Elementary Programming with C

Elementary Programming with C

© 2004 Aptech Limited

All rights reserved

No part of this book may be reproduced in any manner
whatsoever, stored in a retrieval system or transmitted
or translated in any form or manner or by any means,
without the prior written permission of APTECH LIMITED.

All trademarks acknowledged.

APTECH LIMITED
Aptech House,
A-65, MIDC,
Andheri (East),
Mumbai - 400 093.

Dear Learner,

We congratulate you on your decision to pursue an Aptech Worldwide course.

Aptech Ltd. designs its courses using a sound instructional design model – from conceptualization to execution, incorporating the following key aspects:

- Scanning the user system and needs assessment

Needs assessment is carried out to find the educational and training needs of the learner

Technology trends are regularly scanned and tracked by core teams at Aptech Ltd. TAG* analyzes these on a monthly basis to understand the emerging technology training needs for the Industry.

An annual Industry Recruitment Profile Survey# is conducted during August - October to understand the technologies that Industries would be adapting in the next 2 to 3 years. An analysis of these trends & recruitment needs is then carried out to understand the skill requirements for different roles & career opportunities.

The skill requirements are then mapped with the learner profile (user system) to derive the Learning objectives for the different roles.

- Needs analysis and design of curriculum

The Learning objectives are then analyzed and translated into learning tasks. Each learning task or activity is analyzed in terms of knowledge, skills and attitudes that are required to perform that task. Teachers and domain experts do this jointly. These are then grouped in clusters to form the subjects to be covered by the curriculum.

In addition, the society, the teachers, and the industry expect certain knowledge and skills that are related to abilities such as *learning-to-learn, thinking, adaptability, problem solving, positive attitude etc*. These competencies would cover both cognitive and affective domains.

A precedence diagram for the subjects is drawn where the prerequisites for each subject are graphically illustrated. The number of levels in this diagram is determined by the duration of the course in terms of number of semesters etc. Using the precedence diagram and the time duration for each subject, the curriculum is organized.

- Design & development of instructional materials

The content outlines are developed by including additional topics that are required for the completion of the domain and for the logical development of the competencies identified. Evaluation strategy and scheme is developed for the subject. The topics are arranged/organized in a meaningful sequence.

The detailed instructional material – Training aids, Learner material, reference material, project guidelines, etc.- are then developed. Rigorous quality checks are conducted at every stage.

- Strategies for delivery of instruction

  Careful consideration is given for the integral development of abilities like thinking, problem solving, learning-to-learn etc. by selecting appropriate instructional strategies (training methodology), instructional activities and instructional materials.

  The area of IT is fast changing and nebulous. Hence considerable flexibility is provided in the instructional process by specially including creative activities with group interaction between the students and the trainer. The positive aspects of web based learning –acquiring information, organizing information and acting on the basis of insufficient information are some of the aspects, which are incorporated, in the instructional process.

- Assessment of learning

  The learning is assessed through different modes – tests, assignments & projects. The assessment system is designed to evaluate the level of knowledge & skills as defined by the learning objectives.

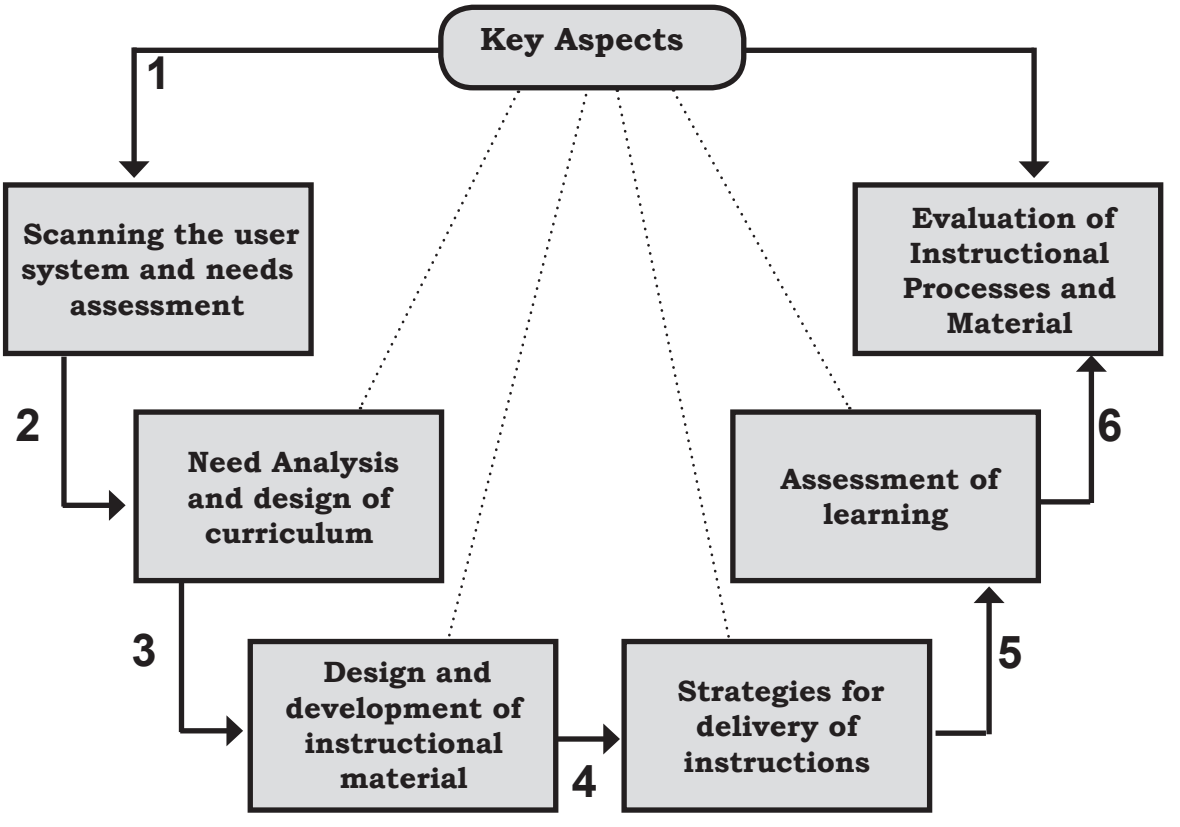
- Evaluation of instructional process and instructional materials

  The instructional process is backed by an elaborate monitoring system to evaluate - on-time delivery, understanding of a subject module, ability of the instructor to impart learning. As an integral part of this process, we request you to kindly send us your feedback in the reply pre-paid form appended at the end of each module.

*TAG – Technology & Academics Group comprises of members from Aptech Ltd., professors from reputed Academic Institutions, Senior Managers from Industry, Technical gurus from Software Majors & representatives from regulatory organizations/forums.

Technology heads of Aptech Ltd. meet on a monthly basis to share and evaluate the technology trends. The group interfaces with the representatives of the TAG thrice a year to review and validate the technology and academic directions and endeavors of Aptech Ltd.

Aptech New Products Design Model



“A little learning is a dangerous thing,
but a lot of ignorance is just as bad”

Preface

It is a known fact that computers make life easy for us. However, another fact that is equally important and significant is that computers by themselves are not intelligent. They have to be instructed or rather 'programmed' to perform the tasks that we want them to. Over the years, several programming languages have been developed to help the programmers to get the computer to perform the required tasks. While the programming languages have been varied in terms of the keywords they have, and the way in which they are written, the basic approach to writing a program has remained more or less the same.

While designing this module, we have identified the topics that are required to build the strong foundation that a programmer needs. Problems have been provided within the "Try It Yourself" sections.

This book is the result of a concentrated effort of the Design Team, which is continuously striving to bring you the best and the latest in Information Technology. The process of design has been a part of the ISO 9001 certification for Aptech-IT Division, Education Support Services. As part of Aptech's quality drive, this team does intensive research and curriculum enrichment to keep it in line with industry trends.

We will be glad to receive your suggestions. Please send us your feedback, addressed to the Design Centre at Aptech's corporate office, Mumbai.

Design Team

“Nothing is a waste of time if you use the experience wisely”

Table of Contents

Sessions

1.	Basics of C - Concepts	1
2.	Variables and Data Types - Concepts	29
3.	Variables and Data Types - Lab	47
4.	Operators and Expressions - Concepts	53
5.	Operators and Expressions - Lab	71
6.	Input and Output in ‘C’ - Concepts	83
7.	Condition - Concepts	107
8.	Condition - Lab	125
9.	Loop - Concepts	137
10.	Loop - Lab	159
11.	Arrays - Concepts	169
12.	Arrays - Lab	187
13.	Pointers - Concepts	201
14.	Pointers - Lab	225
15.	Functions - Concepts	235
16.	Functions - Lab	261
17.	Strings - Concepts	267
18.	Strings - Lab	283
19.	Advanced Data types and Sorting - Concepts	293
20.	Advanced Data types and Sorting - Lab	311
21.	File Handling - Concepts	321

Sessions

22. File Handling - Lab 343

Appendix A i

Glossary i

Session

1 Basics of C

Concepts

Objectives

At the end of this session, you will be able to:

- *Differentiate between Command, Program and Software*
- *Explain the beginning of C*
- *Explain when and why is C used*
- *Discuss the C program structure*
- *Discuss algorithms*
- *Draw flowcharts*
- *List the symbols used in flowcharts*

Introduction

Today computers have pervaded every field. Automation is the key concept that is driving the world. Any kind of job requires some amount of knowledge of IT and programming. C is a high level programming language, which every programmer should know. Hence in this book, we will be studying the C language constructs in detail. To start with let us understand the difference between the words software, program and command.

1.1 Instructions to a Computer

When a computer is started, it automatically does some processes and comes to a particular screen. How does this happen? The answer is simple. The operating system software exists inside the computer. The operating system is referred to as system software. This software starts up the computer and performs some initial settings before giving us an operational screen. To achieve this, the operating system is made up of a set of programs. Every program tries to give solution to one or more problems. Each program is set of instruction to solve the problem it tries to address. Thus a group of instructions make up a program and a group of programs make up software.

Let's consider an analogy to get more clarity: One of our friends comes home and we serve the Strawberry Milk Shake. He finds it so tasty that he too wants the recipe. So, we give him the recipe as:

1. Take some milk

2. Put Strawberry Crush

3. Blend and serve chill

Now if our friend follows these instructions as they are, he too will be able to make a wonderful Milk Shake.

Let us analyze these instructions –

- Take the first instruction: Is it complete? Does it answer the question as to 'Where' the milk has to be taken?

- Take the second instruction; again here it is not very clear as to where the Strawberry Crush should be put

Luckily our friend was intelligent enough to understand in spite of these doubts in the recipe. But if we have to publish this recipe, then should we not be a bit more careful? Well, certainly yes. Let's modify the steps like this:

1. Pour 1 glass of Milk into the Mixer Jar

2. Mix some Strawberry Crush to the Milk in the Mixer

3. Close the Lid of the Jar

4. Switch on the Mixer and start blending

5. Stop the mixer

6. If the Crush is fully mixed with milk then switch off the mixer else start on the Mixer again

7. Once done, pour the contents of the Mixer Jar into another bowl and put it into the refrigerator

8. Leave it for sometime, then take it out and serve

Compare both the recipes for the Milk Shakes and judge yourself which set of instructions is more complete. Surely, the second set will be the one, which anyone could read and understand.

Similarly, the computer also processes the data based on the set of instructions given to it. Needless to say, the set of Instructions given to the computer also should be meaningful and complete. Apart from this, they should be:

1. Sequential

2. Finite

3. Accurate

Each of the instruction in the instruction set is known as a Command and the set itself is known as a Program.

Now let's consider the case of making the computer to add 2 numbers for us.

The program for it could be

1. Take the first number and remember it

2. Take the other number and remember it

3. Perform the '+' operation between the first and the second number; also remember it as the result.

4. Then display the result

5. Stop

The instruction set given here adheres to all the rules mentioned above. So, this instruction set is a program and will successfully make the computer accomplish the task of adding two numbers.

Note: Just like our remembering capability is known as memory, the capability of the computer to remember data given to it is also known as memory. It is only because of this that the computer can take the data at an instance and work with it at another instance, i.e. it first registers the data in its memory and then reads its memory to retrieve values and work with them.

When the job to be given to the computer is big and complicated then all the commands cannot be put into one program, they need to be broken into a number of programs. All the programs are finally integrated to work together. Such an integrated set of programs is known as software.

The relationship between the three concepts of commands, programs and software could be diagrammatically represented as shown in Figure 1.1.

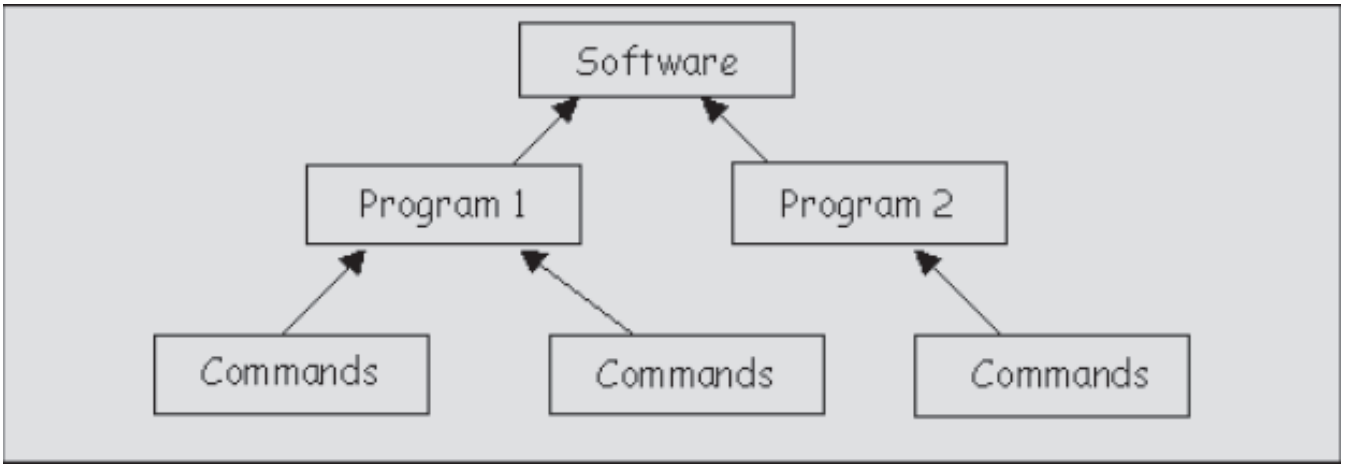


Figure 1.1: Software, Program and Commands

1.2 The C Language

Dennis Ritchie from Bell Laboratories created C, in the early 1970s. C was initially used on a system with UNIX operating system. C was derived from a development process, of an older language BCPL, developed by Martin Richards. BCPL was evolved into a language B, written by Ken Thompson, which was the originator of C.

While, BCPL and B do not support data-types (they are typeless), C provides a variety of data types. The main data- types are characters, integers and floating-point numbers.

C is closely associated with the UNIX system yet C is not tied to any one operating system or machine. C has been effectively used to write programs in different domains.

C was used for systems programming. A system program is associated with the operating system of the computer or its support utilities. Operating Systems, Interpreters, Editors, Assembly programs are usually called systems programs. UNIX Operating System was developed using C. C is now being used by many programmers for kinds of tasks because of its portability and efficiency. C compilers are available for almost all computers. Codes written in C on a one machine can be compiled and run on another machine by making a few or no changes. C compiler produces fast and error-free object code.

C also offers the speed of an assembly language. Programmers can create and maintain library of functions, which can reused by other programs. Thus large projects can be managed easily, with minimum efforts.

1.2.1 C - A Middle Level Language

C is thought of as a middle-level language because it combines elements of high-level languages and functionalities of an assembly (low-level) language. C allows manipulation of the basic elements of a

computer i.e. bits, bytes, addresses etc. Also, C code is very portable, that is, software written on one type of computer can work on another type of computer. Although C has five basic built-in data types, it is not strongly typed language as compared to high-level languages. C allows data type conversions. It allows direct manipulation of bits, bytes, words, and pointers. Thus, it is used for system-level programming.

1.2.2 C - A Structured Language

The term block-structured language does not apply to C. A block-structured language permits procedures and functions to be declared inside another procedure or function. C does not allow creation of functions within functions so it's not a block-structured language. However, it is referred to as a structured language because it is similar in many ways to other structured languages like ALGOL, Pascal and the likes.

C allows synthesis of code and data. This is a distinguishing feature of any structured language. It refers to the ability of a language to collect and hide all information and instructions, necessary to perform a specific task, from the rest of the program. This can be done using functions or code blocks. Functions are used to define and separate, tasks required in a program. This allows programs act as a unit. Code block is a logically connected group of program statements that is treated like a unit. A code block is created by placing a sequence of statements between opening and closing curly braces as shown below.

```
do
{
  i = i + 1;
  .
  .
  .
} while (i < 40);
```

Structured language support several loop constructs, such as while, do-while, and for. These loop constructs help the programmers to control the flow of the program.

1.3 The C Program Structure

C has few keywords, 32 to be precise. These keywords, combined with the formal C syntax, form the C language. But many C compilers have added more keywords to use the memory organization of certain preprocessors.

Some rules for programs written in C are as follows:

- All keywords are lower cased
- C is case sensitive, `do while` is different from **DO WHILE**
- Keywords cannot be used for any other purpose, that is, they cannot be used as a variable or

Concepts

function name

- `main()` is always the first function called when a program execution begins (discussed later in the session)

Consider the following program code:

```
main ()
{
  /* This is a sample program */
  int i = 0;
  i = i + 1;
  .
  .
}
```

Note: Various aspects of a C program are discussed with respect to the above code. This code will be referred to as `sample _ code`, wherever applicable.

1.3.1 Function Definition

C programs are divided into units called functions. The `sample _ code` has only one function `main()`. The operating system always passes control to `main()` when a C program is executed.

The function name is always followed by parentheses. The parentheses may or may not contain parameters.

1.3.2 Delimiters

The function definition is followed by an open curly brace ({). This curly brace signals the beginning of the function. Similarly a closing curly brace (}) after the statements, in the function, indicate the end of the function. The opening brace ({) indicates that a code of block is about to begin and the closing brace (}) terminates the block of code. In `sample_code`, there are two statements between the braces. In addition to functions, the braces are also used to delimit blocks of code in other situations like loops and decision-making statements.

1.3.3 Statement Terminator

Consider the line `int i = 0` in `sample_code` is a statement. A statement in C is terminated with a semicolon (;). A carriage return, **whitespace**, or a **tab** is not understood by the C compiler.

There can be more than one statement on the same line as long as each one of them is terminated with a semi-colon. A statement that does not end in a semicolon is treated as an invalid line of code in C.

1.3.4 Comment Lines

Comments are usually written to describe the task of a particular command, function or an entire program. The compiler ignores them. In C, comments begin with `/*` and are terminated with `*/`, in case the comments contain multiple lines. Care should be taken that the terminating delimiter (*/) is not forgotten. Otherwise, the entire program will be treated like a comment. In `sample_code`, "This is a sample program" is a comment line. In case the comment contains just a single line you can use `//` to indicate that it is a comment. For example:

```
int a=0; //Variable 'a' has been declared as an integer data type
```

1.3.5 The C Library

All C compilers come with a standard library of functions that perform the common tasks. In some installations of C, the library exists in one large file while in others it is contained in numerous small files. While writing a program, the functions contained in the library can be used for various tasks. A function written by a programmer can be placed in the library and be used in as many programs as and when required. Some compilers allow functions to be added in the standard library, while some compilers require a separate library to be created.

