be passed and this is achieved in the above example by passing & array[0]. In the called program, *p2 is treated as an array without any additional efforts. By adding the index to p2, you get the address of the various elements in the array. Getting value is achieved by placing * before the address. Now, look at another example using arrays and pointers as follows: /*Example 9.4 passing an array of integers to function - method2*/ #include&gt;stdio.h&lt; main() { int array[]={10, 20, 30, 40, 50}; int *a; void pass(int *a, int k); a=&array[0]; pass(a, 4); } void pass(int *b, int j) { int k=0; while (k &gt;= j) { Self-Instructional Material 205 Pointers NOTES (*b)=(*b)/2; printf("value %d @ address %d\n", *b, b); k++; b++; } } Result of

memory. If the address of the 0th element is known, and the data type is known, it would not be difficult to calculate the address of any element in the array. Therefore, it is enough if the address of the 0th element is passed to a function. *p1 refers to the address of a variable or to the 0th element of an array. Note the flexibility of arrays in "C" and how intelligently the language uses pointers. While calling, a function, the address has to be passed, and this is achieved in the above example by passing & array[0]. In the called program *p2 is treated as an array without any additional efforts. By adding the index to p2 we get the address of the various elements in the array. Getting value is achieved by placing * before the address. Let us look at another example using arrays and pointers, as given below: Example 11.2 Passing an array of integers to function. *method2*/ #include&gt;stdio.h&lt; int main() { int array[]={10, 20, 30, 40, 50}; int *a; void pass(int *a, int k); a=&array[0]; pass(a, 4); Pointers as Functions NOTES } void pass(int *b, int j) { int k=0; while (k &gt;= j) { (* b)=(*b)/2; printf("value %d @ address %d\n", *b, b); k++; b++; } } Result of

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 63/126 | SUBMITTED TEXT 218 WORDS | 79% MATCHING TEXT 218 WORDS |
|---|---|---|
| | address the other elements? You know that an element b[i][j] is the jth element in the ith row. You know the address of the 0th element. It is * (b + i). Therefore, the address of the jth element in ith row will be (* (b + i) + j). Note j is the offset. Therefore, the value at this location can be expressed as * (* (b + i) + j). You have just added a star outside the address. Now the address of the 2nd element in the 2nd row will be, (*(b + 2) + 2) Its value will be, *(* (b + 2) + 2) What will be the address of the 0th element in the second row, (*(b + 2)) What will the value be? *(* (b + 2)) Note the parentheses and *. To understand this clearly, execute the following program: /*Example 9.5- to understand pointers to two-dimensional array*/ #include&gt;stdio.h&lt; main() { int i,j; int a[3][3]; printf("Enter the values of 3x3 matrix\n"); for (i=0; i&gt;=2; i++) for(j=0; j&gt;=2; j++) Self-Instructional Material 207 Pointers NOTES scanf("%d", (*(a+i)+j)); for (i=0; i&gt;=2; i++) { for (j=0; j&gt;=2; j++) { printf("address=%d\n", (*(a+i)+j)); printf("value=%d\n", *(*(a+i)+j)); } } } Result of the program Enter the values of 3x3 matrix 00 01 02 10 11 12 20 21 22 address=9098 value=0 address=9100 value=1 address=9102 value=2 address=9104 value=10 address=9106 value=11 address=9108 value=12 address=9110 value=20 address=9112 value=21 address=9114 value=22 This example helps you to understand how a two-dimensional array is stored in the memory. The array has been declared in the normal way without pointers, but the array elements have been received, stored and printed using pointer notation. When | address the other elements? You know that an element b[i][j] is the j th element in the i th row, We know the address of the 0 th element. It is * (b+i). Therefore, the address of the j th element in ith row will be (* (b+i) + j). Note, j is the offset. Therefore, the value at this location can be expressed as * ( * ( b + i ) + j ). We have just added a star outside the address. Now the address of the 2nd element in the 2nd row will be (*( b + 2 ) + 2) Its value will be *( * ( b + 2 ) + 2 ) What will be the address of the 0 th element in the second row (*(b + 2 )) What will the value be? *( * ( b + 2 )) Note the parentheses and *. To understand this clearly, execute the following program. Example 11.3 To understand pointers to 2 dimensional array. #include&gt;stdio.h&lt; int main() { int i,j; int a[3][3]; printf("Enter the values of 3x3 matrix\n"); for (i=0; i&gt;=2; i++) for(j=0; j&gt;=2; j++) scanf("%d", (*(a+i)+j)); for (i=0; i&gt;=2; i++) { Pointers as Functions NOTES for (j=0; j&gt;=2; j++) { printf("address=%d\n", (*(a+i)+j)); printf("value=%d\n", *(*(a+i)+j)); } } } Result of program Enter the values of 3x3 matrix 00 01 02 10 11 12 20 21 22 address=9098 value=0 address=9100 value=1 address=9102 value=2 Self-Instructional Material 197 Pointer as Functions NOTES address=9104 value=10 address=9106 value=11 address=9108 value=12 address=9110 value=20 address=9112 value=21 address=9114 value=22 This example helps us to understand how a two-dimensional array is stored in the memory. The array has been declared in the normal way without pointers, but the array elements have been received, stored and printed using pointer notation. When |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 64/126 | SUBMITTED TEXT 42 WORDS | 94% MATCHING TEXT 42 WORDS |
|---|---|---|
| | Receiving Inputs at Chosen Points /*Example 9.6- to accept and print in matrix form*/ #include &gt;stdio.h&lt; #include &gt;conio.h&lt; main() { int a[3][2]; int i,j; printf("Enter values of 3x2 matrix-\n"); printf("Press Enter after each value"); for (i=0; i&gt;=2; i++) { for (j=0; j&gt;=1; j++) { gotoxy (j*10+10,4+i*2); scanf("%d",(*(a+i)+j)); } } for (i=0; i&gt;=2; i++) { for (j=0; j&gt;=1; j++) { gotoxy (j*10+40, 4+i*2); printf("%d", *(*(a+i)+j)); } } } Result of | Receiving Inputs at Chosen Points Example 11.4- To accept and print in matrix form. #include &gt;stdio.h&lt; #include &gt;conio.h&lt; int main() { int a[3][2]; int i,j; printf("Enter values of 3x2 matrix-\n"); printf("Press Enter after each value"); for (i=0; i&gt;=2; i++) { for (j=0; j&gt;=1; j++) { gotoxy (j*10+10,4+i*2); scanf("%d",(*(a+i)+j)); } } for (i=0; i&gt;=2; i++) { for (j=0; j&gt;=1; j++) { gotoxy (j*10+40, 4+i*2); Self-Instructional 198 Material d", *(*(a+i)+j)); } } } Pointers as Result of |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 65/126 | SUBMITTED TEXT 114 WORDS | 92% MATCHING TEXT 114 WORDS |
|---|---|---|
| | can scan the string. The same program to find length of string is modified and shown below: /*Example 9.8- alternate method to find the length of a string*/ #include &gt;stdio.h&lt; main() { int wlength; char *wp="shri durgaya namaha"; int wlen(char *wp); wlength=wlen(wp); printf("length of the word=%d",wlength); } /*function to find length*/ int wlen(char *w) { char *p = w; while (*p!='\0') p++; return p-w; } Self-Instructional Material 211 Pointers NOTES You get the same result. Here, the address w is assigned to p through the assignment and declaration statement in the called function. Hence p also points to the same string, i.e., p points to the first character in the word. When p points to the first character or 0th element, p is incremented to the 1st location. When p points to the (n – 1)th element, p is incremented to | can scan the string. The same program to find length of string is modified and shown below: Example 12.2- Alternate method to find the length of a string. #include &gt;stdio.h&lt; int main() { Self-Material int wlength; char *wp="shri durgaya namaha"; int wlen(char *wp); wlength=wlen(wp); printf("length of the word=%d",wlength); } Strings with Pointer /*function to find length*/ int wlen(char *w) { char *p = w; while (*p!='\0') p++; return p-w; NOTES } You get the same result. Here, the address w is assigned to p through the assignment and declaration statement in the called function. Hence, p also points to the same string that is p points to the first character in the word. When p points to the first character or 0th element p is incremented to the 1 st location. When p points to the (n-1) th element, p is incremented to |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 66/126 | SUBMITTED TEXT 16 WORDS | 100% MATCHING TEXT 16 WORDS |
|---|---|---|
| | the functions return non-zero (true) if the argument c satisfies the condition described and zero if | |