1.4 Compiling and Running a Program

The various stages of translation of a C program from source code to executable code are as follows:

- **Editor/Word Processor**

  The source code is written using an editor or a word processor. The code should be written in the form of standard text files, as C accepts source code only in this form. Some compilers supply programming environments (see appendix) that include an editor.

- **Source Code**

  This is the text of the program, which the user can read. It is the input for the C compiler.

- **C Preprocessor**

  The source code is first passed through the C preprocessor. Preprocessors, act on statements beginning with #. These statements are called directives (explained later). The directives are usually placed at the start of the program, though they can be placed anywhere else. The directives are short names given to a set of code.

- **Expanded C Source Code**

  The C preprocessor expands the directives and produces an output. This is called the expanded C source code. This expanded C source code is then passed on the C compiler.

- **C Compiler**

  The C compiler translates the expanded source code into the machine language, which is understood by the computer.

  If the program is too large it can be put in separate files and each of the files can be compiled separately. The advantage of this separation is that if code in a file is changed, the entire program need not be recompiled.

- **Linker**

  The object code along with support routines from the standard library and any other separately compiled functions are linked together by the linker into an executable code.

- **Loader**

  The executable code is run using the system's loader.

  The above process is shown in figure 1.2.

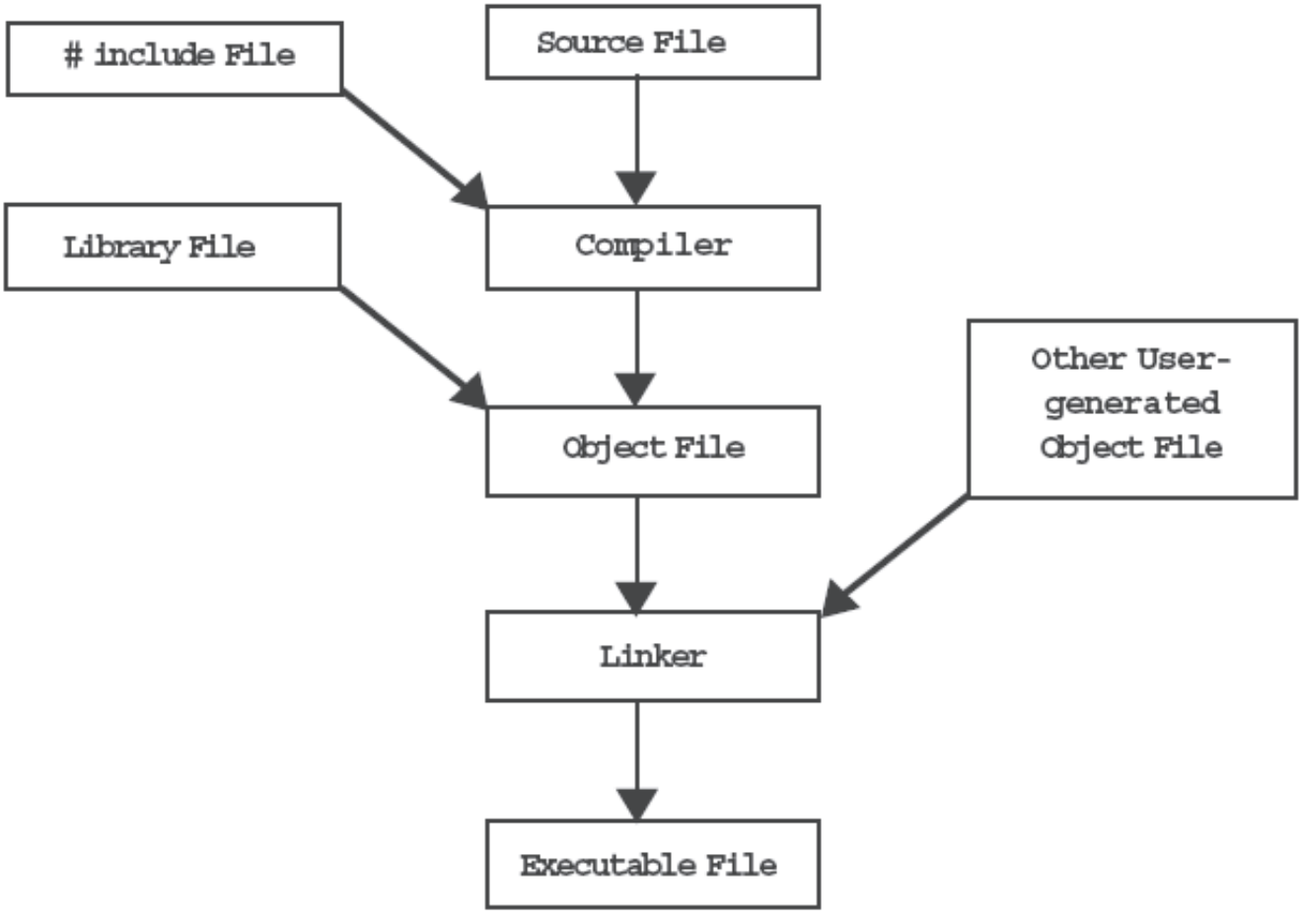


Figure 1.2: Compiling and Running a Program

1.5 The Programming Approach to Solving Problems

We often come across problems that need to be solved and to solve the problem we need to, first, understand the problem, and then working out a set of steps to overcome the problem.

Assume that you want to make a quick visit from your classroom to the cafeteria, which is located in the basement. Having understood the task, we would need to work out the steps involved, before actually performing this task. The steps would look as shown below:

STEP 1: Leave the room

STEP 2: Head towards the staircase

STEP 3: Go down to the basement

Concepts

STEP 4: Head for the cafeteria

The procedure given above lists well defined and clear set of executable steps that need to be performed in order to solve the problem at hand. Such a set of steps is called an algorithm (also called an 'algo' for short).

An algorithm can be defined as a procedure, formula, or recipe for solving a problem. It consists of a set of steps that help to arrive at a solution.

From our discussion it is obvious that in order to solve a problem, we need to first understand the problem. Next, we need to gather all relevant information required. Once this is done, the next step would be to process these bits of information. Finally, we would arrive at the solution to the problem.

The algorithm we have are a set of steps listed in simple language. It is very likely that though the steps written by two people may be similar, the language used to express these steps may be different. It is, therefore, necessary to have some standard method of writing algorithms so that everyone easily understands it. Hence, algorithms are written using two standard methods-pseudo codes, and flowcharts.

Both these methods are used to specify a set of steps that need to be performed in order to arrive at the solution. Referring to the problem of visiting the cafeteria, we have worked out a plan (an algorithm) to reach there. However, to reach the cafe, we still need to actually perform these steps. In the same manner, a pseudo code and a flowchart just represent the steps that need to be followed. The programmers code the actual execution of the steps when they write the code, using some language, to perform the specified steps.

A detailed look at pseudo codes and flowcharts is provided below.

1.5.1 Pseudo code

Note that 'pseudo code' is not actual code (pseudo=false). Pseudo code uses a certain standard set of words, which makes it resemble a code. However, unlike code, pseudo code cannot be compiled or run.

Let us, for example, consider the pseudo code written in Example 1 for displaying a 'Hello World!' message.

Example 1:

```
BEGIN
   DISPLAY 'Hello World!'
END
```

As can be seen in the simple pseudo code above, each pseudo code must start with the word BEGIN or START, and end with END or STOP. To display some value, the word DISPLAY or WRITE is used.

Since the value to be displayed is a constant value in this case, the value (Hello World) is enclosed within quotes. Similarly, to accept a value from the user, the word INPUT or READ is used.

To understand this better, let us have a look at the pseudo code (Refer to Example 2) for accepting two numbers from the user, and for displaying the sum of the two numbers.

Example 2:

```
BEGIN
    INPUT A, B
    DISPLAY A + B
END
```

In this pseudo code, the user inputs two values, which are stored in memory and can be accessed as A and B respectively. Such named locations in memory are called variables. A detailed explanation of variables will be dealt with later in the session. The next step in the pseudo code displays the sum of the values present in variables A and B.

However, the same pseudo code can be modified to store the sum of the variables in a third variable and then display the value in this variable as shown in Example 3.

Example 3:

```
BEGIN
    INPUT A, B
    C = A + B
    DISPLAY C
END
```

A set of instructions or steps in a pseudo code is collectively called a construct. There are three types of programming constructs - sequence, selection, and iteration constructs. In the pseudo codes, we have written above, we have used sequence constructs. These are called so as they are instructions that are performed in a sequence, one after the other, starting from the top. The other two types of constructs will be discussed in the sessions that follow.

1.5.2 Flowcharts

A flowchart is a graphical representation of an algorithm. It charts the flow of instructions or activities in a process. Each such activity is shown using symbols.

To understand this better, let us have a look at a flowchart, given in figure 1.3, for displaying the traditional ‘Hello World!’ message.

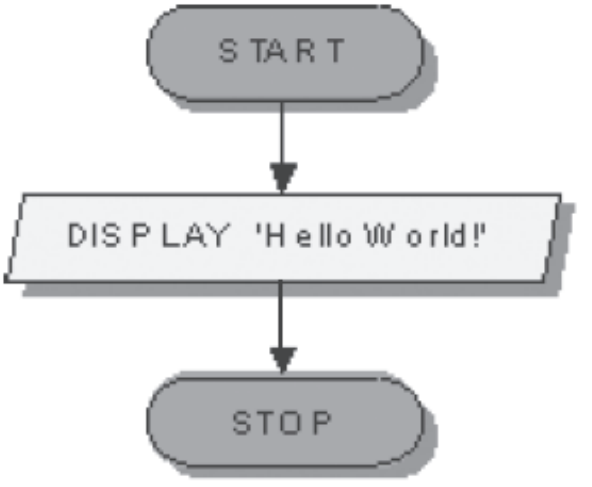


Figure 1.3: Flowchart to add two numbers

Flowchart's like pseudo codes begin with the START or BEGIN keyword, and end with the END or STOP keyword. In a similar way, the DISPLAY keyword is used to display some value to the user. However, here, every keyword is enclosed within symbols. The different symbols used in flowcharting and their relevance are tabulated in Figure 1.4.

Symbol	Description
	Start or End of the Program
	Computational Steps
	Input / Output instructions
	Decision making & Branching
	Connectors
	Flow Line

Fig 1.4: Flowchart Symbols

We shall now consider our earlier example where we accepted two numbers from the user, added them up, (stored the result in a third variable) and displayed the result. The flowchart for this would look as shown in Figure 1.5.

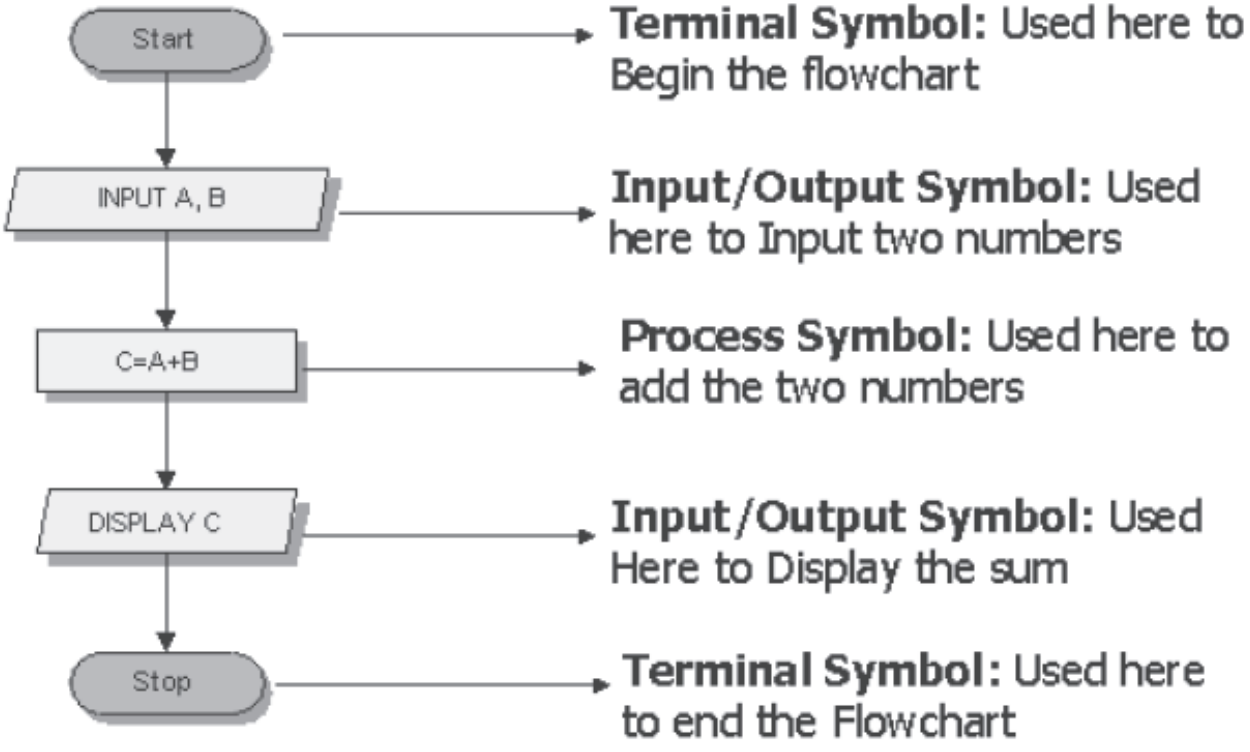


Fig 1.5: Flowchart to add to add two numbers

The step in which the values of the two variables are added and assigned to a third variable is considered to be a process, and is shown by a rectangle.

The flowcharts that we have discussed are small. Most often, flowcharts scan several pages. In such a case, symbols called connectors are used to indicate the location of the joins. They help us identify the flow across the pages. Circles represent connectors and need to contain a letter or a number, as shown in Figure 1.6. Using these we can pick up the link between two incomplete flowcharts.

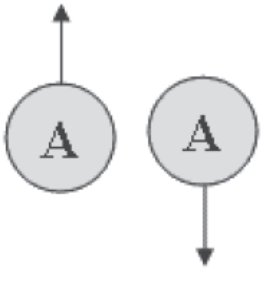


Figure 1.6: Connectors

Concepts

As flowcharts are used by programmers to write the necessary code, it is essential that it be represented in a way which any programmer can easily understands. Consider three programmers using three different languages to code; the problem being same. In this case the pseudo codes handed over to them may be the same. However, the language and therefore the syntax used to write the programs maybe different. But the end result is the same. Therefore, it is necessary that the problem is understood clearly, and the pseudo codes are written with care. We also conclude that pseudo codes are independent of the programming languages.

Some other essential things to be taken care of when drawing a flowchart are:

- Initially concentrate only on the logic of the problem and draw out the main path of the flowchart.
- A flowchart must have only one START and one STOP point.
- It is not necessary to represent each and every step of a program in the flowchart. Only the essential and meaningful steps need to be represented.

We have worked with sequence constructs in which, execution flows through all the instructions from the top in a sequence. We can come across conditions in our program, based on which the path of execution may branch. Such constructs are referred to as selection, conditional or branching constructs. These constructs will be discussed in more detail below.

- **The IF construct**

  A basic selection construct is an 'IF' construct. To understand this construct let us consider an example where the customer is given a discount if he makes a purchase of above $100. Every time a customer is billed, a part of the code has to check if the bill amount exceeds $100. If it does then deduct 10% of the total amount, otherwise, deduct nothing.

  This can be illustrated in rough pseudo code as follows:

  IF customer purchases items worth more than $100

  Give 10% discount

  The construct used here is an IF statement.

  The general form of an IF statement or construct is as follows:

```
IF condition
    Statements → Body of the IF construct
END IF
```

An 'IF' construct, begins with IF followed by the condition. If the condition evaluates to true then control is passed to the statements within its body. If the condition returns false, the statements within the body are not executed, and the program flow continues with the next statement after the END IF. The IF construct, must always end with an END IF, as it marks the end of the construct.

Let us take a look at example 4, which uses IF.

To find if a number is even we proceed as follows.

Example 4:

```
BEGIN
    INPUT num
    r = num MOD 2
    IF r=0
      Display "Number is even"
    END IF
END
```

The above code accepts a number from the user, performs the MOD operation on it, and checks if the remainder is zero. If it is, then it displays a message, otherwise it just exits.

A flowchart for the above pseudo code would look as shown in Figure 1.7.

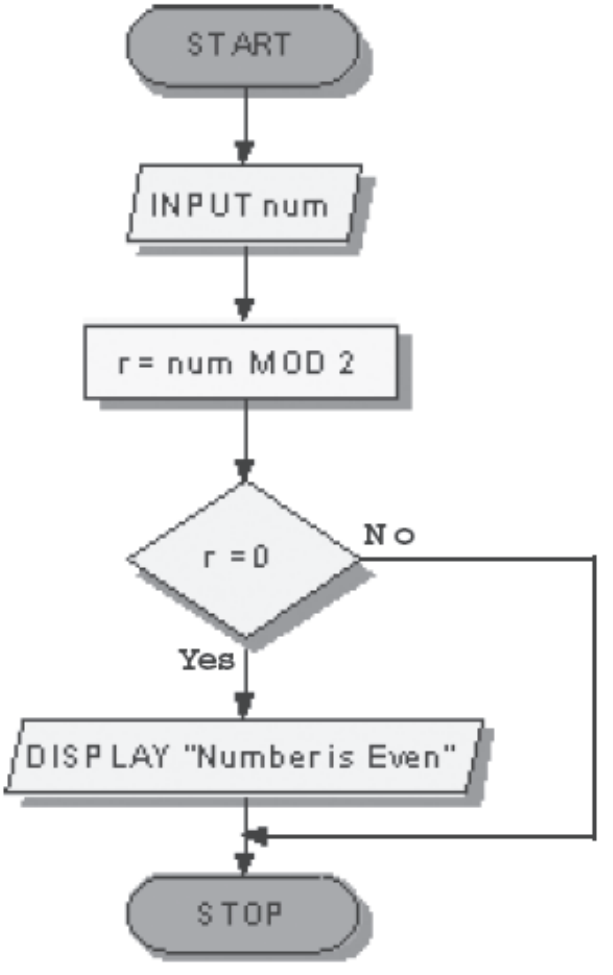


Figure 1.7 : Even Number or Not

The syntax for the IF statement in C is as follows:

```
if (condition)
{
  statements
}
```

➢ **The IF…ELSE Construct**

In Example 4, it will be nicer if we responded with a message saying that the number is not even (therefore odd) instead of just exiting. One way of achieving this can be by adding another IF statement which checks if the number is not divisible. Refer to Example 5.

Example 5:

```
BEGIN
    INPUT num
    r = num MOD 2
    IF r=0
      DISPLAY "Even number"
    END IF
    IF r<>0
      DISPLAY "Odd number"
    END IF
END
```

Programming languages provide us with what is known as an IF…ELSE construct. Using this would be a better and more efficient approach to solving the above problem. The IF…ELSE construct enables the programmer to make a single comparison and then execute the steps depending on whether the result of the comparison is True or False.

The general form of the IF…ELSE statement is as follows:

```
IF condition
  Statement set1
ELSE
  Statement set2
END IF
```

The syntax for the `if…else` construct in C is given below.

```
if(condition)
{
  statement set1
}
else
{
  statement set2
}
```

If the condition evaluates to True, `statement set1` is executed, otherwise `statement set2` is executed, but never both. So, a more efficient code for our even number example (example 5) would be as shown in Example 6.

Concepts

Example 6:

```
BEGIN
   INPUT num
   r=num MOD 2
   IF r=0
     DISPLAY "Even Number"
   ELSE
     DISPLAY "Odd Number"
   END IF
END
```

The flowchart for the above pseudo code is displayed in Figure 1.8.

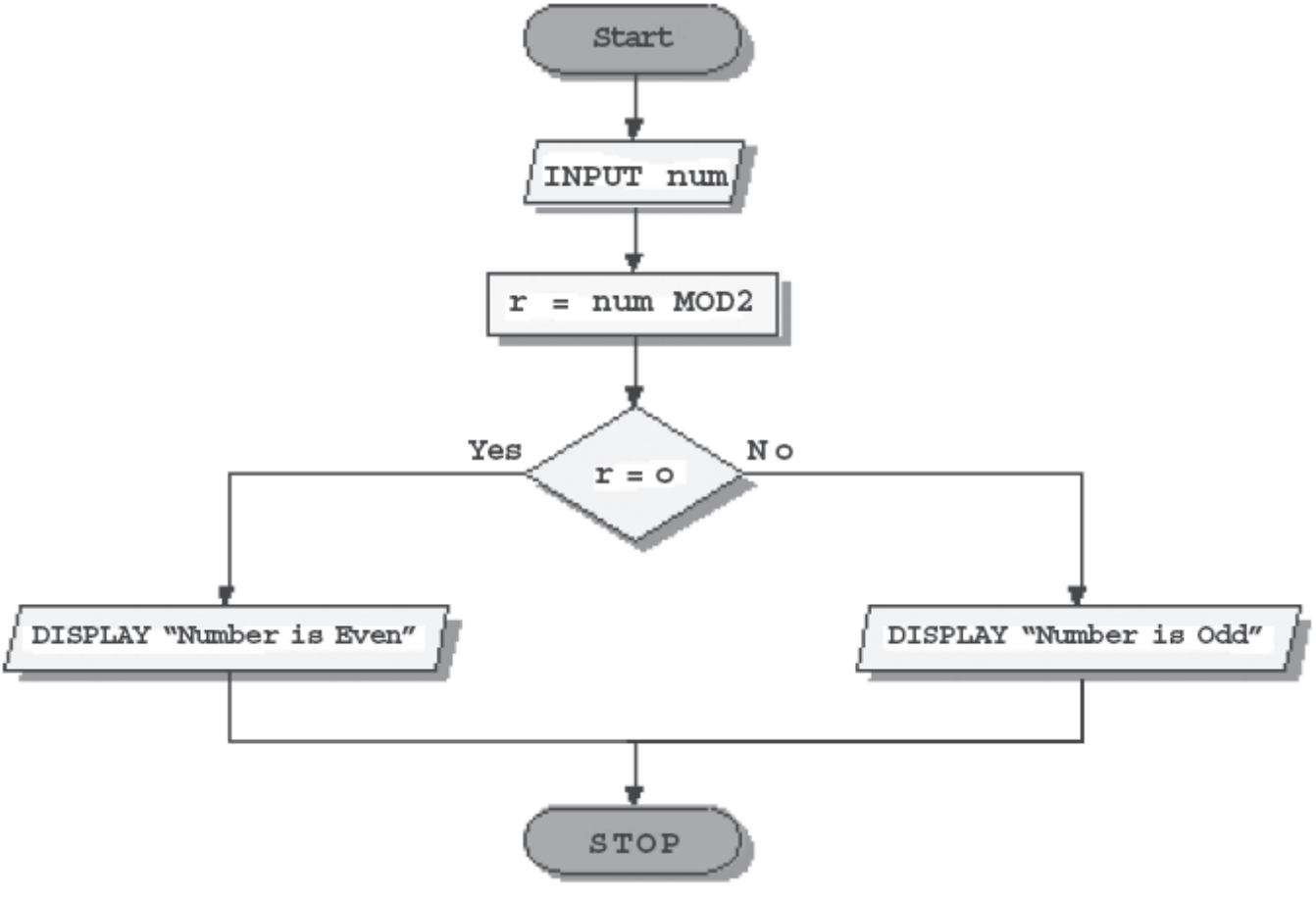


Figure 1.8: Even or Odd number

- **Multiple criteria using AND/OR**

The IF…ELSE construct helps to reduce the complexity enhances the efficiency. To some extent it also improves readability of the code. The IF examples we have discussed till now were fairly simple. They had one IF condition to evaluate. Sometimes we have to check for more than one condition for example say to classify a supplier as a MVS (Most Valuable Supplier), a company checks if he has been with them for at least 10 years, and done at least a total business of $5,000,000 Two conditions are required to be satisfied in order for the supplier to be considered an MVS. This is where the AND operator can be used in with the 'IF' statement. Refer to Example 7.

Example 7:

```
BEGIN
    INPUT yearsWithUs
    INPUT bizDone
    IF yearsWithUs >= 10 AND bizDone >= 5000000
      DISPLAY "Classified as an MVS"
    ELSE
      DISPLAY "A little more effort required!"
    END IF
END
```

Example 7 too was fairly simple, as it had two conditions. In real life situations, we might come to a point where plenty of conditions may need to be checked. Even there we can conveniently use the AND operator to connect them just like how we used it above.

Now supposing the company in the above example changes its rules and decides to give it's suppliers premium MVS status when either of the conditions hold good. That is, if a supplier has either been with them for 10 years or has done business of $5,00,000 or more for them. Here we replace the AND operator with the OR operator. Remember we had learnt in our previous section that the OR operator evaluates to True even if one of the conditions is True.

- **Nested IFs**

Another way to do Example 7 involves using Nested IFs. A nested IF is an IF inside another IF statement. Let us re-write the pseudo code in Example 7 using nested IFs. The code would look like in Example 8.

Example 8:

```
BEGIN
    INPUT yearsWithUs
    INPUT bizDone
```

Concepts

```
    IF yearsWithUs >= 10
      IF bizDone >= 5000000
        DISPLAY "Classified as an MVS"
      ELSE
        DISPLAY "A little more effort required!"
      END IF
    ELSE
      DISPLAY "A little more effort required!"
    END IF
END
```

The above code performs the same function without an 'AND'. However, we do have an IF (checking if bizDone is more than $5,000,000) within another IF (checking yearsWithUs is more than 10). The first IF checks to see if the supplier has been with the company for 10 years or more. If this returns False, it rejects the supplier as a MVS; else it enters the second IF, checking if the bizDone is greater than $5,000,000. If this is True it says that the supplier has been classified as an MVS, else it displays a message rejecting it as an MVS.

The flowchart for the pseudo code in example 8 is displayed in Figure 1.9.

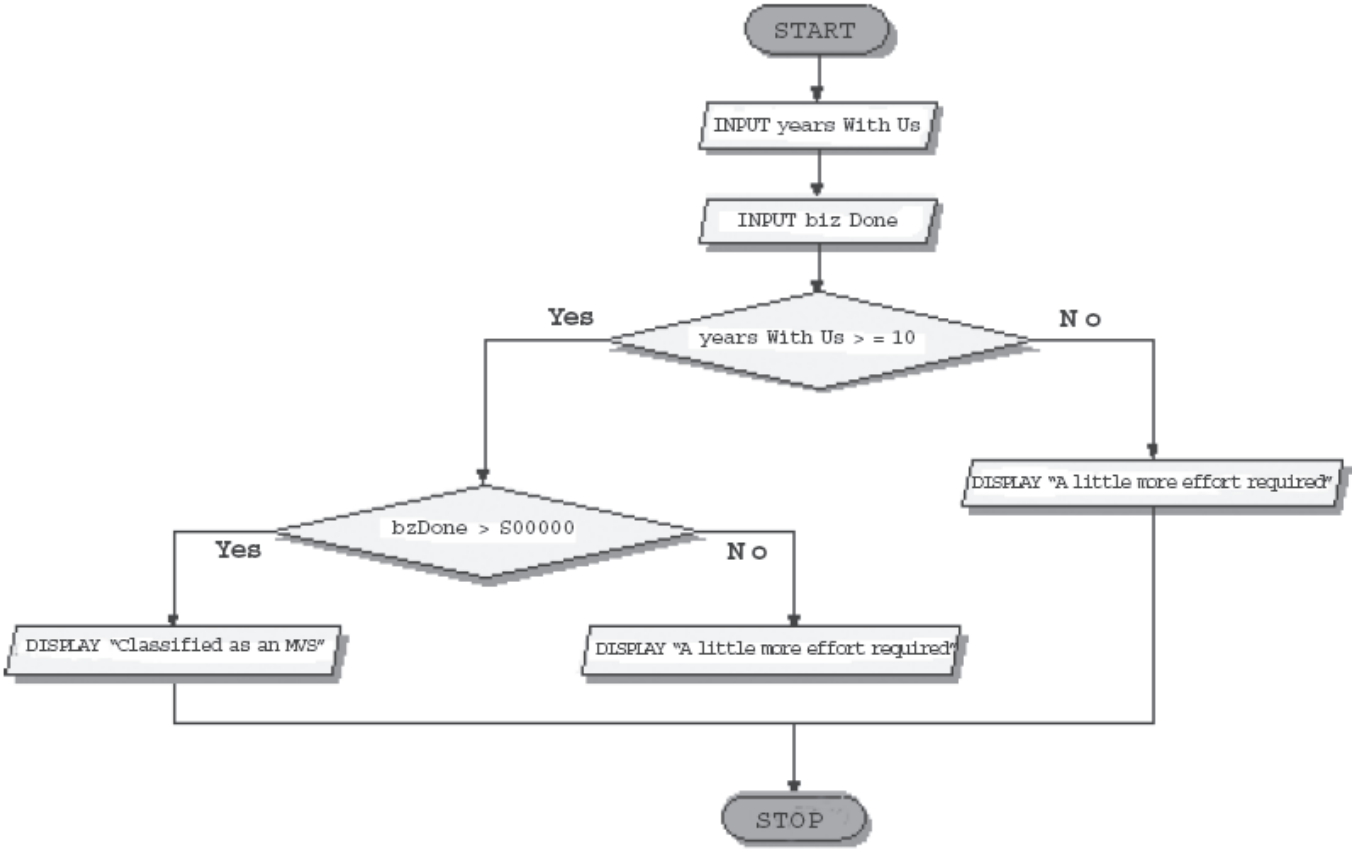


Figure 1.9: Nested IFs

The nested IF's pseudo-code in this case (Example 8) is inefficient. The MVS rejection statement has to be written twice. Besides the programmer has to write more code, the compiler has to evaluate two IF conditions thereby wasting time. While in the same example using the AND operator, just one IF was evaluated. This does not mean that nested IFs are inefficient. It depends on the situation where they are used. Sometimes using the AND operator is better, and in other cases nested IFs are more efficient. Let us consider an example where nested IFs will prove more efficient than using the AND operator.

A company allots basic salary to its employees based on the criteria specified in Table 1.1.

Grade	Experience	Salary
E	2	2000
E	3	3000
M	2	3000
M	3	4000

Table 1.1: Basic Salary

Therefore, for example, an employee with grade E and two years of experience would receive a salary of $2000, and an employee with the same grade but 3 years of experience would get a salary of $3000.

The pseudo code for the problem using the AND operator would be as in Example 9:

Example 9:

```
BEGIN
    INPUT grade
    INPUT exp
    IF grade = ”E” AND exp = 2
      salary=2000
    ELSE
      IF grade = “E” AND exp= 3
        salary=3000
      END IF
    END IF
    IF grade = “M” AND exp = 2
      salary=3000
    ELSE
      IF grade = “M” AND exp = 3
        salary=4000
```

```
      END IF
  END IF
END
```

The first IF checks the employee's grade and his experience. If his grade is E and experience is 2 years he is allotted a salary of $2000, else if his grade is E, but he has got 3 years of experience, his salary will be $3000.

If both the above evaluate to false, then the second IF similarly checks the employee's grade as well as experience, and allots the salary.

Supposing the grade of an employee is E and he has got 2 years of experience. His salary is calculated in the first IF. The ELSE part of the first IF is not executed; instead, however, it goes to the second IF checks the condition which of course would be false, so it checks the ELSE clause for the second IF which again would be false. It, therefore, unnecessarily goes through these steps. In our example we had just two IF statements. This is because we were checking for two grades. If a company has about 15 grade levels, the program would have to go through all IF and ELSE clauses, eating up precious time and system resources even after the salary has been calculated. This is definitely not an efficient way to write code.

Now let us consider the modified pseudo code using nested IFs in Example 10.

Example 10:

```
BEGIN
    INPUT grade
    INPUT exp
    IF grade="E"
      IF exp=2
        salary=2000
      ELSE
        IF exp=3
          salary=3000
        END IF
      END IF
    ELSE
     IF grade="M"
       IF exp=2
         Salary=3000
       ELSE
```

```
        IF exp=3
          Salary=4000
        END IF
       END IF
     END IF
    END IF
```

At first look, the above code may look a little awry. However, it is much more efficient. Let us take the same example. If an employee has a grade of E and has two years of experience, then his salary is calculated to be $2000 right in the first step of the first IF. After this, the code is exited as it does not need to execute any of the ELSE clauses. It therefore does not go through unnecessary IFs. So the code is more efficient and the program runs faster.

- **Loops**

A computer program, as we know, is a set of statements, which are normally executed one after the other. It may be required to repeat certain steps a specific number of times or till a certain condition is met. For example, suppose we wanted to write a program that would display your name 5 times. One way to do this would be by writing pseudo code similar to the one shown in Example 11.

Example 11:

```
BEGIN
   DISPLAY "Scooby"
   DISPLAY "Scooby"
   DISPLAY "Scooby"
   DISPLAY "Scooby"
   DISPLAY "Scooby"
END
```

To repeat the above for 1000 times is not logical. A different programming approach is required. This is where loops come into the picture. It would be easier if we could write the DISPLAY statement once and ask the computer to execute it 1000 times. In order to do this, we need to include the DISPLAY statement in a loop construct and instruct the computer to loop through the statement a 1000 times. A rough pseudo code of what a looping construct would look like is given in Example 12.

Example 12:

```
Do loop 1000 times
  DISPLAY "Scooby"
end loop
```

The block of statement(s) [in this case just the DISPLAY statement] that appear in between the do loop and end loop statements get executed 1000 times. These statements along with the do loop and end loop statements are called as looping or iterative construct. Loops enable programmers to develop to the point programs which otherwise would require thousands of statements to perform. Here we have used Do `loop…end` loop to indicate a general form of a loop.

Another way of writing the same code is shown in Example 13.

Example 13:

```
BEGIN
   cnt=0
   WHILE (cnt < 1000)
   DO
     DISPLAY "Scooby"
     cnt=cnt+1
   END DO
END
```

The flowchart for the pseudo code in Example 13 would look as shown in Figure 1.10.

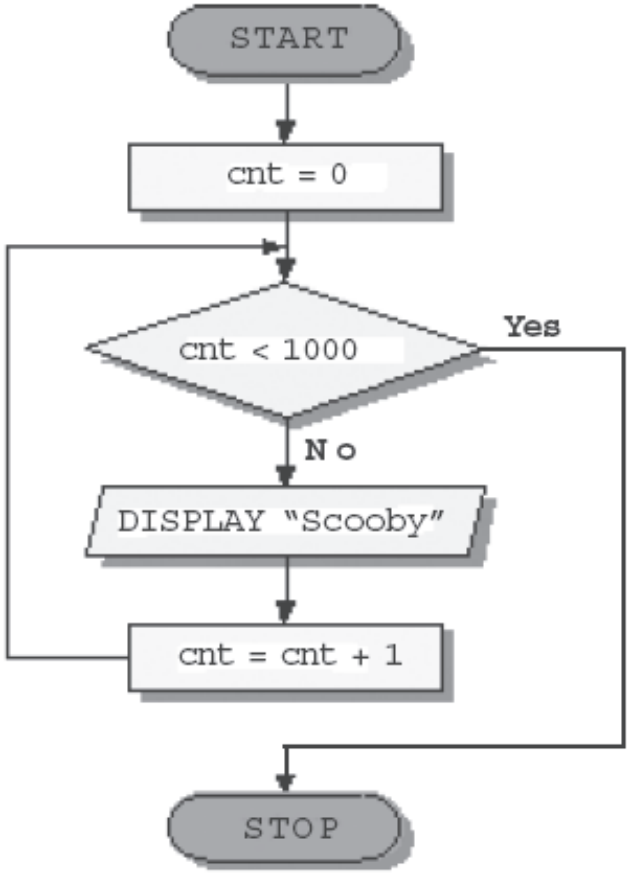


Figure 1.10: Loops

Notice that in Figure 1.10 there is no special symbol for showing loops. We use the branching symbol to check the condition and manage the flow of the program using flow-lines.

Concepts

Summary

- Software is a set of programs.
- A Program is a set of instructions.
- Code blocks, form a base of any C program.
- The C language has 32 keywords.
- Steps involved in solving a problem are studying the problem in detail, gathering the relevant information, processing the information and arriving at the results.
- An Algorithm is a logical and concise list of steps to solve a problem. Algorithms are written using pseudo codes or flowcharts.
- A pseudo code is a representation of an algorithm in language that resembles code.
- A flowchart is a diagrammatic representation of an algorithm.
- Flowcharts can be broken into parts and connectors can be used to indicate the location of the joins.
- When we come across a condition based on which the path of execution may branch. Such constructs are referred to as selection, conditional or branching constructs.
- The basic selection construct is an 'IF' construct.
- The IF …ELSE construct enables the programmer to make a single comparison and then execute the steps depending on whether the result of the comparison is True or False.
- A nested IF is an IF inside another IF statement.
- Often it is necessary to repeat certain steps a specific number of times or till some specified condition is met. The constructs, which achieve these, are known as iterative or looping constructs.

Check Your Progress

1. C allows _____ of code and data.

2. A _____ is a diagrammatic representation that illustrates the sequence of operations to be performed to arrive at a solution.

3. Flowcharts help us review and debug programs easily. (True / False)

4. A flowchart can have any number of start and stop points. (True / False)

5. A ____ is basically the execution of a sequence of statements until a particular condition is True or False.

Try It Yourself

1. Write a pseudo code and draw a flowchart to accept a value in degrees Celsius and to convert it into Fahrenheit. [Hint: C/5 = (F-32)/9]

2. Write a pseudo code and flowchart to accept a student's marks in Physics, Chemistry, and Biology. The total of these marks as well as the average should be displayed.

Session

2 Variables and Data Types

Concepts

Objectives

At the end of this session, you will be able to:

- *Discuss variables*
- *Differentiate between variables and constants*
- *List the different data types and make use of them in C programs*
- *Discuss arithmetic operators*

Introduction

Any application needs to handle data; it needs some place where this data can be temporarily stored. This 'place' where the data is stored is called the memory. The different locations in memory can be identified using unique addresses. Early programming languages required the programmer to keep a track of each memory location in terms of its address, as well as the value in it. This address was used by the programmer to access or alter the contents of the memory. As programming languages evolved, this access/alter of memory was simplified through the introduction of the concept of variables.

2.1 Variables

An application may handle more than one piece of data. In such a case, the application has to assign memory for each piece of data. While assigning memory, consider the following two factors:

1. How much memory is to be assigned?

2. Remembering at what memory address, data is stored.

Earlier, programmers had to write their programs in the language of the machine that is, in 1's and 0's. If the programmer wanted to temporarily store a value, the exact location where the data is stored inside the memory of the computer had to be assigned. This storage location had to be a specific number or memory address.

Modern day languages enable us to use symbolic names known as variables, to refer to the memory location where a particular value is to be stored.