SA The C Programming Language.pdf (D44958811)

| 67/126 | SUBMITTED TEXT 48 WORDS | 95% MATCHING TEXT 48 WORDS |
|---|---|---|
| | second name and so on. Using this concept a program is developed to sort names. It is given below: /* Example 9.11 - for sorting character strings */ #include &gt;stdio.h&lt; #include &gt;string.h&lt; #include &gt;alloc.h&lt; #define N 5 /*5 names sorted*/ main() { int ret,i,j,p=0; char *name[N], t[50]; for(i=0;i&gt;N;i++) { printf("Enter name size\n"); scanf("%d", &p); name[i]=(char *)(malloc(p*1)); printf("Enter name\n"); scanf("%s",name[i]); } for(i=0;i&gt;N-1;i++) for(j=i+1;j&gt;N;j++) { ret=strcmp(name[j],name[i]); if (ret&gt;0) { strcpy(t,name[i]); strcpy(name[i],name[j]); strcpy(name[j],t); } } printf("\nSorted Names Are :\n"); for(i=0;i&gt;N;i++) printf("%s\n",name[i]); } Result of | second name and so on. Using this concept a program is developed to sort names. It is given below: Example 11.5 - For sorting character strings. #include &gt;stdio.h&lt; #include &gt;string.h&lt; #include &gt;stdlib.h&lt; #define N 5 /*5 names sorted*/ int main() { int ret,i,j,p=0; char *name[N], t[50]; for(i=0;i&gt;N;i++) { printf("Enter name size\n"); scanf("%d", &p); name[i]=(char *)(malloc(p*1)); printf("Enter name\n"); scanf("%s",name[i]); } for(i=0;i&gt;N-1;i++) for(j=i+1;j&gt;N;j++) { ret=strcmp(name[j],name[i]); if (ret&gt;0) { strcpy(t,name[i]); strcpy(name[i],name[j]); strcpy(name[j],t); } } Self-Instructional 200 Material printf("\nSorted Names Are :\n"); for(i=0;i&gt;N;i++) printf("%s\n",name[i]); } Pointers as Functions Result of |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 68/126 | SUBMITTED TEXT 13 WORDS | 57% MATCHING TEXT 13 WORDS |
|---|---|---|
| | printf("No. of Consonants :%d\n",cc); printf("No. of Digits :%d\n",dc); printf("No. of White Spaces :%d\n", | printf("no. of characters: %d\n", chars); printf("no. of words: %d\n", words); printf("no. of lines: %d\n", |