The data type decides the amount of memory to be assigned. The names that we assign to variables help us in reusing the data as and when required.

Concepts

We are familiar with the use of letters, which represent quantities in a formula. For example, the area of a rectangle is given by:

```
Area = A = Length x Breadth = L x B
```

The simple interest is given by:

```
Interest = I = Principal x Time x Rate / 100 = P x T x R / 100
```

The letters **A**, **L**, **B**, **I**, **P**, **T**, **R**, are all variables and are short notations that represent various values.

Consider the following example:

The sum of the marks obtained by 5 students is to be displayed. The sum can be displayed with the help of the following instruction.

```
Display the sum of 24, 56, 72, 36 and 82.
```

Once the sum is displayed, it is lost from the computer's memory. Suppose we want to calculate the average of the marks, the sum would have to be recalculated.

A better approach would be to store the result in the computer's memory, and reuse it as and when required.

```
sum = 24 + 56+ 72 + 36 + 82
```

Here, **sum** is a variable that is used to store the sum of the five numbers. When we have to calculate average, it can be done as follows:

```
Avg = sum / 5
```

In C, all variables should be declared before they can be used. Let us, for example, take the example of accepting two numbers and displaying their sum. The code for the same is provided in Example 1.

Example 1:

```
BEGIN
    DISPLAY 'Enter 2 numbers'
    INPUT A, B
    C = A + B
    DISPLAY C
END
```

`A`, `B` and `C` in the code above, are variables. The variable names save you from remembering memory locations by address. When code is written and executed, the operating system takes care of allocating some memory space for these variables. The operating system, map a variable name to a specific memory location. Now, to refer to a particular value in memory, we need to specify just the variable name. In the example above, two values are accepted from the user, and these are stored in some memory location. These memory locations can be accessed via the variable names `A` and `B`. In the next step, the variables are added and the result is stored in a third variable called `C`. Finally, the value in this variable is displayed.

While some languages allow the operating system to delete the contents of memory locations, and allocate this memory for reuse, other languages such as C require the programmer to clear unused memory locations through code. In both these cases, it is the operating system that takes care of allocating and deallocating memory.

The operating system acts as an interface between the memory locations and the programmer. The programmer does not need to keep a track of the various locations, but can let the operating system do that for him. This, however, takes away the control over the memory (the portion of memory in which relevant data is stored) from the programmer, and passes it to the operating system.

2.2 Constants

In case of variables, the values stored in them need not be the same throughout its lifetime. Life of a variable begins when it is declared and lasts till its scope. The statements within this scope block can access the value of this variable. They even can change the value of the variables. There is need for certain items whose value can never be changed.

A **constant** is a value whose worth never changes. For example, `5` is a constant, whose mathematical value is always `5` and can never be changed by anybody. Similarly 'Black' is a constant that represents black color alone. While `5` is termed as **numeric constant**, 'Black' is termed as **string constant**.

2.3 Identifier

The names of variables, functions, labels, and various other user-defined objects are called identifiers. These identifies can contain one or more characters. It is compulsory that the first character of the identifier is a letter or an underscore(_). The subsequent characters can be alphabets, numbers or underscores.

Some correct identifier names are `arena`, `s_count`, `marks40`, and `class_one`. Examples of wrong identifier names are `1sttest`, `oh!god`, and `start... end`.

Identifiers can be of any convenient length, but the number of characters in a variable that are recognized by a compiler varies from compiler to compiler. For example, if a given compiler recognizes first 31 significant digits of an identifier name, then the following will appear same to it.

Concepts

```
This is a testing variable .... testing
This is a testing variable .... testing ... testing
```

Identifiers in C are case sensitive, that is, **arena** is different from **ARENA**.

2.3.1 Guidelines for Specifying Identifier Names

The rules for naming variables are different for each programming language. However, some conventions that are typically followed are:

- Variable names must begin with an alphabet.
- The first character may be followed by a sequence of letters or digits and can also include a special character like the 'underscore'.
- Avoid using the letter **O** in places where it can be confused with the number **0** and similarly the lowercase letter **L** can be mistaken for the number **1**.
- Proper names should be avoided while naming variables.
- Typically, uppercase and lowercase letters are treated as different i.e. variables **ADD**, **add** and **Add** are all different.
- As case-sensitivity considerations vary with programming languages it is advisable to maintain a standard way of naming variables.
- A variable name should be meaningful and descriptive; it should describe the kind of data it holds. For example, if the sum of two numbers is to be found, the variable storing the result may be called **sum**. Naming it **s** or **ab12** is not a good idea.

2.3.2 Keywords

All languages reserve certain words for their internal use. These words hold a special meaning within the context of the particular language, and are referred to as 'keywords'. While naming variables we need to ensure that we do not use one of these keywords as a variable name.

For example, all the data types are reserved as keywords.

Hence, naming a variable **int** will generate an error, but naming a variable **integer** will not.

Some programming languages require the programmer to specify the name of the variable as well as the type of data that is to be stored in it, before actually using a variable. This step is referred to as

'variable declaration'. This step will be clearer when we discuss data types in the next section, for now it is important to note that this step helps the operating system to actually allocate the required amount of space to the variable at the beginning itself.

2.4 Data types

Different types of data that can be stored in variables are :

- **Numbers**
  - Whole numbers.

    Example, 10 or 178993455
  - Real numbers.

    Example, 15.22 or 15463452.25
  - Positive numbers
  - Negative numbers
- **Names**

  Example, John
- **Logical values**

  Example, Y or N

As the data that stored in variables is of different types, it requires different amounts of memory.

The amount of memory to be assigned to a variable depends on the data type stored in it.

To assign memory for a particular piece of data, we have to declare a variable of a particular data type.

The term, declaring a variable, means that some memory is assigned to it. This portion of memory is then referred to by the variable name. The amount of memory assigned by the operating system depends on the type of the data that is going to be stored in the variable. A data type, therefore, describes the kind of data that will fit into a variable.

The general form of declaring a variable is:

Concepts

```
Data type (variable name)
```

Data types, which are commonly used in programming tools, can be classified as:

1. Numeric data type - stores numeric value

2. Alphanumeric data type - stores descriptive information

These data types can have different names in different programming languages. For example, a numeric data type is called `int` in C language while Visual Basic refers to it as `integer`. Similarly, an alphanumeric data type is named `char` in C while in Visual Basic it is named `string`. In any case, the data stored is the same. The only difference is that the variables used in a tool have to be declared with the name of the data type supported by that tool.

C has five basic data types. All other data types are based upon one of these types. The five basic types are:

- `int` is an integer, typically representing the natural size of **integers**.
- `float` and `double` are used for floating point numbers. `float` occupies 4 bytes and can hold numbers with six digits of precision, whereas `double` occupy eight bytes and can hold numbers with ten digits of precision.
- `char` type occupies one byte long and is capable of holding one **character**.
- `void` is typically used to declare a function as returning **no value**. This will be clearer after the session on functions.

The size and range of these types vary with each processor type and with every implementation of the C compiler.

Note: Floating point numbers are used to represent values, which require a decimal point precision.

- **Type `int`**

  The data type, which stores numeric data, is one of the basic data types in any programming language. It consists of a sequence of one or more digits.

  For example in C, to store an integer value in a variable called ‘num’, the declaration would be as follows:

  ```
  int num;
  ```

  The variable ‘num’ cannot store any other type of data like “Alan” or “abc”. This numeric data type

allows integers in the range **-32768** to **32767** to be stored. The OS allocates 16 bits (2 bytes) to a variable that is declared to be of type `int`. Examples: **12322**, **0**, **-232**.

If we assign the value **12322** to **num**, then **num** is a float variable and **12322** is a float constant.

- **Type `float`**

A variable of type `float` is used for storing values containing decimal places. The compiler differentiates between the data types, `float` and `int`.

The main difference between the two is that `int` data type includes only whole numbers, whereas `float` data type can hold either whole or fractional numbers.

For example, in C, to store a `float` value in a variable called 'num', the declaration would be as follows:

```
float num;
```

Variable declared to be of type `float` can store decimal values with a precision of upto 6 digits. The variable is allocated 32 bits (4 bytes) of memory. Examples: **23.05**, **56.5**, **32**.

If we assign the value **23.5** to **num**, then **num** is a float variable and **23.5** is a float constant.

- **Type `double`**

The type `double` is used when the value to be stored exceeds the size limit of type `float`. Variables of the type `double` can roughly store twice the number of digits as `float` type.

The precise number of digits the `float` or `double` data type can store depends upon the particular computer system.

Numbers stored in the data type `float` or `double` are treated identically by the system in terms of calculation. However, using the type `float` saves memory as it takes only half the space as a `double` would.

`double` data type, allows a greater degree of precision (upto 10 digits). A variable declared of type `double` is allocated 64 bits (8 bytes).

For example in C, to store a `double` value in a variable called 'num', the declaration would be as follows:

```
double num;
```

If we assign the value **23.34232324** to **num**, then **num** is a double variable and **23.34232324** is a double constant.

➢ **Type char**

The data type char is used to store a single character of information.

A char type can store a single character enclosed within two single quotation marks. Some valid examples of char types are '**a**', '**m**', '**$**' '**%**'.

It is also possible to store digits as characters by enclosing them within single quotes. These should not be confused with the numeric values. For example, '**1**', '**5**' and '**9**' are not to be confused with the numbers **1**, **5** and **9**.

Consider the statements of C code provided below.

```
char gender;

gender='M';
```

The first line declares the variable 'gender' of data type char. The second line stores an initial value of '**M**' in it. The variable **gender** is a character variable and '**M**' is a character constant. The variable is allocated 8 bits (one byte) of memory.

➢ **Type void**

C has a special data type called void. This data type indicates the C compiler that there is no data of any type. In C, functions return data of some type. However, when a function has nothing to return, the void data type can be used to indicate this.

2.4.1 Basic and Derived Data types

The four (char, int, float and double) data types that we have discussed above are used for actual data representation in the computer's memory. These data types can be modified to fit various situations more precisely. The resulting data types are said to have been derived from these basic types.

A modifier is used to alter the base type to fit various situations. Except for void, all other data types can have modifiers preceding them. Modifiers used with C are signed, unsigned, long and short. All of these can be applied to character and integer data type. The long modifier can also be applied to double.

Some of the modifiers are:

1. `unsigned`

2. `long`

3. `short`

To declare a variable of a derived type, we need to precede the usual variable declaration with one of the modifiers. A detailed explanation of these modifiers and how they can be used is provided below.

➢ The `signed` and `unsigned` Types

Default integer declaration assumes a signed number. The most important use of `signed` is to modify the `char` data type, where `char` is `unsigned` by default.

The `unsigned` type specifies that a variable can take only positive values. This modifier may be used with the `int` and `float` data types. Unsigned can be applied to floating data type in some cases, but this reduces the portability of the code.

By prefixing the `int` type with the word `unsigned`, the range of positive numbers can be doubled.

Consider the statements of C code provided below, which declares a variable of type `unsigned int`, and initializes this variable to **23123**.

```
unsigned int varNum;

varNum = 23123;
```

Note that the space allocated to this type of variable remains the same. That is, the variable `varNum` is allocated 2 bytes since it is an `int`. However, the values that an unsigned `int` supports range from **0** to **65535**, instead of **–32768** to **32767** that an `int` supports. By default, an `int` is a `signed int`.

➢ The `long` and `short` Types

They are used when an integer of length shorter or longer than its normal length is required. A `short` modifier is applied to the data type when a length shorter than the normal integer length is sufficient and a `long` modifier is used when a length of more than the normal integer length is required.

The `short` modifier is used with the `int` data type. It modifies the `int` data type so that it occupies

less memory place. Therefore, while a variable of type `int` occupies 16 bits (2 bytes), a variable of type `short int` (or simply `short`), occupies 8 bits (1 byte), and allows numbers in the range **`-128`** to **`127`**.

The `long` modifier is used to accommodate a wider range of numbers. It can be used with the `int` as well as the `double` data types. When used with the `int` data type, the variable supports numbers between **`-2,147,483,647`** and **`2,147,483,647`**, and occupies 32 bits (4 bytes). Similarly, a 'long double' type of variable occupies 128 bits (16 bytes).

A `long int` variable is declared as follows:

```
long int varNum;
```

It can also be declared simply as `long varNum` as well. A long integer can be declared as `long int` or just `long`. Similarly, a short integer can be declared as `short int` or `short`.

Given below is a table, which shows the range of values for the different data types and the number of bits taken. This is based on the ANSI standard.

Type	Approximate Size in Bits	Minimal Range
char	8	-128 to 127
unsigned	8	0 to 255
signed char	8	-128 to 127
int	16	-32,768 to 32,767
unsigned int	16	0 to 65,535
signed int	16	Same as int
short int	16	Same as int
unsigned short int	8	0 to 65, 535
signed short int	8	Same as short int
signed short int	8	Same as short int
long int	32	-2,147,483,647 to 2,147,483,647
signed long int	32	0 to 4,294,967,295
unsigned long int	32	0 to 4,294,967,295
float	32	Six digits of precision
double	64	Ten digits of precision
long double	128	Ten digits of precision

Table 2.1: Data Types and their range

The following example shows declarations of the above-discussed types.

Example 2:

```
main ()
{
    char abc; /*abc of type character */
    int xyz; /*xyz of type integer */
    float length; /*length of type float */
    double area; /* area of type double */
    long liteyrs; /*liteyrs of type long int */
    short arm; /*arm of type short integer*/
}
```

Let us take the same example which we dealt with in the last session to add two numbers and display their sum. The pseudo code for the same is given in example 3.

Example 3:

```
BEGIN
    INPUT A, B
    C = A + B
    DISPLAY C
END
```

In this example, the values for two variables, **A** and **B**, are accepted. The values are added and the sum is stored in **C** using the statement **C = A + B**. In this statement, **A** and **B** are variables and the symbol **+** is called an operator. Let us discuss the various Arithmetic operators available in C in the following section. However, there are other types of operators available in C which will be covered in the next session.

2.5 Arithmetic Operators

Arithmetic operators are used to perform numerical operations. They are divided into two classes: **unary** and **binary** arithmetic operators.

Table 2.2 lists the arithmetic operators and their action.

Unary Operators	Action	Binary Operators	Action
-	Unary minus	+	Addition
+ +	Increment	-	Subtraction
--	Decrement	*	Multiplication
		%	Modulus

Unary Operators	Action	Binary Operators	Action
		/	Division
		^	Exponentiation

Table 2.2: Arithmetic operators and their action

➢ **Binary Operators**

In C, the binary operators work as they do in other languages. Operators like `+`, `-`, `*` and `/` can be applied to almost any built-in data type allowed by C. When `/` is applied to an integer or character, any remainder will be truncated. For example, 5/2 will equal to 2 in integer division. The modulus operator (`%`) gives the remainder of an integer division. For example 5%2 will give 1. However, `%` cannot be used with floating point types.

Let us consider an example for the exponentiation operator.

```
9 ^ 2
```

Here, 9 is the base and 2 is the exponent.

The number to the left of '^' is known as the base and the number to the right of '^' is known as the exponent.

The result of `9^2` evaluates to is `9 * 9 = 81`.

Another example is:

```
5 ^ 3
```

This effectively means:

```
5 * 5 * 5
```

Thus, `5 ^ 3 = 5 * 5 * 5 = 125`.

Note: The programming languages like Basic, supports the `^` operator to carry out exponentiation. However, ANSI C does not support the symbol, `^`, for exponentiation. The alternate way to calculate exponentiation in C is to use `pow()` function defined in `math.h`. The syntax to carry out the same is as follows:

```
#include<math.h>
void main(void)
{
    ...
    /* The following function will calculate x to the power y */
    z = pow(x,y);
    ...
}
```

The following example shows all the binary operators used in C. Note that the functions `printf()` and `getchar()` are not yet covered. We will be discussing the same in the forthcoming sessions.

Example 4:

```
#include<stdio.h>
main()
{
    int x,y;
    x = 5;
    y = 2;
    printf("The integers are : %d & %d\n", x, y);
    printf("The addition gives : %d\n", x+y);
    printf("The subtraction gives : %d\n", x-y);
    printf("The multiplication gives : %d\n", x*y);
    printf("The division gives : %d\n", x/y);
    printf("The modulus gives : %d\n", x%y);
    getchar();
}
```

The above code will produce the following output:

```
The integers are : 5 & 2
The addition gives : 7
The subtraction gives : 3
The multiplication gives : 10
The division gives : 2
The modulus gives : 1
```

Concepts

➢ Unary Arithmetic Operators

The unary arithmetic operators are unary minus (–), increment operator (++) and decrement operator (––).

- **Unary Minus (–)**

  The symbol is the same as used for binary subtraction. Unary minus is used to indicate or change the algebraic sign of a value. For Example,

  ```
  a = -75;
  ```

  ```
  b = -a;
  ```

  The above assignments result in `a` being assigned the value **`- 75`** and **`b`** being assigned the value **`75`** (**`-(-75)`**). The minus sign used in this way is called the unary operator because it takes just one operand.

  Strictly speaking, there is no unary + in C. Hence, an assignment statement like,

  ```
  invld_pls = +50;
  ```

  where, **`invld_pls`** is an integer variable, is not valid in standard C. However, many compilers do not object to such usage.

- **Increment and Decrement Operators**

  C includes two useful operators that are generally not found in other computer languages. These are the increment operator (++) and the decrement operator (––). The operator ++ adds 1 to its operand, whereas the operator –– subtracts 1 from its operand.

  To be precise,

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

  can be written as

  ```
  x++;
  ```

  and

  ```
  x = x - 1;
  ```

can be written as

```
x--;
```

Both these operators may either precede or follow the operand, i.e.

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

can be represented as

```
x++ or ++x;
```

Similar operations hold true for -- operator.

The difference between pre-fixing and post-fixing the operator is useful when it is used in an expression. When the operator precedes the operand, C performs the increment or decrement operation before using the value of the operand. This is known as pre-fixing. If the operator follows the operand, the value of the operand is used before incrementing or decrementing it. This is known as post-fixing.

Consider the following:

```
a = 10;
b = 5;
c = a * b++;
```

In the above expression, the current value of **b** is used for the product and then the value of **b** is incremented. That is, **c** is assigned **50(a*b)** and then the value of **b** is incremented to **6**.

If, however, the above expression was

```
c = a * ++b;
```

the value stored in **c** would be **60**, because **b** would first be incremented by **1** and then the value stored in **c(10*6)**.

In a context where the incrementing or the decrementing effect alone is required, the operators can be used either as postfix or prefix.

Most C compilers produce very fast, efficient object code for increment and decrement operations. This code is better than that generated by using the equivalent assignment statement. So, increment and decrement operators should be used whenever possible.

Concepts

Summary

- Most often, an application needs to handle data; it needs some place where this data can be temporarily stored. This 'place' where this data is stored is called the memory.
- Modern day languages enable us to use symbolic names known as variables, to refer to the memory location where a particular value is to be stored.
- There is no limit to the number of memory locations that a program can use.
- A constant is a value whose worth never changes.
- The names of variables, functions, labels, and various other user-defined objects are called identifiers.
- All languages reserve certain words for their internal use. They are called keywords.
- The main data types of C are character, integer, float, double float and void.
- Modifiers are used to alter the basic data types so as to fit into various situations. Unsigned, short and long are the three modifiers available in C.
- C supports two types of Arithmetic operators: Unary and Binary.
- Increment(++) and decrement(– –) are unary operators acting only on numeric variables.
- Arithmetic binary operations are + - * / % which act only on numeric constants, variables or expressions.
- The modulus operator % acts only on integers and results in remainder after integer division.