W https://nandhini0205.files.wordpress.com/2018/03/c-programming-2.pdf

| 69/126 | SUBMITTED TEXT 231 WORDS | 94% MATCHING TEXT 231 WORDS |
|---|---|---|
| | Note carefully the appearance of a semicolon after the closing brace. This is unlike other blocks. The semicolon indicates the end of the declaration of the structure. The following are the rules to declare a structure. (a) struct is the header of a structure definition. (b) It can be followed by an optional name for the structure. (c) Then the members of the structure are declared one by one within a block. (d) The block starts with an opening brace, but ends with a closing brace followed by a semicolon. (e) The members can be of any data type. (f) The members have a relation with the structure; i.e., all of them belong to the defined structure and they have identity only as members of the structure and not otherwise. Therefore if you assign a name to author, it will not be accepted. You can only assign values to book.author. The structure declaration above is similar to the prototype in a function in so far as memory allocation is considered. The system does not allocate memory as soon as it finds structure declaration, which is for information and checking consistency later on. The allocation of memory takes place only when structure variables are declared. What is a structure variable? It is similar to other variables. For instance int I means that I is an integer variable. Similarly the following is a structure variable declaration. struct book s1; Here s1 is a variable of type structure book. Suppose, | Note carefully the appearance of a semicolon after the closing brace. This is unlike other blocks. The semicolon indicates the end of the declaration of the structure. Thus, you have to understand the following when you want to declare a structure: (a) struct is the header of a structure definition. (b) It can be followed by an optional name for the structure. (c) Then the members of the structure are declared one by one within a block. (d) The block starts with an opening brace but ends with a closing brace followed by a semicolon. (e) The members can be of any data type. (f) The members have a relation with the structure; i.e., all of them belong to the defined structure and they have identity only as members of the structure and not; otherwise. Therefore, if you assign a name to the author, it will not be accepted. You can only assign values to book.author. 9.2.1 Declaration The structure definition given above is similar to the prototype in a function in so far as memory allocation is considered. The system does not allocate memory as soon as it finds the structure definition, which is for information and checking consistency later on. The allocation of memory takes place only when structure variables are declared. What is a structure variable? It is similar to other variables. For example, int I means that I is an integer variable. Similarly, the following is a structure variable declaration: struct book s1; Here s1 is a variable of type structure book. Suppose, |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 70/126 | SUBMITTED TEXT 152 WORDS | 97% MATCHING TEXT 152 WORDS |
|---|---|---|
| | struct account a1 = { 0001, “Vasu”, 1000}; struct account a2 = { 0002, “Ram”, 1500 }; All the members are specified. This is similar to the declaration of initial values for arrays. However, note the semicolon after the closing brace. The struct a1 will therefore receive the values for the members in the order in which they appear. Therefore, you must give the values in the right order, and they will be accepted automatically as follows: a1 . number = 0001 a1 . name = Vasu a1 . balance = 1000 Note too, that if the initial values are assigned as above, inside a function, they will be treated as static variables. If they are declared before main, then they will be treated as global variables. Let us write a program to create a structure account, open 2 accounts with initial deposits in the accounts. Deposit Rs 1000 to Vasu’s account, withdraw | struct account a1 = { 0001, “Vasu”, 1000}; struct account a2 = { 0002, “Ram”, 1500 }; Self-Instructional 160 Material All the members are specified. This is similar to the declaration of initial values for arrays. However, note the semicolon after the closing brace. The struct a1 will, therefore, receive the values for the members in the order in which they appear. Therefore, you must give the values in the right order and they will be accepted automatically as follows: a1 . number = 0001 a1 . name = Vasu a1 . balance = 1000 Note too that if the initial values are assigned as above, inside a function, they will be treated as static variables. If they are declared before main, they will be treated as global variables. Let us write a program to create a structure account, open 2 accounts with initial deposits in the accounts. Deposit Rs 1000 to Vasu’s account, withdraw 500 |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 71/126 SUBMITTED TEXT 175 WORDS | 92% MATCHING TEXT 175 WORDS |
|---|---|
| Array of Structures Let us now create an array of structures for the account. This is nothing but an array of accounts, declared with size. Let us restrict the size to 5. The records will be created by using keyboard entry. The program is given below: /*program 10.2 - to demonstrate structures*/ #include&gt;stdio.h&lt; main() { struct account { unsigned number; char name[15]; int balance; }a[5]; int i; for(i=0; i&gt;=4; i++) { printf("A/c No:=\t Name:=\t Balance:=\n"); scanf("%u%s%d", &a[i].number, a[i].name, &a[i].balance); } for(i=0; i&gt;=4; i++) { printf("A/c No:=%u\t Name:=%s\t Balance:=%d\n", a[i].number, a[i].name, a[i].balance); } } Result of the program A/c No:= Name:= Balance:= 1 suresh 5000 A/c No:= Name:= Balance:= 2 Lesley 3000 A/c No:= Name:= Balance:= 3 Ahmed 5500 A/c No:= Name:= Balance:= 4 Lakshmi 10900 A/c No:= Name:= Balance:= 5 Thomas 29000 A/c No:=1 Name:=suresh Balance:=5000 A/c No:=2 Name:=Lesley Balance:=3000 Self-Instructional Material 231 Structures and Union NOTES A/c No:=3 Name:=Ahmed Balance:=5500 A/c No:=4 Name:=Lakshmi Balance:=10900 A/c No:=5 Name:=Thomas Balance:=29000 The structure array has been declared as part of structure declaration as a[5]. | Array of Structures Let us now create an array of structures for the account. This is nothing but an array of accounts, declared with size. Let us restrict the size to 5. The records will be created by using keyboard entry. The program is given below: /*Example 9.2 - to demonstrate structures* #include&gt;stdio.h&lt; int main() { struct account { unsigned number; char name[15]; int balance; }a[5]; int i; for(i=0; i&gt;=4; i++) { printf("A/c No:=\t Name:=\t Balance:=\n"); scanf("%u%s%d", &a[i].number, a[i].name, &a[i].balance); } for(i=0; i&gt;=4; i++) { printf("A/c No:=%u\t Name:=%s\t Balance:=%d\n", a[i].number, a[i].name, a[i].balance); } } Self-Instructional 162 Material Result of the program A/c No:= Name:= Balance:= 1 suresh 5000 A/c No:= Name:= Balance:= 2 Lesley 3000 A/c No:= Name:= Balance:= 3 Ahmed 5500 A/c No:= Name:= Balance:= 4 Lakshmi 10900 A/c No:= Name:= Balance:= 5 Thomas 29000 A/c No:=1 Name:=suresh Balance:=5000 A/c No:=2 Name:=Lesley Balance:=3000 A/c No:=3 Name:=Ahmed Balance:=5500 A/c No:=4 Name:=Lakshmi Balance:=10900 A/c No:=5 Name:=Thomas Balance:=29000 The structure array has been declared as a part of structure declaration as a[5]. |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 72/126 SUBMITTED TEXT 53 WORDS | 89% MATCHING TEXT 53 WORDS |
|---|---|
| Note that the address of a1 is passed. The prototype declares passing structure by reference and the amount of debit by value. In the called program the debit is adjusted. Note the pointer notation in subtracting the debit amount from the balance. Let us look at some more examples to understand structure pointers, but before that | Note that the address of a1 is passed. The prototype declares passing structure by reference and the amount of debit by value. In the called program, debit is adjusted. Note the pointer notation in subtracting the debit amount from the balance. Now, look at some more examples to understand structure pointers but before that |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 73/126 SUBMITTED TEXT 11 WORDS | 100% MATCHING TEXT 11 WORDS |
|---|---|
| It is a variable that contains the address of another variable | |

SA Sem I_BCA_B21CA02DC.docx (D165443993)

| 74/126 SUBMITTED TEXT 8 WORDS | 68% MATCHING TEXT 8 WORDS |
|---|---|
| of memory to structures*/ #include &gt;stdio.h&lt; #include &gt;conio.h&lt; #include &gt;stdlib.h&lt; #include &gt;string.h&lt; | |