Check Your Progress

1. C is case sensitive. **(True / False)**

2. The number 10 is a _____.

3. The first character of the identifier can be a number. **(True / False)**

4. Using the type _____ saves memory as it takes only half the space as a _____ would.

5. The _____ data type is used to indicate the C compiler that no value is being returned.

6. _____ and _____ are the two classes of arithmetic operators.

   A. Bitwise & and | B. Unary and Binary

   C. Logical AND D. None of the above

7. The unary arithmetic operators are ___and ___ .

   A. ++ and – – B. % and ^

   C. ^ and $ D. None of the above

Concepts

Try It Yourself

1. Match the following:

Column A	Column B
8	
10.34	Invalid Identifier Names
A B C	Integer Constants
abc	Character Constants
23	Double
12112134.86868686886	Floating Point Numbers
_A1	Valid Identifier Names
$abc	
'A'	

Hint: Multiple items in Column A can map to a single element in Column B.

2. What will be the value of the variables at the end in each of the following code statements:

a.
```
int a=4^4
```

b.
```
int a=23.34
```

c.
```
a = 10

b = a + a++
```

d.
```
a=-5

b=-a
```

Session 3

Variables and Data Types (Lab)

Objectives

At the end of this session, you will be able to:

- *Use variable, data type and arithmetic expression*

Part I – For the first 1 Hour and 30 Minutes :

3.1 Variables

As we already know, Variables are names given to locations in the memory of computer which may store different values at different instances of time. In this session, let us focus on how to create and use variables.

3.1.1 Creating a variable

Variable creation involves the data type and the logical name for that variable. For example,

```
int currentVal;
```

In the example above, the name of variable is **`currentVal`** which is of integer data-type.

3.2 Data Type

A data type defines what kind of value will be stored by a particular variable. For example,

```
int currentVal;
```

In the example above, **`int`** specifies that **`currentVal`** variable will store integer value.

3.3 Arithmetic Expression

A C arithmetic expression consists of a variable name on the left hand side of “=” and a variable names and constants on the right hand side of “=”. The variables and constants appearing on the right hand side of “=” are connected by arithmetic operators like +, -, *, and /. For example,

```
delta = alpha * beta / gamma + 3.2 * 2 / 5;
```

Now, Let us look at a complete program to calculate the simple interest.

1. **Invoke the editor in which you can type the C program.**
2. **Create a new file.**
3. **Type the following code:**

```
#include<stdio.h>
void main()
{
    int principal, period;
    float rate, si;
    principal = 1000;
    period = 3;
    rate = 8.5;
    si = principal * period * rate / 100;
    printf("%f", si);
}
```

To see the output, follow these steps:

4. **Save the file with the name myprogram1.C**
5. **Compile the file, myprogram1.C**
6. **Execute the program, myprogram1.C**
7. **Return to the editor.**

The sample output of the above program is shown in Figure 3.1.

Figure 3.1: Output of myprogram1.C

1. **Create a new file.**

2. **Type the following code :**

```
#include<stdio.h>
void main()
{
    int a, b, c, sum;
    printf("\n Enter any three numbers: ");
    scanf("%d %d %d", &a, &b, &c);
    sum = a + b + c;
    printf("\n Sum = %d", sum);
}
```

3. **Save the file with the name myprogramII.C**

4. **Compile the file, myprogramII.C**

5. **Execute the program, myprogramII.C**

6. **Return to the editor.**

The sample output of the above program will be as shown in Figure 3.2.

Figure 3.2: Output I of myprogramII.C

Part II – For the next 30 Minutes:

1. Write a C program that accepts a number and square the number.

   To do this,

   a. Accept the number.

   b. Multiply the number with itself and display the square.

Try It Yourself

1. Write a C program to find the area and perimeter of a circle.

2. Write a C program that accepts the salary and age from the user and displays the same on the screen as output.

“Information’s pretty thin stuff unless mixed with experience”

Session

4 Operators and Expressions

Concepts

Objectives

At the end of this session, you will be able to:

- *Explain Assignment Operators*
- *Understand Arithmetic Expressions*
- *Explain Relational and Logical Operators*
- *Understand Bitwise Logical Operators and Expressions*
- *Explain Casts*
- *Understand the Precedence of Operators*

Introduction

C has a very rich set of operators. Operators are the tools that manipulate data. An operator is a symbol, which represents some particular operation to be performed on the data. C defines four classes of operators: **arithmetic**, **relational**, **logical**, and **bitwise**. In addition C has some special operators for special tasks.

Operators operate on constants or variables, which are called **operands**. Variables have already been discussed in the previous session. Constants are fixed values that the program cannot alter. C constants can be of any of the basic data types. Operators can be classified as either **unary**, **binary** or **ternary** operators. Unary operators act on only one data element, binary operators act on two data elements and ternary operators operate on three data elements.

```
c = a + b;
```

Here **a**, **b**, **c** are the operands and '=' and '+' are the operators.

4.1 Expressions

An expression can be any combination of operators and operands. Operators perform operations like addition, subtraction, comparison etc. Operands are the variables or values on which the operations are performed. For example, in **a + b, a** and **b** are operands and + is the operator. The whole thing together is an expression.

Concepts

During the execution of the program, the actual value of the variable (if any) is used along with the constants that are present in the expression. Evaluation takes place with the help of operators. Thus every C expression has a value.

The following are examples of expressions:

```
2
x
3 + 7
2 × y + 5
2 + 6 × (4 - 2)
z + 3 × (8 - z)
```

Roland weighs 70 kilograms, and Mark weighs **k** kilograms. Write an expression for their combined (or total) weight. The combined weight in kilograms of these two people is the sum of their weights, which is **70 + k.**

Evaluate the expression `4 × z + 12` when `z = 15`.

We replace each occurrence of **z** with the number 15, and simplify using the usual rules: parentheses (or brackets) first, then exponents, multiplication and division, then addition and subtraction.

```
4 × z + 12 becomes
4 × 15 + 12 = (multiplication comes before addition)
60 + 12 =
72
```

The Assignment Operator

Before looking into other operators, it is essential to discuss the assignment operator (=). It is the most common operator in any language, and well known to everyone. In C, the assignment operator can be used with any valid C expression. The general form of the assignment operator is:

```
variable_name = expression;
```

Multiple Assignment

Many variables can be assigned the same value in a single statement. This is done by using the multiple assignment syntax. For example:

```
a = b = c =10;
```

The above line of code assigns the value 10 to **a, b**, and **c**. However this cannot be done at the time of declaration of the variables. For example,

```
int a = int b = int c = 0;
```

This above statement gives an error because the syntax for multiple assignments is wrong here.

Arithmetic Expressions

The operations are carried out in a specific (or a particular) order, to arrive at the final value. This order is known as order of precedence (discussed later).

Mathematical expressions can be expressed in C using the arithmetic operators with numeric and character operands. Such expressions are known as **Arithmetic Expressions**. Examples of arithmetic expressions are:

```
a * (b+c/d)/22;
++i % 7;
5 + (c = 3+8);
```

As seen above, operands can be constants, variables or a combination of the two. Also, an expression can be a combination of several smaller sub-expressions. For instance, in the first expression, **c/d** is a sub-expression, and in the third expression **c = 3+8** is also a sub-expression.

4.2 Relational Operators and Expressions

Relational operators are used to test the relationship between two variables, or between a variable and a constant. For example, the test for the larger of the two numbers, **a** and **b**, is made by means of the greater than (>) sign between the two operands **a** and **b (a > b)**.

In C, **true** is any value other than zero, and false is zero. Expressions that use relational operators, return 0 for false and 1 for true. For example, consider the following expression:

```
a == 14;
```

This expression checks for equality, in the sense that it checks whether the value contained in the variable **a** is equal to 14 or not. The value of this expression is said to be `0` (`false`) if **a** has a value that is not equal to 14 and `1` (`true`) if it is 14.

The relational operators and the operation they perform are listed in Table 4.1 below.

Operator	Relational Operators Action
>	Greater than
> =	Greater than or equal
<	Less than
< =	Less than or equal
= =	Equal
!=	Not equal

Table 4.1: Relational operators and their action

4.3 Logical Operators and expressions

Logical operators are symbols that are used to combine or negate expressions containing relational operators.

Expressions that use logical operators return `0` for `false` and `1` for `true`.

The logical operators and the operation they perform are listed in Table 4.2 below.

Operator	Logical Operators Action
&&	A N D
||	O R
!	NOT

Table 4.2: Logical operators and their action

As for the explanation of the logical operators, consider the following. Any operator that uses two characters as a symbol should have no space between the two characters, that is == is not correct if it is written as = =.

Suppose a program has to perform certain steps if the conditions **`a < 10`** and **`b == 7`** are satisfied. This condition is coded using the relational operator in conjunction with the logical operator `AND`. This `AND` operator is represented by `&&`, and the condition will be written as:

```
(a < 10) && (b == 7);
```

Similarly the `OR` is operator used to check whether one of the conditions is true. It is represented by two consecutive pipe signs (`||`). If the above example is considered in such a way that the steps in the program have to be performed if either of the statements are true then the condition will be coded as:

```
(a < 10) || (b == 7);
```

The third logical operator NOT is represented by a single exclamation mark (!). This operator reverses the true value of the expression. For example, if a program has to do some steps if a variable **s** is less than 10, the condition can be written as

```
(s < 10);
```

as well as

```
(! (s >= 10)) /* s is not equal to or more than 10 */
```

Both relational and logical operators are lower in precedence than the arithmetic operators. For example, **5 > 4 + 3** is evaluated as if it is written as **5 > (4 + 3)**, that is, **4+3** will be evaluated first and then the relational operation will be carried out. The result for this will be false, that is 0 will be returned.

The following statement

```
printf("%d", 5> 4 + 3);
```

will give

```
0
```

as 5 is less than 7 (4 + 3).

4.4 Bitwise Logical Operators and Expressions

Consider an operand 12 for example. Bitwise operator will treat this number 12 as 1100. Bitwise operators treat the operands as bits rather than numerical value. The numerical values could be of any base: decimal, octal or hexadecimal. The Bitwise operator will convert the operand to its binary representation and accordingly return 1 or a 0.

Bitwise logical operators are &, | , ^, ~, etc., the details of each of which are summarized in Table 4.3 below.

Operator	Description
Bitwise AND (x & y)	Each bit position returns a one if bits of both operands are ones.
Bitwise OR (x | y)	Each bit position returns a one if bits of either operand are ones.
Bitwise NOT (~ x)	Reverses the bits of its operand.
Bitwise XOR (x ^ y)	Each bit position returns a one if either bits of the operands are ones but not both.

Table 4.3: Bitwise logical operators

Bitwise operators treat number types as a 32-bit value, which changes bits according to the operator and then converts them back to the number, when done. For example,

The expression 10 & 15 implies (1010 & 1111) which returns a value 1010 which means 10.

The expression 10 | 15 implies (1010 | 1111) which returns a value 1111 which means 15.

The expression 10 ^ 15 implies (1010 ^ 1111) which returns a value 0101 which means 5.

The expression ~10 implies (~1010) which returns a value -11.

4.5 Mixed Mode Expressions & Type Conversions

A mixed mode expression is one in which the operands of an operator belong to different data types. These operands are generally converted to the same type. All operands are converted to the data type of the largest operand. This is called **type promotion**. The order of the various data types is:

`char` < `int` < `long` < `loat` < `double`

The **automatic type** conversions for evaluating an expression are tabulated below

a. `char` and `short` are converted to `int`, and `float` is converted to `double`

b. If either operand is `double`, the other is converted to `double`, and the result is `double`

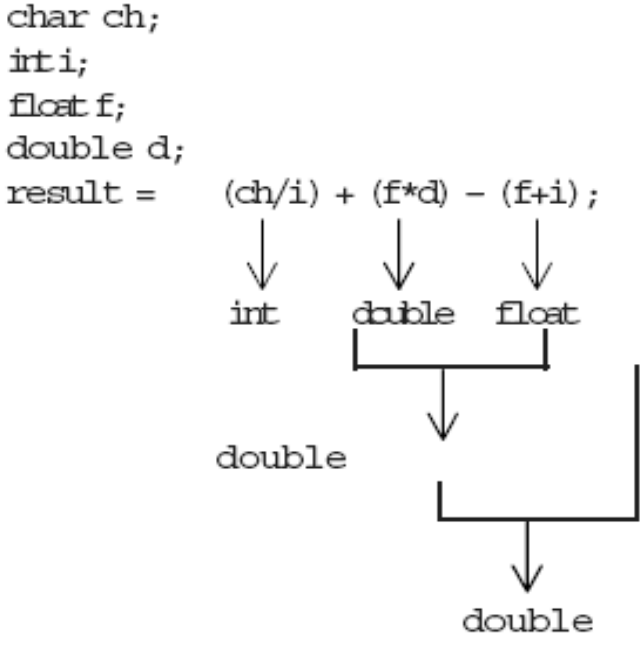
c. If either operand is `long`, the other is converted to `long`, and the result is `double`

d. If either operand is `unsigned`, the other is also converted to unsigned, and the result is also `unsigned`

e. Otherwise all that are left are the operands of type `int`, and the result is `int`

One rule not discussed above is when one operand is `long` and the other is unsigned, and the value of the `unsigned` cannot be represented by `long`. In this case, both operands are converted to `unsigned long`.

Once these conversion rules have been applied, each pair of operands is of the same type and the result of each operation is the same as the type of both operands.



In the above example, first the character **ch** is converted to an integer and `float` **f** is converted to `double`. Then the outcome of **ch/i** is converted to a `double` because **f*d** is `double`. The final result is `double` because, by this time, both operands are `double`.

4.5.1 Casts

It is normally a good practice to change all `integer` constants to `float` if the expression involves arithmetic operations with real values; otherwise some expressions may lose their true value. For example, consider

```
int x,y,z;
x = 10;
y = 100;
z = x/y;
```

In this case **z** will be assigned 0 as integer division takes place and the fractional part (0.10) is truncated.

An expression can be forced to be of a certain type by using a **cast**. The general syntax of cast is:

```
(type) cast
```

where **type** is a valid C data type. For example to make sure that an expression **a/b**, where **a** and **b** are integers, evaluates to a `float`, the following line of code can be written:

Concepts

```
(float) a/b;
```

The cast operator can be used on constants, expressions as well as variables as shown below:

```
(int) 17.487;
(double) (5 * 4 / 8);
(float) (a + 7));
```

In the second example, the cast operator does not serve its purpose because it is performed only after the entire expression within the parentheses is evaluated. The expression `5 * 4 / 8` evaluates to `2` (due to integer truncation), so the resultant `double` value is equal to `2.0`.

Another example:

```
int i = 1, j = 3;
x = i / j; /* x 0.0 */
x = (float) i/(float) j; /* x 0.33 */
```

4.6 Precedence of Operators

Precedence establishes the hierarchy of one set of operators over another when an arithmetic expression has to be evaluated. In short, it refers to the order in which C evaluates operators. The precedence of the arithmetic operators is given in Table 4.4 below.

Operator Class	Operators	Associativity
Unary	- ++ – –	Right to left
Binary	^	Left to Right
Binary	* / %	Left to Right
Binary	+ -	Left to Right
Binary	=	Right to Left

Table 4.4: Order of precedence of the arithmetic operators

Operators in the same line in the above table have the same precedence. The evaluation of operators in an arithmetic expression takes place from left to right for operators having equal precedence. The operators `*`, `/`, and `%` have the same precedence, which is higher than that of `+` and `–` (binary).

The precedence of these operators can be overridden by enclosing the required part of the expression in parentheses (). An expression within parenthesis is always evaluated first. One set of parentheses may be enclosed within another. This is called nesting of parentheses. In such a case, evaluation is done by expanding the innermost parentheses first and then working to the outermost pair of parentheses.

If there are several sets of parentheses in an expression then evaluation takes place from left to right.

Associativity tells how an operator associates with its operands. For example, the unary associates with the quantity to its right, and in division the left operand is divided by the right. The assignment operator (=) associates from right to left. Hence the expression on the right is evaluated first and its value is assigned to the variables on the left.

Associativity also refers to the order in which C evaluates operators in an expression having same precedence. Such operators can operate either left to right or vice versa as shown in Table 4.5.

For example,

```
a = b =10/2;
```

assigns the value `5` to **`b`** which is then assigned to **`a`**. Hence the associativity is said to be from right to left. Also,

```
-8 * 4 % 2 - 3
```

is evaluated in the following sequence:

Sequence	Operation done	Result
1.	- 8 (unary minus)	negative of 8
2.	- 8 * 4	- 32
3.	- 32 % 2	0
4.	0-3	-3

In the above, the unary minus is evaluated first as it has highest precedence. The evaluation of `*` and `%` takes place from left to right. This is followed by evaluation of the binary `-` operator.

Order of Sub-expressions

Complex expressions could contain smaller expressions within them called sub expressions. C does not specify in what order the sub-expressions are evaluated. An expression like

```
a * b / c + d * c;
```

guarantees that the sub-expression **`a * b/c`** and **`d*c`** will be evaluated before addition. Furthermore, the left-to-right rule for multiplication and division guarantees that **`a`** will be multiplied by **`b`** and then the product will be divided by **`c`**. But there is no rule for governing whether **`a*b /c`** is evaluate before or after **`d*c`**. This option is left open to the designers of the compiler to decide. The left-to-right rule just applies to a sequence of operators at the same level. That is, it applies to multiplication and division in **`a*b/c`**. But it ends there since the next operator `+` has a different precedence level.

Because of this indeterminacy, expressions whose final value depends upon the order in which the sub-expressions are evaluated should not be used. Consider the following example,

```
a * b + c * b ++ ;
```

One compiler might evaluate the left term first and use the same value of **b** in both the subexpressions. But, a second compiler might evaluate the right terms first and increment **b** before it is used in the left term.

It is preferable that increment or decrement operators should not be used on a variable that appears more than once in an expression.

Precedence between Comparison Operators

Some arithmetic operators, as we saw in the previous section, have some kind of precedence over others. There is no such precedence among comparison operators. They are therefore always evaluated left to right.

Precedence for Logical Operators

The table below presents the order of precedence for logical operators.

Precedence	Operator
1	NOT
2	A N D
3	OR

Table 4.5: Order of precedence for logical operators

When multiple instances of a logical operator are used in a condition, they are evaluated from right to left.

For example, consider the following condition:

False **OR** True **AND NOT** False **AND** True

This condition gets evaluated as shown below:

1. False **OR** True **AND** [**NOT** False] **AND** True **NOT** has the highest precedence.

2. False **OR** True **AND** [True **AND** True]
   Here, **AND** is the operator of the highest precedence and operators of the same precedence are evaluated from right to left.

3. False **OR** [True **AND** True]

4. [False **OR** True]

5. True

Precedence among the Different Types of Operators

When an equation uses more than one type of operator then the order of precedence has to be established with the different types of operators.

The given table presents the precedence among the different types of operators.

Precedence	Type of Operator
1	Arithmetic
2	Comparison
3	Logical

Table 4.6: Order of precedence among different types of operators

Thus, in an equation involving all three types of operators, the arithmetic operators are evaluated first followed by comparison and then logical. The precedence of operators within a particular type has been defined earlier.

Consider the following example:

```
2*3+4/2 > 3 AND 3<5 OR 10<9
```

The evaluation is as shown:

1. `[2*3+4/2] > 3 AND 3<5 OR 10<9`

   First the arithmetic operators are dealt with. The order of precedence for the evaluation arithmetic operators is as in Table 6.

2. `[[2*3]+[4/2]] > 3 AND 3<5 OR 10<9`

3. `[6+2] >3 AND 3<5 OR 10<9`

4. `[8 >3] AND [3<5] OR [10<9]`

   Next to be evaluated are the comparison operators all of which have the same precedence and so

are evaluated from left to right.

5. `True AND True OR False`

   The last to be evaluated are the logical operators. AND takes precedence over OR.

6. `[True AND True] OR False`

7. `True OR False`

8. `True`

The Parentheses (or brackets)

The precedence of operators can be modified by using the parentheses to specify which part of the equation is to be solved first.

- When there are parentheses within parentheses, the innermost parentheses are to be evaluated first.
- When an equation involves multiple sets of parentheses, they are evaluated left to right.

Consider the following example:

```
5+9*3^2-4 > 10 AND (2+2^4-8/4 > 6 OR (2<6 AND 10>11))
```

The solution is:

1. `5+9*3^2-4 > 10 AND (2+2^4-8/4 > 6 OR (True AND False))`

   The inner parenthesis takes precedence over all other operators and the evaluation within this is as per the regular conventions.

2. `5+9*3^2-4 > 10 AND (2+2^4-8/4 > 6 OR False)`

3. `5+9*3^2-4 > 10 AND (2+16-8/4 > 6 OR False)`

   Next the outer parentheses is evaluated Refer to the tables for the precedence of the various operators used.

4. `5+9*3^2-4 > 10 AND (2+16-2 > 6 OR False)`

5. `5+9*3^2-4 > 10 AND (18-2 > 6 OR False)`

6. `5+9*3^2-4 > 10 AND (16 > 6 OR False)`

7. `5+9*3^2-4 > 10 AND (True OR False)`

8. `5+9*3^2-4 > 10 AND True`

9. `5+9*9-4>10 AND True`

   The expression to the left is evaluated as per the conventions.

10. `5+81-4>10 AND True`

11. `86-4>10 AND True`

12. `82>10 AND True`

13. `True AND True`

14. `True`

Concepts

Summary

- C defines four classes of operators: arithmetic, relational, logical, and bitwise.
- All operators in C follow a precedence order.
- Relational operators are used to test the relationship between two variables, or between a variable and a constant.
- Logical operators are symbols that are used to combine or negate expressions containing relational operators.
- Bitwise operators treat the operands as bits rather than numerical value.
- Assignment (=) is considered an operator with right to left associativity.
- Precedence establishes the hierarchy of one set of operators over another when an expression has to be evaluated.

Check Your Progress

1. _____ are the tools that manipulate data.

   A. Operators B. Operands

   C. Expressions D. None of the above

2. An _____ consists of a combination of operators and operands.

   A. Expression B. Functions

   C. Pointers D. None of the above

3. _____ establishes the hierarchy of one set of operators over another when an arithmetic expression has to be evaluated.

   A. Operands B. Precedence

   C. Operator D. None of the above

4. _____ is one in which the operands of an operator belong to different data types.

   A. Single Mode Expression B. Mixed Mode Expression

   C. Precedence D. None of the above

5. An expression can be forced to be of a certain type by using a _____.

   A. Cast B. Precedence

   C. Operator D. None of the above

6. _____ are used to combine or negate expressions containing relational operators.

   A. Logical Operators B. Bitwise Operators

   C. Complex Operators D. None of the above

Check Your Progress

7. Bitwise logical operators are __, ___, __ and __ .

   A. % , ^ , * and @ B. &, |, ~ and ^

   C. !,], & and * D. None of the above

8. The precedence of operators can be overridden by enclosing the required part of the expression in _____.

   A. Curly Brackets B. Caret Symbol

   C. Parentheses D. None of the above

Try It Yourself

1. Write a program to accept and add three numbers.

2. For the following values, write a program to evaluate the expression

```
z = a*b+(c/d)-e*f;
```

a=10

b=7

c=15.75

d=4

e=2

f=5.6

3. Write a program to evaluate the area and perimeter of the rectangle.

4. Write a program to evaluate the volume of a cylinder.

5. Write a program to evaluate the net salary of an employee given the following constraints:

Basic salary : $ 12000

DA : 12% of Basic salary

HRA : $150

TA : $120

Others : $450

Tax cuts – a) PF :14% of Basic salary and b) IT: 15% of Basic salary

Net Salary = Basic Salary + DA + HRA + TA + Others – (PF + IT)

“Learn all you can from the mistakes of others. You won’t have time to make them all yourself”

Session

5 Operators and Expressions (Lab)

Objectives

At the end of this session, you will be able to:

- *Use of the arithmetic, comparison, and logical operators*
- *Use type conversions*
- *Understand the precedence among operators*

The steps given in this session are carefully thought and explained in great details so that the understanding of the tool is complete. Follow the steps carefully.

In this section you will write a program to calculate the simple interest on a loan taken.

The formula for calculating the simple interest is **`p * n * r /100`**. Here '**`p`**' stands for principal amount, '**`n`**' stands for number of years and '**`r`**' stands for rate of interest.

The program declares three '**`float`**' variables named **`p`**, **`n`** and **`r`**. Note that the variables are declared in the same line of code using a comma (,) to separate one from the other. Each of these variables is assigned (or given) a value.

Consider the following line of code:

```
printf("\nAmount is : %f", p*n*r/100);
```

In '`printf( )`' above, we have used '`%f`'. '`%f`' is used to display the value of the '`float`' variables mentioned after the comma at the end of the '`printf()`'. In '`printf( )`' the formula to calculate the simple interest is given. That is **`p`**, **`n`** and **`r`** are multiplied and the product is then divided by **`100`**. Thus '`printf()`' displays the amount of simple interest.

Invoke Borland C.

5.1 Calculating Simple Interest

1. **Create a new file.**
2. **Type the following code in the 'Edit window' :**

```
#include <stdio.h>
#include <conio.h>
void main()
{
    float p,n,r;
    clrscr();
    p = 1000;
    n = 2.5;
    r = 10.5;
    printf("\n Amount is : %f", p*n*r/100);
}
```

3. **Save the file with the name `simple.c`**

4. **Compile the file `simple.c`**

5. **Execute the program `simple.c`**

6. **Return to the editor.**

OUTPUT :

```
The Amount is: 262.500000
```

5.2 Using Arithmetic Operators

In this section, you will write a program using the arithmetic operators.

This program declares four 'integer' variables named **a**, **b**, **c** and **d**. Value is assigned to the variables **a**, **b** and **c**.

Consider the following line of code:

```
d = a*b+c/2;
```

a is multiplied by **b**. **c** is divided by **2**. Then the product of **a*b** is added to the quotient (or division) of **c/2**. This value is assigned to the variable **d** using the (=) operator. The expression is evaluated as :

1. `50 * 24 = 1200`

2. `68 / 2 = 34`

3. `1200 + 34 = 1234`

4. `d = 1234`

'`printf( )`' displays the value of the variable **d**.

See the last expression:

```
d = a*(b+c+(a-c)*b);
```

Here, the inner parenthesis (or brackets) is having the highest precedence. So **(a-c)** is calculated first. After that the outer parentheses expression is calculated in the manner explained. The result of **(a-c)** is multiplied with **b** because '*' has the highest precedence than '–' and '+' . The expression is evaluated as:

1. **50*(24+68+(50-68)*24)**

2. **50*(24+68+-18*24)**

3. **50*(24+68+-432)**

4. **50*(92-432)**

5. **50*-340**

6. **d = -17000**

Other expressions are evaluated according to the operators used. The result is displayed by the 'printf( )'.

1. Create a new file.

2. Type the following code in the 'Edit Window':

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int a,b,c,d;
    clrscr();
    a = 50;
    b = 24;
```

```
    c = 68;
    d = a*b+c/2;
    printf("\n The value after a*b+c/2 is : %d",d);
    d = a%b;
    printf("\n The value after a mod b is : %d",d);
    d = a*b-c;
    printf("\n The value after a*b-c is : %d",d);
    d = a/b+c;
    printf("\n The value after a/b+c is : %d",d);
    d = a+b*c;
    printf("\n The value after a+b*c is : %d",d);
    d = (a+b)*c;
    printf("\n The value after (a+b)*c is : %d",d);
    d = a*(b+c+(a-c)*b);
    printf("\n The value after a*(b+c+(a-c)*b) is : %d",d);
}
```

3. **Save the file with the name `arith.c`**

4. **Compile the file `arith.c`**

5. **Execute the program `arith.c`**

6. **Return to the editor.**

OUTPUT :

```
The value after a*b+c/2 is : 1234
The value after a mod b is : 2
The value after a*b-c is : 1132
The value after a/b+c is : 70
The value after a+b*c is : 1682
The value after (a+b)*c is : 5032
The value after a*(b+c+(a-c)+b) is : -17000
```

5.3 Using Comparison and Logical operators

In this section you will write a program using the comparison and logical operators.

Three integer variables named **a**, **b** and **c** are declared in this program. Values are assigned to the

variables. Let **a** = **5**, **b** = **6** & **c** = **7**.

Consider the following lines of code:

1. `a + b >= c;`

   First **a+b** will be calculated (arithmetic operators have higher precedence than comparison operators) i.e. , **11**. Next the value **11** is compared with **c**. The result is **1** (**true**) because **11 > 7**.

2. Consider another expression:

   `a > 10 && b < 5 ;`

Here the first calculation will be `a > 10` and `b < 5`, because comparison operators (> <) have more precedence than logical AND operator (`&&`). The equation will be evaluated as:

1. 5 > 10 && 6 < 5

2. **FALSE && FALSE**

3. **FALSE** i.e., 0

1. Create a new file.

2. Type the following code in the 'Edit Window':

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int a = 5, b = 6, c = 7;
    printf ("int a = 5, b = 6, c = 7;\n");
    printf("The value of a > b is \t%i\n\n", a > b);
    printf("The value of b < c is \t%i\n\n", b < c);
    printf("The value of a + b >= c is \t%i\n\n", a + b >= c);
    printf("The value of a - b <= b - c is\t%i\n\n", a-b<=b-c);
    printf("The value of b-a =b - c is\t%i\n\n", b - a == b - c);
    printf("The value of a*b!= c * c is\t%i\n\n", a * b < c * c);
    printf("Result of a>10 && b <5 = %d\n\n",a>10&&b<5);
    printf("Result of a>100 || b<50=%d\n\n",a>100 || b<50);
}
```

3. **Save the file with the name `compare.c`**

4. **Compile the file `compare.c`**

5. **Execute the program `compare.c`**

6. **Return to the editor.**

OUTPUT :

```
int a = 5, b = 6, c = 7;
The value of a > b is 0
The value of b < c is 1
The value of a + b >= c is 1
The value of a - b <= b - c is 1
The value of b - a == b - c is 0
The value of a * b != c * c is 1
Result of a > 10 && b <5 = 0
Result of a>100 || b<50 = 1
```

5.4 Using Type-conversion

In this section you will write a program to understand type conversion.

In the first expression, all are entirely '`int`' context, `40 / 17 * 13 / 3` would evaluate to `8 (40 / 17` rounds to `2, 2 * 13 = 26`, `26 / 3` rounds to `8`)

The second expression:

1. to evaluate `40 / 17 * 13 / 3.0`

2. `40 / 17` again rounds to `2`

3. 2 * 13 = 26

4. but `3.0` forces the final division into a double context, thus `26.0 / 3.0 = 8.666667`

In the third expression, if we move the decimal point to `13 (40 / 17 * 13.0 / 3)`, the result will still be `8.666667` because:

1. `40 / 17` rounds to `2`

2. the `13.0` forces the multiplication to a double but `2.0 * 13.0` still equals `26.0`

3. and `26.0` still forces the final division into a double context, so `26.0 / 3.0 = 8.666667`

In the last expression, if we move the decimal point to `17 (40 / 17.0 * 13 / 3)`, the result now becomes `10.196078` because:

1. `17.0` forces the initial division into a double context and `40.0 / 17.0 = 2.352941`

2. `2.352941 * 13.0 = 30.588233`

3. and `30.588233 / 3.0 = 10.196078`

1. Create a new file.

2. Type the following code in the 'Edit Window' :

```
#include <stdio.h>
#include <conio.h>
void main()
{
    clrscr();
    printf("40/17*13/3 = %d",40/17*13/3);
    printf("\n\n40/17*13/3.0 = %lf",40/17*13/3.0);
    printf("\n\n40/17*13.0/3 = %lf",40/17*13.0/3);
    printf("\n\n40/17.0*13/3 = %lf",40/17.0*13/3);
}
```

3. Save the file with the name `type.c`

4. Compile the file `type.c`

5. Execute the program `type.c`

6. Return to the editor.

OUTPUT :

```
40/17*13/3 = 8
40/17*13/3.0 = 8.666667
40/17*13.0/3 = 8.666667
40/17.0*13/3 = 10.196078
```

5.5 Precedence of operators

In this section you will write a program to see the precedence among operators.

The following expression will be evaluated as follows:

```
(4-2*9/6<=3 && (10*2/4-3>3|| (1<5 && 8>10)))
```

Follow the rules that we have studied in the session “Operators and Expressions”

(Note that the expressions shown as bold below are evaluated first)

1. (4-2*9/6<=3 && (10*2/4-3>3|| **(1<5 && 8>10)**))
2. (4-2*9/6<=3 && (10*2/4-3>3|| **(1 && 0)**))
3. (4-2*9/6<=3 && **(10*2/4-3>3|| 0)**)
4. (4-2*9/6<=3 && **(20/4-3>3|| 0)**)
5. (4-2*9/6<=3 && **(5-3>3|| 0)**)
6. (4-2*9/6<=3 && **(2>3|| 0)**)
7. (4-2*9/6<=3 && **(0|| 0)**)
8. (4-**2*9**/6<=3 && 0)
9. (4-**18/6**<=3 && 0)
10. (**4-3**<=3 && 0)
11. (**1<=3** && 0)
12. (**1 && 0**)
13. **0** (**False**)

1. Create a new file.
2. Type the following code in the ‘Edit Window’ :

```
#include <stdio.h>
#include <conio.h>
void main()
{
    clrscr();
    printf("Result=%d",(4-2*9/6<=3&&(10*2/4-3>3||(1<5&&8>10))));
}
```

3. **Save the file with the name `precede.c`**

4. **Compile the file `precede.c`**

5. **Execute the program `precede.c`**

6. **Return to the editor.**

OUTPUT :

```
Result = 0
```

Part II: For the next 30 Minutes:

1. Solve the following expression:

   `10 * 3 ^ 6 * 6 + 5 – 2 AND (2 * 2 + 6 /3 > 1 OR 2 >8)`

   To do this :

   Type the above expression using `printf()` statement . AND can be replaced by `&&` (ampersand symbol) and OR can be replaced by `||` (double pipe symbol)

2. Assume all the variables are of type `int`. Find the values of each of the following variables:

```
a.  x = (2+3) * 6;

b.  x = (12 + 6) /2 *3;

c.  y = x = (2 + 3) / 4;

d.  y = 3 + 2 *(x = 7/2);

e.  x = (int)3.8 + 3.3;

f.  x = (2 + 3) * 10.5;

g.  x = 3/5 * 22.0;

h.  x = 22.0 * 3/5;
```

Try It Yourself

1. What is the assigned (left-hand side) value in each case?

```
int s, m=3, n=5, r, t;

float x=3.0, y;

t = n/m;

r = n%m;

y = n/m;

t = x*y-m/2;

x = x*2.0;

s = (m+n)/r;

y = --n;
```

2. Write a program which will take the input as a floating (real) number. This number represents the centimeters. Print out the equivalent number of feet(floating, 1 decimal) and inches (floating, 1 decimal), with feet and the inches given to an accuracy of one decimal place.

   Assume 2.54 centimeters per inch, and 12 inches per foot.

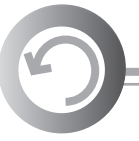
   If the input value is 333.3, the output format should be:

   333.3 centimeters is 10.9 feet

   333.33 centimeters is 131.2 inches

3. Find the value of iResult for the following assignment statements:

```
int iResult, a = 10, b = 8, c = 6, d = 5, e =2;
```



Try It Yourself

```
iResult = a - b - c - d;

iResult = a - b + c - d;

iResult = a + b / c / d;

iResult = a + b / c * d;

iResult = a / b * c * d;

iResult = a % b / c * d;

iResult = a % b % c % d;

iResult = a - (b - c) - d;

iResult = (a - (b - c)) - d;

iResult = a - ((b - c) - d);

iResult = a % (b % c) * d * e;

iResult = a + (b - c) * d - e;

iResult = (a + b) * c + d * e;

iResult = (a + b) * (c / d) % e;
```

Session

6 Input and Output in 'C'

Concepts

Objectives

At the end of this session, you will be able to:

- *To understand formatted I/O functions scanf() and printf()*
- *To use character I/O functions getchar() and putchar()*

Introduction

In any programming language, assigning values to variables and printing them after processing can be done in two ways,

- Through standard Input/ Output (I/O) media
- Through files

This session covers basic input and output. Usually input and output (I/O), form an important part of any program. To do anything useful, your program needs to be able to accept input data and report back your results. In C, the standard library provides routines for input and output. The standard library has functions for I/O that handle input, output, and character and string manipulation. In this lesson, all the input functions described read from standard input and all the output functions described write to standard output. Standard input is usually the keyboard. Standard output is usually the monitor (also called the console). Input and Output can be rerouted from or to files instead of the standard devices. The files may be on a disk or on any other storage medium. The output may be sent to the printer also.

6.1 The Header File <stdio.h>

In the examples, given so far you would have seen the following line,

```
#include <stdio.h>
```

This is a preprocessor command. In standard C, the `#` should be in the first column. **`stdio.h`** is a file and is called the header file. It contains the macros for many of the input/output functions used in C. The `printf()`, `scanf()`, `putchar()` and `getchar()` functions are designed in such a way that they require the macros in **`stdio.h`** for proper execution.

Concepts

6.2 Input and Output in C

The standard library in C provides two functions that perform formatted input and output. They are:

- `printf()` – for formatted output
- `scanf()` – for formatted input

These functions are called formatted functions as they can read and print data in various prespecified formats which are under the user's control. Format specifiers specify the format in which the values of the variables are to be input and printed.

6.2.1 printf()

You are already familiar with this function, since it has been used in the earlier sessions. We shall have a detailed look at the function here. The function `printf()` is used to display data on the standard output – console. The general format of the function is:

```
printf("control string", argument list);
```

The argument list consists of constants, variables, expressions or functions separated by commas. There must be one format command in the control string for each argument in the list. The format commands must match the argument list in number, type and order. The control string must always be enclosed within double quotes(" "), which are its delimiters. The control string consists of one or more of three types of items which are explained below:

- **Text characters** – This consists of printable characters that are to be printed as they are. Spaces are often used to separate output fields.
- **Format Commands** define the way the data items in the argument list are to be displayed. A format command begins with a % sign and is followed by a format code appropriate for the data item. % is used by the `printf()` function to identify conversion specifications. The format commands and the data items are matched in order and typed from left to right. One format code is required for every data item that has to be printed.
- **Nonprinting Characters** – This includes tabs, blanks and new lines.

Each format command consists of one or more format codes. A format code consists of a % and a type specifier. Table 6.1 lists the various format codes supported by the `printf()` statement:

Format	printf()	scanf()
Single Character	%c	%c
String	%s	%s

Format	printf()	scanf()
Signed decimal integer	%d	%d
Floating point (decimal notation)	%f	%f or %e
Floating point (decimal notation)	%lf	%lf
Floating point (exponential notation)	%e	%f or %e
Floating point (%f or %e , whichever is shorter)	%g	
Unsigned decimal integer	%u	%u
Unsigned hexadecimal integer (uses "ABCDEF")	%x	%x
Unsigned octal integer	%o	%o

Table 6.1: printf() Format Codes

In the above table `c`, `d`, `f`, `lf`, `e`, `g`, `u`, `s`, `o` and `x` are the type specifiers.