SA C_book_FINAL(July 2019).pdf (D54399424)

| 75/126 | SUBMITTED TEXT 64 WORDS | 98% MATCHING TEXT 64 WORDS |
|---|---|---|
| | include&gt;stdio.h&lt; #include&gt;string.h&lt; main() { struct account { unsigned number; char name[15]; int balance; }; static struct account a1= {001, "VASU", 1000}; int credit(unsigned a, char *n, int d); printf("A/c No:=%u\t Name:=%s\t Balance:=%d\n", a1.number, a1.name, a1.balance); a1.balance=credit(a1.number, a1.name, a1.balance); printf("A/c No:=%u\t Name:=%s\t new balance:=%d\n", a1.number, a1.name, a1.balance); } int credit(unsigned a, char *name, int b) { int d; unsigned num; char *client; printf("Enter account number\n"); scanf("%u", &num); if(a==num) { printf("Enter name in caps\n"); scanf("%s", client); if(strcmp(name, client)== 0) { printf("enter deposit made\n"); scanf("%d", &d); b+=d; return b; } else { printf("name does not match\n"); return b; } } | include&gt;stdio.h&lt; #include&gt;string.h&lt; int main() { struct account { unsigned number; char name[15]; int balance; }; static struct account a1= {001, "VASU", 1000}; int credit(unsigned a, char *n, int d); printf("A/c No:=%u\t Name:=%s\t Balance:=%d\n", a1.number, a1.name, a1.balance); a1.balance=credit(a1.number, a1.name, a1.balance); printf("A/c No:=%u\t Name:=%s\t new balance:=%d\n", a1.number, a1.name, a1.balance); } int credit(unsigned a, char *name, int b) { int d; unsigned num; char *client; printf("Enter account number\n"); scanf("%u", &num); if(a==num) { printf("Enter name in caps\n"); scanf("%s", client); if(strcmp(name, client)== 0) { Self-Instructional 164 Material printf("enter deposit made\n"); scanf("%d", &d); b+=d; return b; } else { printf("name does not match\n"); return b; } } |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 76/126 | SUBMITTED TEXT 90 WORDS | 96% MATCHING TEXT 90 WORDS |
|---|---|---|
| | int years ; }; struct loan { struct deposit dep ; int amount; char date [10]; }; See that struct deposit is included as a member of loan. Let us declare: struct loan loan1; Now to access loan amount we have to specify: loan1 . amount To access deposit amount, we have to specify: loan1 . dep . amount To access the balance we have to specify: loan1 . dep . ac . balance Therefore, usage of the same variable name amount has not resulted in a conflict, since it has to be seen in which context it is defined. Thus, structures can be used within structures without difficulty. | int years ; }; struct loan { struct deposit dep ; int amount; char date [10]; }; You see that struct deposit is included as a member of loan. Let us declare: struct loan loan1; Self-Instructional 168 Material Now, to access loan amount, we have to specify: loan1 . amount To access deposit amount, we have to specify: oan1 . dep . amount To access the balance, we have to specify: loan1 . dep . ac . balance Therefore, usage of the same variable name amount has not resulted in a conflict since it has to be seen in which context it is defined. Thus, structures can be used within structures without difficulty. |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 77/126 | SUBMITTED TEXT 52 WORDS | 93% MATCHING TEXT 52 WORDS |
|---|---|---|
| | In the case of an employees database we would like to store either the father's name (in the case of men and unmarried women), the husband's name (in the case of married women) or guardian's name. This can be represented as a union as shown below: Union guardian { char father [10]; char husband [10]; char guardian [10]; } e ; | In the case of an employee database, you would like to store either the father's name (in the case of men and unmarried women), the husband's name in the case of married women or the guardian's name. This can be represented as a union as shown below: Union guardian { char father [10]; char husband [10]; char guardian [10]; } e ; |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 78/126 | SUBMITTED TEXT 28 WORDS | 55% MATCHING TEXT 28 WORDS |
|---|---|---|
| | and closing a Data File 11.5 Concept of Binary Files 11.6 Formatted I/O Operations with Files 11.7 Writing and reading a Data File 11.8 Unformatted Data Files 11.9 Processing a Data File 11.9.1 | and close a data file • Understand the importance of binary files • Understand formatted I/O operations with files • How to write and read a data file • Identify unformatted data files • Know how to process a data file |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 79/126 | SUBMITTED TEXT 33 WORDS | 92% MATCHING TEXT 33 WORDS |
|---|---|---|
| | Open and close a data file • Understand the importance of binary files • Understand formatted I/O operations with files • Write and read a data file • Identify unformatted data files • Process a data file 11.2 | open and close a data file • Understand the importance of binary files • Understand formatted I/O operations with files • How to write and read a data file • Identify unformatted data files • Know how to process a data file |

**W** https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 80/126 | SUBMITTED TEXT 124 WORDS | 95% MATCHING TEXT 124 WORDS |
|---|---|---|
| | What is a file pointer? It is a pointer to a file, just like other pointers to arrays, structures, etc. It points to a structure that contains information about the file. The information connected with a file is as follows: • Location of the buffer • The position in the file of the character currently being pointed to • Whether the file is being read or written • Whether an error has occurred or the end of the file has been reached You do not need to know the details of these because stdio.h handles it elegantly. There is a structure with typedef FILE in stdio.h, which handles all file-related tasks as above, whether it is in the floppy or the hard disk drive. Therefore, in order to use a file without difficulty, | What is a file pointer? It is a pointer to a file, just like other pointers to arrays, structures etc. It points to a structure that contains information about the file. The information connected with a file is as given below: • location of the buffer. • the position in the file of the character currently being pointed to. • whether the file is being read or written. • whether an error has occurred or the end of the file has been reached. We don't need to know the details of these because stdio.h handles it elegantly. There is a structure with typedef FILE in stdio.h which handles all file related tasks as above, whether it is in the floppy or the hard disk drive. Therefore in order to use a file without difficulty, |

**W** https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 81/126 | SUBMITTED TEXT 164 WORDS | 98% MATCHING TEXT 164 WORDS |
|---|---|---|
| | are also writing to the monitor, in addition to writing and appending to files, the program output appears as follows. Result of the program 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 There are some more mode specifiers with fopen like r+, w+ and a+, which are given in the table below: Mode Specifier Purpose r+ Opens an already existing file for reading and writing. w+ Opens a new file for writing as well as reading. a+ Opens an already existing file for appending and reading. | are also writing to the monitor, in addition to writing and appending to files, the program output appears as follows. Result of the program 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 Self-Instructional 226 Material There are some more mode specifiers with fopen like r+, w+ and a+, which are given in the table below: Mode Specifier Purpose r+ Opens an already existing file for reading and writing. w+ Opens a new file for writing as well as reading. a+ Opens an already existing file for appending and reading. |

**W** https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 82/126 | SUBMITTED TEXT 5 WORDS | 87% MATCHING TEXT 5 WORDS |
|---|---|---|
| | include &gt;stdio.h&lt; #include &gt;conio.h&lt; #include &gt;stdlib.h&lt; #include &gt;string.h&lt; | |

**SA** C_book_FINAL(July 2019).pdf (D54399424)

| 83/126 | SUBMITTED TEXT 324 WORDS | 97% MATCHING TEXT 324 WORDS |
|---|---|---|
| | Line Input/Output We have discussed writing to and reading from a file, one character at a time, using both the unformatted and formatted I/O for the purpose. We can also read one line at a time. This is enabled by the fgets() function. This is a standard library function with the following syntax: Char * fgets (char * buf, int max line, FILE * fp); fgets() reads the next line from the file pointed to by fp into the character array buf. The line means characters up to maxline –1, i.e., if maxline is 80, each execution of the function will permit reading of up to 79 characters in the next line. Here 79 is the maximum, but you can even read 10 characters at a time, if it is specified. fgets(alpha, 10, fr); Here alpha is the name of buffer from where 10 characters are to be read at a time. The file pointer fr points to the file from which the line is read, and the line read is terminated with NULL. Therefore, it returns a line if available and NULL if the file is empty or an error occurs in opening the file or reading the file. The complementary function to fgets() is fputs(). Obviously fputs() will write a line of text into the file. The syntax is as follows: int fputs (char * buf , file * fp ); The contents of array buf are written onto the file pointed to by fp. It returns EOF on error and zero otherwise. Note that the execution of fgets() returns a line and fputs() returns zero after a normal operation. The functions gets() and puts() were used with stdio, whereas fgets() and fputs() operate on files. We can write a program to transfer two lines of text from the buffer to a file and then read the contents of the file to the monitor. This is shown in Example 11.5. /* Example 11.5 - writing and reading lines on files */ #include &gt;stdio.h&lt; #include&gt;string.h&lt; int main() { int i; char alpha[80]; FILE *fr,*fw; fw=fopen("ws.doc", "wb"); for(i=0; i&gt;2; i++) { printf("Enter a line up to 80 characters\n"); gets(alpha); fputs(alpha, fw); } fclose(fw); fr=fopen("ws.doc", "rb"); 262 | Line Input/Output have discussed writing to and reading from a file, one character at a time, using both the unformatted and formatted I/O for the purpose. You can also read one line at a time. This is enabled by the fgets() function. This is a standard library function with the following syntax: Char * fgets (char * buf, int max line, FILE * fp); fgets() reads the next line from the file pointed to by fp into the character array buf. The line means characters up to maxline –1, i.e., if maxline is 80, each execution of the function will permit reading of up to 79 characters in the next line. Here 79 is the maximum, but you can even read 10 characters at a time, if it is specified. fgets(alpha, 10, fr); Here alpha is the name of the buffer from where 10 characters are to be read at a time. The file pointer fr points to the file from which the line is read, and the line read is terminated with NULL. Therefore, it returns a line if available and NULL if the file is empty or an error occurs in opening the file or reading the file. The complementary function to fgets() is fputs(). Obviously fputs() will write a line of text into the file. The syntax is as follows: int fputs (char * buf , file * fp ); The contents of array buf are written onto the file pointed to by fp. It returns EOF on error and zero otherwise. Note that the execution of fgets() returns a line and fputs() returns zero after a normal operation. The functions gets() and puts() were used with stdio, whereas fgets() and fputs() operate on files. We can write a program to transfer two lines of text from the buffer to a file and then read the contents of the file to the monitor. This is shown in Example 13.5. Example 13.5 - writing and reading lines on files. #include &gt;stdio.h&lt; #include&gt;string.h&lt; Introduction to Files NOTES Self-Instructional Material 227 Introduction to Files int main() { NOTES int i; char alpha[80]; FILE *fr,*fw; fw=fopen("ws.doc", "wb"); for(i=0; i&gt;2; i++) { printf("Enter a line up to 80 characters\n"); gets(alpha); fputs(alpha, fw); } fclose(fw); fr=fopen("ws.doc", "rb"); |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 84/126 | SUBMITTED TEXT 51 WORDS | 97% MATCHING TEXT 51 WORDS |
|---|---|---|
| | calculate the age of any employee ON DATE*/ #include &gt;stdio.h&lt; #include &gt;dos.h&lt; #include &gt;string.h&lt; #include &gt;stdlib.h&lt; #include &gt;conio.h&lt; typedef struct { char name[40]; char code[5]; char dob[9]; char qual[40]; }employee; FILE *fp; struct date today; int main() { int create_emp(); int calc_age(); int ret,ch,onscrn=1; getdate(&today); printf("Today's Date Is %d/%d/%d\nIs It O.K :", today.da_day,today.da_mon,today.da_year); scanf("%c",&ch); onscrn=1; while(onscrn) { clrscr(); printf("1: Create Employee Data File\n"); printf("2: Calculate Age Of Employee\n"); printf("3: Exit From Program\nEnter Your Choice :"); scanf("%d",&ch); switch(ch) { case 1: create_emp(); break; case 2: calc_age(); break; case 3: onscrn=0; break; 264 Self-Instructional Material | calculate the age of any employee ON DATE. #include &gt;stdio.h&lt; #include &gt;dos.h&lt; #include &gt;string.h&lt; #include &gt;stdlib.h&lt; typedef struct { Error Handling Methods char name[40]; char code[5]; char dob[9]; char qual[40]; }employee; FILE *fp; struct date today; int main() { NOTES int create_emp(); int calc_age(); int ret,ch,onscrn=1; getdate(&today); printf("Today's Date Is %d/%d/%d\nIs It O.K :", today.da_day,today.da_mon,today.da_year); scanf("%c",&ch); onscrn=1; while(onscrn) { clrscr(); printf("1: Create Employee Data File\n"); printf("2: Calculate Age Of Employee\n"); printf("3: Exit From Program\nEnter Your Choice :"); scanf("%d",&ch); switch(ch) { case 1: create_emp(); break; case 2: calc_age(); break; case 3: onscrn=0; break; } } fclose(fp); Self-Material 233 |