The printing conventions of the various format codes are summarized in Table 6.2:

Format Code	Printing Conventions
%d	The number of digits in the integer.
%f	The integer part of the number will be printed as such. The decimal part will consist of 6 digits. If the decimal part of the number is smaller than 6, it will be padded with trailing zeroes to the right, else it will be rounded at the right.
%e	One digit to the left of the decimal point and 6 places to the right , as in %f above.

Table 6.2: Printing Conventions

Since %, \ and " have special uses in the control string , if they have to be inserted as part of a text string and printed on the console, they must be used as seen in Table 6.3 :

\\	to print \ character
\	" to print " character
% %	to print % character

Table 6.3: Control String Special Characters

Concepts

No	Statements	Control String	What the control string contains	Argument List	Explanation of the argument list	Screen Display
1.	printf("%d",300);	%d	Consists of format command only	300	Constant	300
2.	printf("%d",10+5);	%d	Consists of format command only	10 + 5	Expression	15
3.	printf("Good Morning Mr. Lee.");	Good Morning Mr. Lee.	Consists of text characters only	Nil	Nil	Good Morning Mr. Lee.
4.	int count = 100; printf("%d",count);	%d	Consists of format command only	Count	Variable	100
5.	printf("\nhello");	\nhello	Consists of nonprinting character & text characters	Nil	Nil	hello on a new line
6.	#define str " Good Apple " ..... printf("%s",str);	%s	Consists of format command only	Str	Symbolic constant	Good Apple

7.	.....
	int count,stud_num;
	count=0;
	stud_nim=100;
	printf("%d %d\n",
	count, stud_num);
		%d %d	Consists of format command and escape sequence	count, stud_num	Two variables	0 , 100

Table 6.4: Control Strings and Format Codes

Example 1:

```
/* This is a simple program which demonstrates how a string can be printed
from within a format command and also as an argument. This program also
displays a single character, integer, and float. */

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

```
    float b = 24.67892345;
    char ch = 'A';
    printf("Integer data = %d", a);
    printf("Float Data = %f",b);
    printf("Character = %c",ch);
    printf("This prints the string");
    printf("%s","This also prints a string");
}
```

A sample run is shown below:

```
Integer data = 10
Float Data = 24.678923
Character = A
This prints the string
This also prints a string
```

➢ **Modifiers for Format Commands in printf()**

The format commands may have modifiers, to suitably modify the basic conversion specifications. The following are valid modifiers acceptable in the `printf()` statement. If more than one modifier is used, then they must be in the same order as given below.

'-' Modifier

The data item will be left-justified within its field; the item will be printed beginning from the leftmost position of its field.

Field Width Modifier

They can be used with type `float`, `double` or `char` array (`string`). The field width modifier, which is an integer, defines the minimum field width for the data item. Data items for smaller width will be output right-justified within the field. Larger data items will be printed by using as many extra positions as required. e.g. **`%10f`** is the format command for a type `float` data item with minimum field width 10.

Precision Modifier

This modifier can be used with type `float`, `double` or `char` array (`string`). The modifier is written as **`.m`** where **`m`** is an integer. If used with data type `float` or `double`, the digit string indicates the maximum number of digits to be printed to the right of the decimal. When used with a string, it indicates the maximum number of characters to be printed.

If the fractional part of a type `float` or `double` data item exceeds the precision modifier, then the number will be rounded. If a string length exceeds the specified length, then the string will be truncated (cut of at the end). Padding with zeroes occurs for numbers when the actual number of digits in a data item is less than that specified by the modifier. Similarly blanks are padded for strings. e.g. **`%10.3f`** is the format command for a type float data item, with minimum field width 10 and 3 places after the decimal.

'0' Modifier

The default padding in a field is done with spaces. If the user wishes to pad a field with zeroes, this modifier must be used.

'1' Modifier

This modifier can be used to display integers as long int or a double precision argument. The corresponding format code is `%ld`.

'h' Modifier

This modifier is used to display short integers. The corresponding format code is `%hd`.

'*' Modifier

This modifier is used if the user does not want to specify the field width in advance, but wants the program to specify it. But along with this modifier an argument is required which tells what the field width should be.

Let us now see how these modifiers work. First we see their effect when used with integer data items.

Example 2:

```
/* This program demonstrates the use of Modifiers in printf() */
#include <stdio.h>
void main()
{
    printf("The number 555 in various forms:\n");
    printf("Without any modifier: \n");
    printf("[%d]\n",555);
    printf("With - modifier :\n");
    printf("[%-d]\n",555);
```

```
    printf("With digit string 10 as modifier :\n");
    printf("[%10d]\n",555);
    printf("With 0 as modifier : \n");
    printf("[%0d]\n",555);
    printf("With 0 and digit string 10 as modifiers :\n");
    printf("[%010d]\n",555);
    printf("With -, 0 and digit string 10 as modifiers: \n");
    printf("[%-010d]\n",555);
}
```

A sample run is shown below :

```
The number 555 in various forms:
Without any modifier:
[555]
With - modifier :
[555]
With digit string 10 as modifier :
[555]
With 0 as modifier :
[555]
With 0 and digit string 10 as modifiers :
[0000000555]
With -, 0 and digit string 10 as modifiers:
[555]
```

We have used `[` and `]` to show where the field begins and where it ends. When we use `%d` with no modifiers, we see that it uses a field with the same width as the integer. When using `%10,d` we see that it uses a field 10 spaces wide and the number is right-justified, by default. If we use the – modifier, the number is left-justified in the same field. If we use the 0 modifier, we see that the number is padded with 0's instead of blanks.

Now let us see how modifiers can be used with floating-point numbers.

Example 3:

```
/* This program demonstrates the use of Modifiers in printf() */
#include <stdio.h>
void main()
{
```

Concepts

```
    printf("The number 555.55 in various forms:\n");
    printf("In float form without modifiers :\n");
     printf("[%f]\n",555.55); printf("In exponential form without any
modifier: \n");
    printf("[%e]\n",555.55);
    printf("In float form with - modifier:\n");
    printf("[%-f]\n",555.55);
    printf("In float form with digit string 10.3 as
modifier\n");
    printf("[%10.3f]\n",555.55);
    printf("In float form with 0 as modifier: \n");
    printf("[%0f]\n",555.55);
    printf("In float form with 0 and digit string 10.3 ");
    printf("as modifiers:\n");
    printf("[%010.3f]\n",555.55);
    printf("In float form with -, 0 ");
    printf("and digit string 10.3 as modifiers: \n");
    printf("[%-010.3f]\n",555.55);
    printf("In exponential form with 0 ");
    printf("and digit string 10.3 as modifiers:\n");
    printf("[%010.3e]\n",555.55);
    printf("In exponential form with -, 0 ");
    printf("and digit string 10.3 as modifiers : \n");
    printf("[%-010.3e]\n\n",555.55);
}
```

A sample output is shown below :

```
The number 555.55 in various forms:
In float form without modifiers:
[555.550000]
In exponential form without any modifier:
[5.555500e+02]
In float form with - modifier:
[555.550000]
In float form with digit string 10.3 as modifier
[555.550]
```

```
In float form with 0 as modifier:
[555.550000]
In float form with 0 and digit string 10.3 as modifiers:
[000555.550]
In float form with -, 0 and digit string 10.3 as modifiers:
[555.550]
In exponential form with 0 and digit string 10.3 as modifiers:
[05.555e+02]
In exponential form with -,0 and digit string 10.3 as modifiers :
[5.555e+02]
```

In the default version of `%f`, we can see that there are six decimal digits and the default specification of `%e` is one digit to the left of the decimal point and six digits to the right of the decimal point. Notice how in the last two examples, the number of digits to the right of the decimal point being 3, causes the output to be rounded off.

Now let us see how modifiers can be used with strings. Notice how the field is expanded to contain the entire string. Also note how the precision specification `.4` limits the number of characters to be printed.

Example 4:

```
/* Program to show the use of modifiers with strings */
#include <stdio.h>
void main()
{
    printf("A string in various forms :\n");
    printf("Without any format command :\n");
    printf("Good day Mr. Lee. \n");
    printf("With format command but without any modifier:\n");
    printf("[%s]\n","Good day Mr. Lee.");
    printf("With digit string 4 as modifier :\n");
    printf("[%4s]\n","Good day Mr. Lee.");
    printf("With digit string 19 as modifier: \n");
    printf("[%19s]\n","Good day Mr. Lee.");
    printf("With digit string 23 as modifier: \n");
    printf("[%23s]\n","Good day Mr. Lee.");
    printf("With digit string 25.4 as modifier: \n");
    printf("[%25.4s]\n","Good day Mr.Lee.");
```

```
    printf("With - and digit string 25.4 as modifiers :\n");
    printf("[%-25.4s]\n","Good day Mr.shroff.");
}
```

A sample output is shown below:

```
A string in various forms :
Without any format command :
Good day Mr. Lee.
With format command but without any modifier:
[Good day Mr. Lee.]
With digit string 4 as modifier :
[Good day Mr. Lee.]
With digit string 19 as modifier:
[Good day Mr. Lee.]
With digit string 23 as modifier:
[ Good day Mr. Lee.]
With digit string 25.4 as modifier:
[ Good]
With - and digit string 25.4 as modifiers :
[Good ]
```

The characters we type at the keyboard are not stored as characters. Instead they are stored as numbers in the **ASCII (American Standard Code for Information Interchange)** code format. The values that a variable holds are interpreted as a character or a numeric depending on the type of the variable that holds it . The following example demonstrates this:

Example 5:

```
#include <stdio.h>
void main()
{
    int a = 80;
    char b= 'C';
    printf("\nThis is the number stored in 'a' %d",a);
    printf("\nThis is a character interpreted from 'a' %c",a);
    printf("\nThis is also a character stored in 'b' %c",b);
    printf("\nHey!the character of 'b' is printed as a number!%d",b);
}
```

A sample output is shown below:

```
This is the number stored in ‘a’ 80
This is a character interpreted from ‘a’ P
This is also a character stored in ‘b’ C
Hey! the character of ‘b’ is printed as a number !67
```

This result demonstrates the use of format specifications and the interpretation of ASCII codes. Though the variables **a** and **b** have been declared as `int` and `char` variables, they have been printed as character and number using different format specifiers. This feature of C gives flexibility in the manipulation of data.

How can you use the `printf()` statement to print a string which extends beyond a line of 80 characters? You must terminate each line by a \ symbol as shown in the example below:

Example 6

```
/* Program demonstrates how to print a long string */
#include <stdio.h>
void main()
{
    printf(“aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\aaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaa\aaaaaaaaaaaaaaaaaaaaaaaaaaaaa\aaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\aaaaaaaaaaaaaaaaaaaaaaaaaaaaa”);
}
```

A sample output is shown below:

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

In the above example, the string in the `printf()` statement is 252 characters long. Since a line of text contains 80 characters, the string extends beyond three lines in the output shown above.

6.2.2 scanf()

The `scanf()` function is used to accept data. The general format of the `scanf()` function is as given below.

Concepts

```
scanf("control string", argument list);
```

The format used in the `printf()` statement are used with the same syntax in the `scanf()` statements too.

The format commands, modifiers and argument list discussed for `printf()` is valid for `scanf()` also, subject to the following differences:

➢ **Differences in argument list of between printf() and scanf()**

`printf()` uses variable names, constants, symbolic constants and expressions, but `scanf()` uses pointers to variables. A pointer to a variable is a data item which contains the address of the location where the variable is stored in memory. Pointers will be discussed in detail later. When using `scanf()` follow these rules, for the argument list:

- If you wish to read in the value of a variable of basic data type, type the variable name with & symbol before it.
- When reading the value of a variable of derived data type, do not use & before the variable name.

➢ **Differences in the format commands of the printf() and scanf()**

a. There is no `%g` option.

b. The `%f` and `%e` format codes are in effect the same. Both accept an optional sign, a string of digits with or without a decimal point, and an optional exponent field.

How scanf() works?

`scanf()` uses nonprinting characters like blank, tab, new line to decide when an input field ends and input field begins. It matches the format commands with the fields in the argument list in the same order in which they are specified, skipping over the white space characters in between. Therefore the input can be spread over more than one line, as long as we have at least one tab, space or new line between each input field. It skips white spaces and line boundaries to obtain the data.

Example 7:

The following program demonstrates the use of the `scanf()` function.

```
#include <stdio.h>
void main()
{
```

```
   int a;
   float d;
   char ch, name[40];
   printf("Please enter the data\n");
   scanf("%d %f %c %s", &a, &d, &ch, name);
   printf("\n The values accepted are : %d, %f, %c, %s", a, d, ch, name);
}
```

A sample output is shown below:

```
Please enter the data
12 67.9 F MARK
The values accepted are :12, 67.900002, F, MARK
```

The input could have been

```
12 67.9

F MARK
```

or even

```
12

67.9

F

MARK
```

for it to be properly accepted into **a**, **d**, **ch**, and **name**.

Consider another example:

Example 8:

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

```
   float x;
   char c;
   ……………………
   scanf("%3d %5f %c",&i,&x, &c);
}
```

If the data items are entered as

```
21 10.345 F
```

When the program is executed, then 21 will be assigned to i , 10.34 will be assigned to **x** and the character 5 will be assigned to **c**. The remaining character **F** will be ignored.

If you specify a field width in `scanf()`, say `%10s`, then `scanf()` collects upto 10 characters or to the first white space character, whichever comes first. This is true for type `int`, `float` and `double` as well.

The example below demonstrates the use of the `scanf()` function to enter a string consisting of uppercase letters and blank spaces. The string will be of undetermined length, but it will be limited to 79 characters (actually, 80 characters including the null character that is added at the end).

Example 9:

```
#include <stdio.h>
void main()
{
    char line[80]; /* line[80] is an array which stores 80 characters */
    …………
    scanf("%[ ABCDEFGHIJKLMNOPQRSTUVWXYZ]",line);
    …………
}
```

The format code `%[]` means that characters defined within `[]` can be accepted as valid string characters. If the string **BEIJING CITY** is entered from the standard input device when the program is executed, the entire string will be assigned to the array line since the string is composed entirely of uppercase letters and blank spaces. If the string was written as Beijing City however, then only the single letter **B** would be assigned to line, since the first lowercase letter (in this case, **e**) would be interpreted as the first character beyond the string.

To accept any character up to a new line character, we use the format code **%[^\n]**, which implies that the string will accept any character except `\n`, which is a new line. The caret (^) implies that all characters except those following the caret will be accepted as valid input characters.

Example 10:

```
#include <stdio.h>
void main()
{
    char line[80];
    ……————
    scanf("%[^\n]",line);
    ……————
}
```

When the `scanf()` function is executed, a string of undetermined length (but not more than 79 characters) will be entered from the standard input device and assigned to line. There will be no restriction on the characters that compose the string, except that they all fit on one line. For example, the string

```
All's well that ends well !
```

could be entered from the keyboard and assigned to line.

The `*` modifier works differently in `scanf()`. The asterisk is used to indicate that a field is to be ignored or skipped.

For example consider the program:

Example 11:

```
#include <stdio.h>
void main()
{
    char item[20];
    int partno;
    float cost;
    ……————————
    scanf("%s %*d %f",item, &partno, &cost);
    ……————
}
```

If the corresponding data items are

```
battery 12345 0.05
```

then **`battery`** will be assigned to **`item`** and 0.05 will be assigned to cost but 12345 will not be assigned

Concepts

to **partno** because of the assignment suppression character asterisk(*).

Any other character in `scanf()`, which is not part of the format code within the control string, must be typed in input in exactly the same way otherwise it causes errors. This feature is used to accept comma(,) delimited input.

For example consider the data stream

```
10, 15, 17
```

and the input command

```
scanf("%d, %f, %c", &intgr, &flt, &ch);
```

Note that the commas in the conversion string, match the commas in the input stream and hence will serve as delimiters.

White space characters in the control string are normally ignored except that it causes problems with **%c** format code. If we use **%c** specifier, then a space is considered a valid character.

Consider the code segment:

```
int x, y;
char ch;
scanf("%2d %c %d",&x, &ch, &y);
printf("%d %d %d\n",x, ch, y);
```

with the input

```
14 c 5
```

14 will be assigned to **x**, the character **ch** has the space (decimal 99) as its value, and **y** is assigned the value of character **c** which is decimal 99.

Consider the following code:

```
#include <stdio.h>
void main()
{
    char c1, c2, c3;
    ............
```

```
    scanf("%c%c%c",&c1, &c2, &c3);
    ..................
}
```

If the input data is

```
a b c
```

(with blank spaces between the letters), then the following assignments would result

```
c1 = a, c2 = <blank space>, c3 = b
```

Here we can see **c2** contains a blank space because the input contains white space character. To skip over such white space characters and read the next non-white space character, the conversion group **%1s** should be used.

```
scanf("%c%1s%1s",&c1, &c2, &c3);
```

Then the same input of data would result in the following assignments

```
c1 = a, c2 =b, c3 = c
```

as intended.

6.3 Buffered I/O

C language as such does not define input and output operations by itself. All I/O operations are performed by the functions available in the C function library. C function library contains one distinct system of routine that handles these operations. That is:

Buffered I/O – used to read and write ASCII characters

A buffer is a temporary storage area, either in the memory or on the controller card for the device. In buffered I/O, characters typed at the keyboard are collected until the user presses the return or the enter key, when the characters are made available to the program, as a block.

Buffered I/O can be further subdivided into:

- Console I/O
- Buffered File I/O

Concepts

Console I/O refers to operations that occur at the keyboard and the screen of your computer. Buffered File I/O refers to operations that are performed to read and write data onto a file. We will discuss the Console I/O.

In C, console is considered as a stream device. The Console I/O functions direct their operations to the standard input and output of the system.

The simplest Console I/O functions are:

- `getchar()` – which reads one (and only one) character from the keyboard.
- `putchar()` – which outputs a single character on the screen.

6.3.1 getchar()

The function `getchar()` is used to read input data, a character at a time from the keyboard. In most implementations of C, `getchar()` buffers characters until the user types a carriage return. Therefore it waits until the return key is pressed. The `getchar()` function has no argument, but the parentheses must still be present. It simply fetches the next character and makes it available to the program. We say that the function returns a value, which is a character.

The following program demonstrates the use of the `getchar()` function.

Example 12:

```
/* Program to demonstrate the use of getchar() */
#include <stdio.h>
void main()
{
    char letter;
    printf("\nPlease enter any character : ");
    letter = getchar();
    printf("\nThe character entered by you is %c . ", letter);
}
```

A sample output is shown below:

```
Please enter any character: S
The character entered by you is S .
```

In the above program, **`letter`** is a variable declared to be of type `char` so as to accept character input. A message

```
Please enter any character:
```

will appear on the screen. You enter a character, say **s**, through the keyboard, and press carriage return. The function **`getchar()`** fetches the character entered by you and assigns the same to the variable letter. Later it is displayed on the screen with the message.