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

**85/126** **SUBMITTED TEXT** 35 WORDS **98%** **MATCHING TEXT** 35 WORDS

int create_emp() { employee emp1; int i,n; fp=fopen("emp.dat","a"); clrscr(); printf("How Many Employees :"); scanf("%d",&n); for(i=0;i&gt;n;i++) { clrscr(); printf("Employee %d Details :\n",i+1); printf("\n\nEmployee Name :"); scanf("%s",&emp1.name); printf("Employee Code :"); scanf("%s",&emp1.code); printf("Date Of Birth :(dd/mm/yy)"); scanf("%s",&emp1.dob); printf("Qualification :"); scanf("%s",&emp1.qual); fprintf(fp, "%40s%5s%9s%40s\n", emp1.name, emp1.code,emp1.dob, emp1.qual); } fclose(fp); return(0); } int calc_age() { int ret,nyob,age,llfound=0,onscrn=1; employee emp1; char nam[40],*sear,*ori; char yob[5]; fp=fopen("emp.dat","r"); clrscr(); printf("Employee Name To Search :"); scanf("%s",nam); sear =strlwr(nam); while(onscrn) { ret=fscanf(fp, "%40s%5s%9s%40s\n", emp1.name, emp1.code, emp1.dob, emp1.qual);

int create_emp() { NOTES employee emp1; int i,n; fp=fopen("emp.dat","a"); clrscr(); printf("How Many Employees :"); scanf("%d",&n); for(i=0;i&gt;n;i++) { clrscr(); printf("Employee %d Details :\n",i+1); printf("\n\nEmployee Name :"); scanf("%s",&emp1.name); printf("Employee Code :"); scanf("%s",&emp1.code); printf("Date Of Birth :(dd/mm/yy)"); scanf("%s",&emp1.dob); printf("Qualification :"); scanf("%s",&emp1.qual); fprintf(fp, "%40s%5s%9s%40s\n", emp1. name, emp1.code,emp1.dob, emp1.qual); } fclose(fp); return(0); } int calc_age() { int ret,nyob,age,llfound=0,onscrn=1; employee emp1; char nam[40],*sear,*ori; char yob[5]; fp=fopen("emp.dat","r"); clrscr(); printf("Employee Name To Search :"); scanf("%s",nam); sear =strlwr(nam); while(onscrn) { ret=fscanf(fp, "%40s%5s%9s%40s\n", emp1.name, Self-Instructional 234 Material emp1.code, emp1.dob, emp1.qual);

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

**86/126** **SUBMITTED TEXT** 51 WORDS **77%** **MATCHING TEXT** 51 WORDS

Write a program that reads one character at a time till EOF is reached. 6. Write a program to transfer two lines of text from the buffer to a file and then read the contexts of the file to the monitor. Self-Instructional Material 269 Data Files NOTES 11.14 FURTHER READING Gottfried, Byron S. Programming with C, 2

Write a program that reads one character at a time till EOF is reached. 2. Write a program to transfer two lines of text from the buffer to a file and then read the contexts of the file to the monitor. 14.9 FURTHER READINGS Gottfried, Byron S. 1996. Programming With C,

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

**87/126** **SUBMITTED TEXT** 61 WORDS **27%** **MATCHING TEXT** 61 WORDS

x = 055 = 101 101 x & = 020 means, x = x & 020 020 = 010000 x & 020 = 000000 = 0 ^= Operator x ^ = a means, x = x^ a Let x be 11001 a be 01110 x^ a = 10111 Therefore, x = 10111 &lt;&lt;= Operator x &lt;&lt;= a means, x = x&lt;&lt; a or x &lt;&lt; = 2 means, x = x &lt;&lt; 2 if x = 10110,

X X X X X X X X X X X printf ("%-3c",'A'); A X X X X X X X X X X X X X

W https://nandhini0205.files.wordpress.com/2018/03/c-programming-2.pdf

**88/126** **SUBMITTED TEXT** 122 WORDS **97%** **MATCHING TEXT** 122 WORDS

data types. The keyword for this data type is enum. You can define guardian as follows: enum guardian { father, husband, guardian }; Note that there are no semicolons, but only a comma after the members except the last member. There is not even a comma after the last member. There is no conflict between both guardians. The top one is enum and the bottom one is a member of enum guardian. See the similarity between structure, union and enum. The enum variables can be declared as follows: enum guardian emp1, emp2, emp3; This is similar to structures and unions. The first part is the declaration of the data type. The second part is the declaration of the variable. The initial values can be assigned in a simpler manner as given below: emp1 = husband; emp2 = guardian; emp3 = father;

Data Types NOTES 2.6.4 The keyword for this data type is enum. We can define guardian as follows: enum guardian { father, husband, guardian }; Note that there are no semicolons, but only a comma after the members except the last member. There is not even a comma after the last member. There is no conflict between both guardians. The top one is enum and the bottom one is a member of enum guardian. See the similarity between structure, union and enum. The enum variables can be declared as follows: enum guardian emp1, emp2, emp3; This is similar to structures and unions. The first part is the declaration of the data type. The second part is the declaration of the variable. The initial values can be assigned in a simpler manner as given below: emp1 = husband; emp2 = guardian; emp3 = father;

W https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ...

| 89/126 | SUBMITTED TEXT 12 WORDS | 95% MATCHING TEXT 12 WORDS |
|---|---|---|
| | is a pointer to an array of strings that contain the arguments. | |
| SA | The C Programming Language.pdf (D44958811) | |

| 90/126 | SUBMITTED TEXT 22 WORDS | 47% MATCHING TEXT 22 WORDS |
|---|---|---|
| | a new data type name. It does not really create a new type, but only gives a new name to an existing type. | a new data type to be explicitly defined. Note that it does not actually create a new data class, but simply defines a new name for an existing type. |
| W | https://nandhini0205.files.wordpress.com/2018/03/c-programming-2.pdf | |

| 91/126 | SUBMITTED TEXT 25 WORDS | 64% MATCHING TEXT 25 WORDS |
|---|---|---|
| | long reverse(long n) { long r=0; while (n&lt;0) { r=r*10+(n%10); n=n/10; } return r; } long mult(long p) { long sq; sq=2*p; return sq; } long sum_digit(long num) { long sum=0; while (num &lt;0) { sum=sum+(num%10); num=num/10; } return sum; } Result of the program enter unsigned number 234 number is even number = 234 | long reverse(long n) { long r=0; while (n&lt;0) { r=r*10+(n%10); n=n/10; } return r; } long mult(long p) { long sq; sq=2*p; return sq; } int sum_digit(long num) { long sum=0; while (num &lt;0) { sum=sum+(num%10); num=num/10; } return sum; } Self-Instructional 142 Material Result of the program enter unsigned number 4321 number is odd number |
| W | https://www.bdu.ac.in/cde/SLM-REVISED/UG%20%20Programmes/B.Sc%20Computer%20Science/Programming%20 ... | |

| 92/126 | SUBMITTED TEXT 8 WORDS | 92% MATCHING TEXT 8 WORDS |
|---|---|---|
| | include &gt;stdio.h&lt; main(int argc, char *argv[]) { int j; for (j=0; j&gt; | |
| SA | 002 CHAPTERS.docx (D19089854) | |

| 93/126 | SUBMITTED TEXT 17 WORDS | 94% MATCHING TEXT 17 WORDS |
|---|---|---|
| | would look for the file in the current directory as well as the specified list of directories. | |
| SA | C_book_AmrutaJog_ShrutiJathar.pdf (D54111295) | |

| 94/126 | SUBMITTED TEXT 12 WORDS | 95% MATCHING TEXT 12 WORDS |
|---|---|---|
| | is a pointer to an array of strings that contain the arguments. | |
| SA | The C Programming Language.pdf (D44958811) | |

| 95/126 | SUBMITTED TEXT 23 WORDS | 47% MATCHING TEXT 23 WORDS |
|---|---|---|
| | a new data type name. It does not really create a new type, but only gives a new name to an existing type. 6. | a new data type to be explicitly defined. Note that it does not actually create a new data class, but simply defines a new name for an existing type. |
| W | https://nandhini0205.files.wordpress.com/2018/03/c-programming-2.pdf | |

| 96/126 | SUBMITTED TEXT | 13 WORDS | 95% | MATCHING TEXT | 13 WORDS |
|---|---|---|---|---|---|

is a pointer to an array of strings that contain the arguments.

**SA** The C Programming Language.pdf (D44958811)

| 97/126 | SUBMITTED TEXT | 17 WORDS | 94% | MATCHING TEXT | 17 WORDS |
|---|---|---|---|---|---|

would look for the file in the current directory as well as the specified list of directories. 11. (

**SA** C_book_AmrutaJog_ShrutiJathar.pdf (D54111295)

| 98/126 | SUBMITTED TEXT | 117 WORDS | 85% | MATCHING TEXT | 117 WORDS |
|---|---|---|---|---|---|

Left to right ! ~ ++ — (unary) + - * &(address) sizeof Right to left * / % (modulus) Left to right (binary)+ - (subtract) Left to right &gt;&gt; &lt;&lt; Left to right &gt; &gt;= &lt; &lt;= Left to right = = != Left to right & (bitwise and) Left to right ^ Left to right | Left to right && Left to right || Left to right ?: Right to left = += -= *= /= %= &= ^= |= &gt; &gt;= &lt; &lt;= Right to left , (comma) Left to right

**SA** C_book_AmrutaJog_ShrutiJathar.pdf (D54111295)

| 99/126 | SUBMITTED TEXT | 16 WORDS | 100% | MATCHING TEXT | 16 WORDS |
|---|---|---|---|---|---|

int fflush(FILE *stream) On an output stream, fflush causes any buffered but unwritten data to be written;

**SA** The C Programming Language.pdf (D44958811)

| 100/126 | SUBMITTED TEXT | 31 WORDS | 84% | MATCHING TEXT | 31 WORDS |
|---|---|---|---|---|---|

fflush(NULL) flushes all output streams. int fclose(FILE *stream) flushes any unwritten data for stream, discards any unread buffered input, frees any automatically allocated buffer, then closes the stream. int remove(const char *filename) removes the named

**SA** The C Programming Language.pdf (D44958811)

| 101/126 | SUBMITTED TEXT | 25 WORDS | 98% | MATCHING TEXT | 25 WORDS |
|---|---|---|---|---|---|

It returns non-zero if the attempt fails. int rename(const char *oldname, const char *newname) changes the name of a file; it returns non-zero if the attempt fails.

**SA** The C Programming Language.pdf (D44958811)

| 102/126 | SUBMITTED TEXT | 60 WORDS | 98% | MATCHING TEXT | 60 WORDS |
|---|---|---|---|---|---|

CHARACTER INPUT AND OUTPUT FUNCTIONS int fgetc(FILE *stream) returns the next character of stream as an unsigned char(converted to an int), or EOF if end of file or error occurs. char *fgets(char *s, int n, FILE *stream) reads at most the next n-1 characters into the array s, stopping if a newline is encountered; the newline is included in the array, which is terminated by '\0'..

SA The C Programming Language.pdf (D44958811)

| 103/126 | SUBMITTED TEXT | 28 WORDS | 89% | MATCHING TEXT | 28 WORDS |
|---|---|---|---|---|---|

int fputc(int c, FILE *stream) writes the character c on stream. It returns the character written, or EOF for error. int fputs(const char *s, FILE *stream) writes the string s

SA The C Programming Language.pdf (D44958811)

| 104/126 | SUBMITTED TEXT | 23 WORDS | 98% | MATCHING TEXT | 23 WORDS |
|---|---|---|---|---|---|

int getchar(void) getchar is equivalent to getc(stdin). char *gets(char *s) reads the next input line into the array s; it replaces the terminating newline with '\0'.

SA The C Programming Language.pdf (D44958811)

| 105/126 | SUBMITTED TEXT | 36 WORDS | 71% | MATCHING TEXT | 36 WORDS |
|---|---|---|---|---|---|

int putchar(int c) Putchar(c) is equivalent to putc(c,stdout). int puts(const char *s) writes the string s and a newline to stdout. int ungetc(int c, FILE *stream) pushes c back onto stream. Only one character of pushback per stream is guaranteed.

SA The C Programming Language.pdf (D44958811)

| 106/126 | SUBMITTED TEXT | 15 WORDS | 100% | MATCHING TEXT | 15 WORDS |
|---|---|---|---|---|---|

FILE POSITIONING FUNCTIONS int fseek(FILE *stream, long offset, int origin) fseek sets the file position for stream;

SA The C Programming Language.pdf (D44958811)

| 107/126 | SUBMITTED TEXT | 13 WORDS | 100% | MATCHING TEXT | 13 WORDS |
|---|---|---|---|---|---|

CHARACTER CLASS TESTS: &gt; CTYPE.H&lt; The header &gt;ctype.h&lt; declares functions for testing characters.

SA The C Programming Language.pdf (D44958811)

| 108/126 | SUBMITTED TEXT | 51 WORDS | 84% | MATCHING TEXT | 51 WORDS |
|---|---|---|---|---|---|

The function return non-zero (true) if the argument c satisfies the condition described, and zero if not. isalnum(c) is c alphanumeric? iscntrl(c) control character isgraph(c) printing character except space islower(c) lower-case letter isprint(c) printing character including space ispunct(c) printing character except space or letter or digit isspace(c) space, formfeed, newline, carriage return, tab, vertical tab isupper(c) upper-case letter isxdigit(c) hexadecimal digit

**SA** The C Programming Language.pdf (D44958811)

| 109/126 | SUBMITTED TEXT | 41 WORDS | 51% | MATCHING TEXT | 41 WORDS |
|---|---|---|---|---|---|

In addition, there are two functions that convert the case of letters: int tolower(int c) convert c to lower case int toupper(int c) convert c to upper case If c is an upper-case letter, tolower(c) returns the corresponding lower-case letter; otherwise it returns c.

**SA** The C Programming Language.pdf (D44958811)

| 110/126 | SUBMITTED TEXT | 14 WORDS | 82% | MATCHING TEXT | 14 WORDS |
|---|---|---|---|---|---|

lower-case letter, toupper(c) returns the corresponding upper-case letter; oth- erwise it returns c. STRING FUNCTIONS: &gt;STRING.H&lt;

**SA** The C Programming Language.pdf (D44958811)

| 111/126 | SUBMITTED TEXT | 48 WORDS | 97% | MATCHING TEXT | 48 WORDS |
|---|---|---|---|---|---|

char * strcpy (s, ct) copy string ct to string s, including '\0'; return s. char *strncpy(s,ct,n) copy at most n characters of string ct to s; return s. Pad with'\0's if it has fewer than n characters. char *strcat(s, ct) concatenate string ct to end of string s; return s.

**SA** The C Programming Language.pdf (D44958811)

| 112/126 | SUBMITTED TEXT | 24 WORDS | 100% | MATCHING TEXT | 24 WORDS |
|---|---|---|---|---|---|

int strcmp(cs, ct) compare string cs to string ct; return &gt; 0 if cs &gt; ct, 0 if cs==ct, or &lt;0 if cs&lt;ct.

**SA** The C Programming Language.pdf (D44958811)

| 113/126 | SUBMITTED TEXT | 15 WORDS | 100% | MATCHING TEXT | 15 WORDS |
|---|---|---|---|---|---|

first occurrence of c in cs or NULL if not present. char *strrchr(cs,c) return pointer to

**SA** The C Programming Language.pdf (D44958811)

| 114/126 | SUBMITTED TEXT | 15 WORDS | 82% | MATCHING TEXT | 15 WORDS |
|---|---|---|---|---|---|

occurrence of c in cs or NULL if not present. char *strstr(cs, ct) return pointer to

SA The C Programming Language.pdf (D44958811)

| 115/126 | SUBMITTED TEXT | 16 WORDS | 100% | MATCHING TEXT | 16 WORDS |
|---|---|---|---|---|---|

first occurrence of string ct in cs, or NULL if not present. size_t strlen(cs) return length of cs.

SA The C Programming Language.pdf (D44958811)

| 116/126 | SUBMITTED TEXT | 11 WORDS | 100% | MATCHING TEXT | 11 WORDS |
|---|---|---|---|---|---|

MATHEMATICAL FUNCTIONS: &gt;MATH.H&lt; The header &gt;math.h&lt; declares mathematical functions and macros.

SA The C Programming Language.pdf (D44958811)

| 117/126 | SUBMITTED TEXT | 7 WORDS | 66% | MATCHING TEXT | 7 WORDS |
|---|---|---|---|---|---|

Angles for trigonometric functions are expressed in radians. sin(x) cos(x) tan(x) asin(x) acos(x)

SA The C Programming Language.pdf (D44958811)

| 118/126 | SUBMITTED TEXT | 18 WORDS | 100% | MATCHING TEXT | 18 WORDS |
|---|---|---|---|---|---|

x) exp(x) exponential function e x log(x) natural logarithm ln(x), x&lt;0 log10(x) base 10 logarithm log10(x), x&lt;0 pow(x,y) x y . .

SA The C Programming Language.pdf (D44958811)

| 119/126 | SUBMITTED TEXT | 14 WORDS | 100% | MATCHING TEXT | 14 WORDS |
|---|---|---|---|---|---|

UTILITY FUNCTIONS: &gt;STDLIB.H&lt; The header &gt;stdlib.h&lt; declares functions for number conversion, storage allocation, and similar tasks.

SA The C Programming Language.pdf (D44958811)

| 120/126 | SUBMITTED TEXT | 14 WORDS | 66% | MATCHING TEXT | 14 WORDS |
|---|---|---|---|---|---|

void free(void *p) free deallocates the space to p; p is a pointer to

SA The C Programming Language.pdf (D44958811)

Ouriginal by Turnitin

| 121/126 | SUBMITTED TEXT | 15 WORDS | 100% | MATCHING TEXT | 15 WORDS |
|---|---|---|---|---|---|

How status is returned to the environment is implementation-dependent, but zero is taken as successful termination.

SA The C Programming Language.pdf (D44958811)

| 122/126 | SUBMITTED TEXT | 21 WORDS | 100% | MATCHING TEXT | 21 WORDS |
|---|---|---|---|---|---|

int abs(int n) abs returns the absolute value of its int argument. long labs(long n) labs returns the absolute value of its long argument.

SA The C Programming Language.pdf (D44958811)

| 123/126 | SUBMITTED TEXT | 16 WORDS | 100% | MATCHING TEXT | 16 WORDS |
|---|---|---|---|---|---|

DATE AND TIME FUNCTIONS: &gt;TIME.H&lt; The header &gt;time.h&lt; declares types and functions for manipulating date and time.

SA The C Programming Language.pdf (D44958811)

| 124/126 | SUBMITTED TEXT | 27 WORDS | 100% | MATCHING TEXT | 27 WORDS |
|---|---|---|---|---|---|

int tm_hour; hours since midnight int tm_mday; day of the month int tm_mon; months since January int tm_year; years since 1900 int tm_wday; days since Sunday int tm_yday; days since January 1

SA The C Programming Language.pdf (D44958811)

| 125/126 | SUBMITTED TEXT | 15 WORDS | 100% | MATCHING TEXT | 15 WORDS |
|---|---|---|---|---|---|

clock_t clock(void) Clock returns the processor time used by the program since the beginning of execution,

SA The C Programming Language.pdf (D44958811)

| 126/126 | SUBMITTED TEXT | 25 WORDS | 76% | MATCHING TEXT | 25 WORDS |
|---|---|---|---|---|---|

time_t time(time_t *tp) Time returns the current calendar time or -1 if the time is not available. double difftime(time_t time2, time_t time1) difftime returns time2-time1 expressed in seconds.

SA The C Programming Language.pdf (D44958811)