```
The character entered by you is S .
```

6.3.2 putchar()

`putchar()` is the character output function in C, which displays a character on the screen at the cursor position. This function requires an argument. The argument of the `putchar()` function can be any one of the following:

- A single character constant
- An escape sequence
- A character variable

If the argument is a constant, it must be enclosed in single quotes. Table 6.5 demonstrates some of the options available in `putchar()` and their effect.

Argument	Function	Effect
character variable	`putchar(c)`	Displays the contents of character variable c
character constant	`putchar('A')`	Displays the letter A
numeric constant	`putchar('5')`	Displays the digit 5
escape sequence	`putchar('\t')`	Inserts a tab space character at the cursor position
escape sequence	`putchar('\n')`	Inserts a carriage return at the cursor position

Table 6.5: putchar() options and their effects

The following program demonstrates the working of `putchar()`:

Example 13:

```
/* This program demonstrates the use of constants and escape
sequences in putchar() */
#include <stdio.h>
void main()
{
```

Concepts

```
    putchar('H'); putchar('\n');
    putchar('\t');
    putchar('E'); putchar('\n');
    putchar('\t'); putchar('\t');
    putchar('L'); putchar('\n');
    putchar('\t'); putchar('\t'); putchar('\t');
    putchar('L'); putchar('\n');
    putchar('\t'); putchar('\t'); putchar('\t');
    putchar('\t');
    putchar('O');
}
```

A sample output is shown below:

```
H
   E
      L
          L
             O
```

The difference between `getchar()` and `putchar()` is that `putchar()` requires an argument, while `getchar()` the earlier does not.

Example 14:

```
/* Program demonstrates getchar() and putchar() */
#include <stdio.h>
void main()
{
    char letter;
    printf("You can enter a character now: ");
    letter = getchar();
    putchar(letter);
}
```

A sample output is shown below:

```
You can enter a character now: F

F
```

Summary

- In C, I/O is performed using functions. Any program in C has access to three standard files. They are standard input file (called stdin), standard output file (called stdout) and the standard error (called stderr). Normally the standard input file is the keyboard, the standard output file is the screen and the standard error file is also the screen.
- The header file <stdio.h> contains the macros for many of the input / output functions used in C.
- Console I/O refers to operations that occur at the keyboard and the screen of your computer. It consists of formatted and unformatted functions.
- The formatted I/O functions are printf() and scanf().
- The unformatted functions are getchar() and putchar().
- The scanf() function is used to take the formatted input data, whereas the printf() function is used to print data in the specified format.
- The control string of printf() and scanf() must always be present inside " and ". The string consists of a set of format commands. Each format command consists of a %, an optional set of modifiers and a type specifier.
- The major difference between printf() and scanf() is that the scanf() function uses addresses of the variables rather than the variable names themselves.
- The getchar() function reads a character from the keyboard.
- The putchar(ch) function sends the character ch to the screen.
- The difference between getchar() and putchar() is that putchar() takes an argument while getchar() does not.

Concepts

Check Your Progress

1. The formatted I/O functions are _____ and _____.

   A. printf() and scanf() B. getchar() and putchar()

   C. puts() and gets() D. None of the above

2. scanf() uses _____ to variables rather than variable names.

   A. functions B. pointers

   C. arrays D. None of the above

3. _____ specify the form by which the values of the variables are to be input and printed.

   A. Text B. format specifier

   C. argument D. None of the above

4. ___ is used by the printf() function to identify conversion specifications.

   A. % B. &

   C. * D. None of the above

5. getchar() is a function without any arguments **[T/F]**

6. _____ is a temporary storage area in memory.

   A. ROM B. Register

   C. Buffer D. None of the above

7. Escape sequence can be placed outside the control string in printf(). **[T/F]**

Try It Yourself

1. A. Use the `printf()` statement and do the following :

   a. Print out the value of the integer variable **`sum`**

   b. Print out the text string "**`Welcome`**", followed by a new line.

   c. Print out the character variable **`letter`**

   d. Print out the `float` variable **`discount`**

   e. Print out the `float` variable **`dump`** using two decimal places

   B. Use the `scanf()` statement and do the following:

   a. To read a decimal value from the keyboard, into the integer variable **`sum`**

   b. To read a `float` variable into the variable **`discount_rate`**

2. Write a program which prints the ASCII values of the characters '**`A`**' and '**`b`**'.

3. Consider the following program:

```
#include <stdio.h>
void main()
{
    int breadth;
    float length, height;
    scanf("%d%f%6.2f",breadth,&length,height);
    printf("%d %f %e",&breadth, length, height);
}
```

Correct the errors in the above program.

Concepts

Try It Yourself

4. Write a program which takes **`name`**, **`basic`** , **`daper`** (ie, percentage of D.A), **`bonper`** (ie, percentage bonus) and **`loandet`** (loan amount to be debited) for an employee. Calculate the salary using the following relation:

```
salary = basic + basic * daper /100 +bonper * basic/100 - loandet
```

Data is:

name	basic	daper	bonper	loandet
M A R K	2500	55	33.33	250.00

Calculate salary and then print the result under the following headings.

(Salary to be printed to the nearest dollar.)

Name **Basic** **Salary**

5. Write a program that asks for your first name and last name, and then prints the names in the format last name, first name.

Session

7 Condition

Concepts

Objectives

At the end of this session, you will be able to:

- *Explain the Selection Construct*
  - The if Statement
  - The if – else statement
  - The Multi if statement
  - The Nested if statement
  - The switch Statement

Introduction

The programming elements discussed till now have made it possible to write most of the programs. However, one major problem attached to these programs is that when executed, they always perform the same series of actions, in the same way, exactly once. While programming, we need to perform a certain action only if certain criteria is met (selection) and also frequently.

7.1 What is a conditional statement?

Conditional statements enable us to change the flow of the program. Based on a condition, a statement or a sequence of statements take alternative actions.

Most of the programming tools use the `if` statement to make decisions. One of the most fundamental concepts of computer science is if certain condition is true, the computer is directed to take one course of action; and if the condition is false, it is directed to do something else.

To find whether a number is even or odd we proceed as follows:

1. Accept a number.
2. Find the remainder by dividing the number by 2.
3. If the remainder is zero , the number is “Even”.

Concepts

OR

If the remainder is not zero, then the number is “Odd”.

The second step checks if the number divided by two gives a zero remainder. In which case, we take a certain course of action that is displaying it as an even number. If it does not give a zero remainder, a different course of action is taken which displays the number is odd.

A point to be noted is that a conditional statement evaluates to either a true (non zero) or a false (zero) value in C.

7.2 Selection Statements

C supports two types of selection statements:

- The `if` statement
- The `switch` statement

Let us explore these two selection statements.

7.2.1 The ‘if’ Statement

The `if` statement allows decisions to be made by checking for a given condition to be true or false. Such conditions involve the comparison and logical operators discussed in the fourth session.

The general form of the `if` statement is:

```
if (expression)
    statements;
```

The expression should always be enclosed in parentheses. The statement after the keyword `if` is the condition (or expression) to be checked. This is followed by a statement or a set of statements, which are executed only if the condition (or expression) evaluates to `true`.

Example 1:

```
#include <stdio.h>
void main()
{
    int x, y;
    char a = 'y';
```

```
    x = y = 0;
    if (a == 'y')
    {
        x += 5;
        printf("The numbers are %d and \t%d", x, y);
    }
}
```

The above will produce an output as given below:

```
The numbers are 5 and     0
```

This is because the variable `a` has been assigned the value **y**.

Note that the block of statements following the `if` statement are enclosed in curly braces { }. Remember that if the constructs have more than one statement following it, the statements have to be treated like a block and have to be enclosed within curly braces. If the curly braces are not placed in the above case, only the first statement (x += 5) will be executed when the `if` statement evaluates to true.

Given below is an example which tests whether a given year is a leap year or not. A leap year is one which is divisible by 4 or 400 but not by 100. We use an `if` statement to test this condition.

Example 2:

```
/* To test for a leap year */
#include <stdio.h>
void main()
{
    int year;
    printf("\nPlease enter a year :");
    scanf("%d",&year);
    if(year % 4 == 0 && year % 100 != 0 || year % 400 == 0)
        printf("\n%d is a leap year", year);
}
```

A sample output is shown below:

```
Please enter a year : 1988
1988 is a leap year
```

The condition year % 4 == 0 && year % 100 != 0 || year % 400 == 0 results in a value of 1 if the given year is a leap year. In such a case, the year followed by the message "is a leap year" is printed on the screen.

Concepts

No message is printed if the condition does not hold good.

7.2.2 The 'if…else' statement

So far we have studied the simplest form of the `if` statement, which gives us a choice of executing a statement or a block of statements or skipping them. C also lets us choose between two statements by using the `if-else` structure. The formal syntax is:

```
if (expression)
    statement-1;
else
    statement-2;
```

The expression is evaluated; if it is true (non-zero), `statement-1` is executed. If it is false (zero) then `statement-2` is executed. The statements following `if` and `else` can be simple or compound. The indentation is not required but it is good programming style. It shows at a glance those statements whose execution depends on a test. (Indentation is the margin from the left side of the paper)

Let us now write a program which tests whether a number is even or odd. If the remainder, after dividing the number by 2, is zero it will display "The number is Even" otherwise, it will display "The number is Odd".

Example 3:

```
#include <stdio.h>
void main()
{
    int num , res ;
    printf("Enter a number :");
    scanf("%d",&num);
    res = num % 2;
    if (res == 0)
        printf("The number is Even");
    else
        printf("The number is Odd");
}
```

Consider another example which converts an uppercase character to lowercase character. If the character is not an uppercase letter, it is printed without change. The program uses the `if-else` construct to test whether a letter is uppercase and then to choose between the options available.

Example 4

```
/* To convert an uppercase character to lowercase */
#include <stdio.h>
void main()
{
    char c;
    printf("Please enter a character : ");
    scanf("%c",&c);
    if (c >= 'A' && c <= 'Z')
        printf("Lowercase character = %c", c + 'a' - 'A');
    else
        printf("Character Entered is = %c",c);
}
```

The expression `c >= 'A' && c <='Z'` tests if a letter is uppercase. If the expression evaluates to `true`, the input character is converted to lowercase using the expression `c + 'a' - 'A'`, and output using the `printf()` function. If the value of the expression is `false`, the statement following `else` is executed and the character is output without any change.

7.2.3 Multiple Choice – 'else–if ' statements

The `if` statement lets us choose whether or not to do some action. The `if-else` statement lets us choose between two actions. C also offers us more than two choices. We can extend the `if-else` structure with `else-if` to accommodate this fact. That is, the `else` part of an `if-else` statement consists of another `if-else` statement. This enables multiple conditions to be tested and hence offers several choices.

The general syntax for this is:

```
if (expression) statement;
else
if (expression) statement;
else
if (expression) statement;
.
.
else statement;
```

This construct is also known as the **if-else-if ladder** or **if-else-if staircase**.

Concepts

The above indentation is easily understandable with one or two `if`s. It can get confusing when the number of `if`s increases because in that case it gets deeply indented. So, the `if-else-if` statement is generally indented as:

```
if (expression)
    statement;
else if (expression)
    statement;
else if (expression)
    statement;
    .
    .
else
    statement;
```

The conditions are evaluated from top to down. As soon as a `true` condition is found, the statement associated with it is executed and the rest of the ladder is bypassed. If none of the conditions are true, then the final `else` is executed. If the final `else` is not present, no action takes place if all other conditions are false.

The following example takes a number from the user. If the number is between 1 and 3 it prints the number, or else it prints “Invalid choice”.

Example 5:

```
#include <stdio.h>
main()
{
    int x;
    x = 0;
    clrscr ();
    printf(“Enter Choice (1 - 3) : “);
    scanf(“%d”, &x);
    if (x == 1)
        printf (“\nChoice is 1”);
    else if (x == 2)
        printf (“\nChoice is 2”);
    else if (x == 3)
        printf (“\nChoice is 3”);
    else
        printf (“\nInvalid Choice”);
}
```

In the above program,

If x = 1, "**Choice is 1**" is displayed.

If x = 2, "**Choice is 2**" is displayed.

If x = 3, "**Choice is 3**" is displayed.

If x is any number other than 1, 2 or 3, "**Invalid choice**" is displayed.

If we wish to have more than one statement following the `if` or the `else` statement, they should be grouped together between curly brackets { }. Such a grouping is called a 'compound statement' or 'a block'.

```
if (result >= 45) {
    printf("Passed\n");
    printf("Congratulations\n");
}
else {
    printf("Failed\n");
    printf("Good luck next time\n");
}
```

7.2.4 Nested if

A nested `if` is an `if` statement, which is placed within another `if` or `else` statement. In C, an `else` statement always refers to the nearest `if` statement that is within the same block as the `else` statement and is not already associated with an `if` statement. For example,

```
if (expression-l)
{
    if (expression-2)
        statement1;
    if (expression-3)
        statement2;
    else
        statement3; /* with if(expression-3) */
}
else
    statement4; /* with if(expression-l) */
```

Concepts

In the above segment, control comes to the second `if` statement if the value of `expression-1` is `true`. If `expression-2` is `true` then `statement1` is executed. The `statement2` is executed when `expression-3` is `true`, otherwise `statement3` is executed. If `expression-1` turns out to be `false` then `statement4` is executed.

Since the `else` part of the `else-if` is optional, it is possible to have constructs similar to the one given below:

```
if (condition-l)
    if (condition-2)
       statement-1;
    else
        statement-2;
next statement;
```

In the above segment of code, if `condition-1` turns out to be `true`, then the control is transferred to the second `if` statement and `condition-2` is evaluated. If it is `true`, then `statement-1` is executed, otherwise `statement-2` is executed, followed by the execution of the `next statement`. If `condition-1` is `false`, then control passes directly to `next statement`.

Consider an example where marks1 and marks2 are the scores obtained by a student in two subjects. Grace marks of 5 are added to marks2 if marks1 is greater than 50 but marks2 is less than 50. If marks2 is greater than or equal to 50 then the grade obtained by the student is 'A'. This condition can be coded using the `if` construct as follows:

```
if (marks1 > 50 && marks2 <50)
    marks2 = marks2 + 5;
if (marks2 >=50)
    grade = 'A';
```

Some users may be tempted to code as follows:

```
if (marks1 > 50 )
if (marks2 <50)
    marks2 = marks2 + 5;
else
    grade = 'A';
```

In this segment, **'A'** will be assigned to grade only when marks1 is greater than 50 and marks2 is greater than or equal to 50. But, according to our requirement, the grade should be assigned the value **'A'** after testing and adding grace marks to marks2. Also the grade is independent of the value in marks1.

Because the `else` part of an `if` is optional, it is not clear when an `else` is omitted from a nested `if`

sequence. This is resolved by using the rule that the `else` is associated with the closest previous if not associated with an `else`.

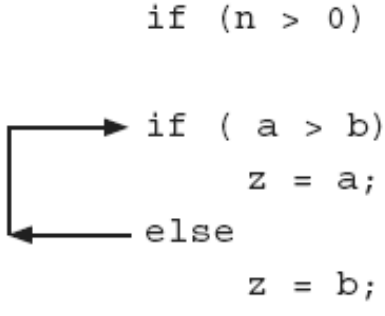
For example, in

```
if (n > 0)
    if (a > b)
        z = a;
    else
        z = b;
```

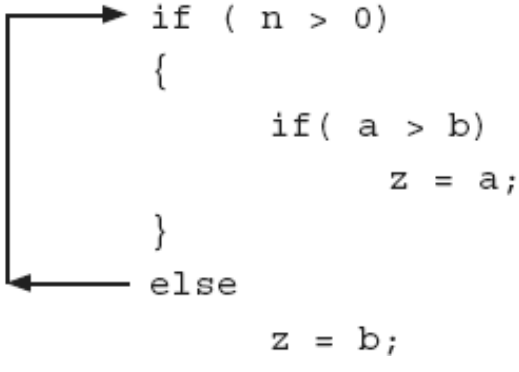
the `else` goes with the inner `if`. The indentation is a way of showing this relationship. However, indentation does not associate an `else` with an `if`. If that is not what you want curly braces { } must be used to force the proper association.

```
if (n > 0)
{
    if (a > b)
        z = a;
}
else
    z = b;
```

The following figure shows the association between `if` and `else` in a sequence of nested `if` statements.



`else` goes with the most recent `if`



`else` goes with the first `if` since braces enclose the inner `if` statement

According to the ANSI standard, at least 15 levels of nesting must be supported. However, most compilers allow for much more than that.

Concepts

As an example of nested `if`, consider the following:

Example 6:

```
#include <stdio.h>
void main ()
{
    int x, y;
    x = y = 0;
    clrscr ();
    printf (“Enter Choice (1 - 3) :” );
    scanf (“%d”, &x);
    if (x == 1)
    {
        printf(“\nEnter value for y (1 - 5) :” );
        scanf (“%d”, &y);
        if (y <= 5)
            printf(“\nThe value for y is : %d”, y);
        else
            printf(“\nThe value of y exceeds 5” );
    }
    else
        printf (“\nChoice entered was not 1”);
}
```

In the above program, if the value for `x` is entered as 1, the user is asked to enter the value for `y` or else “Choice entered was not 1” is displayed. The first `if` has a nested `if` which displays the value of `y` if the user has entered a value less than 5 for `y`, or else it displays “The value of y exceeds 5”.

A complete program below shows the usefulness of nested `if`.

A company manufactures three products namely computer stationery, fixed disks and computers. The following codes are used to indicate them.

Product	Code
Computer Stationery	1
Fixed Disks	2
Computers	3

The company has a discount policy as follows:

Product	Order Amount	Discount Rate
Computer Stationery	$ 500/- or more	12 %
Computer Stationery	$ 300/- or more	8 %
Computer Stationery	below $300 /-	2 %
Fixed Disks	$ 2000/- or more	10 %
Fixed Disks	$ 1500/- or more	5 %
Computers	$ 5000/- or more	10 %
Computers	$ 2500/- or more	5 %

The program to calculate the discount amounts as per this policy is given below:

Example 7:

```
#include <stdio.h>
void main()
{
    int productcode;
    float orderamount, rate = 0.0 ;
    printf("\n Please enter the product code :" );
    scanf("%d", &productcode);
    printf("Please enter the order amount :" );
    scanf(" %f" , &orderamount);
    if (productcode == 1)
    {
        if (orderamount >= 500 )
            rate = 0.12;
        else if (orderamount >= 300 )
            rate = 0.08;
        else
            rate = 0.02;
    }
    else if ( productcode == 2)
    {
        if (orderamount >= 2000)
            rate = 0.10 ;
        else if (orderamount >= 1500)
            rate = 0.05;
    }
```

Concepts

```
    else if ( productcode == 3)
    {
        if (orderamount >= 5000)
            rate = 0.10 ;
        else if (orderamount >= 2500)
            rate = 0.05;
    }
    orderamount - = orderamount * rate;
    printf( "The net order amount is % .2f \n", orderamount);
}
```

A sample output is shown below:

```
Please enter the product code : 3
Please enter the order amount : 6000
The net order amount is 5400
```

Observe that the last `else` in a sequence of `else-if`'s need not check any condition specifically. For example, if the product code is 1 and the order amount is less than $ 300, there was no need to check, since all the possibilities were already taken care of. The output of the above program for a product code of 3 and order amount of $ 6000 is as shown above:

Modify the above program to take care of a condition where the input consists of an invalid product code. This can be easily achieved by adding an `else` to the sequence of if's which are testing the product code. If such a product code is encountered, the program should terminate without calculating the net order amount.

7.2.5 The 'switch' Statement

The `switch` statement is a multi-way decision maker that tests the value of an expression against a list of integer or character constants. When a match is found, the statements associated with that constant are executed. The general syntax of the `switch` statement is:

```
switch (expression)
{
    case constant1:
        statement sequence
        break;
    case constant2:
        statement sequence
        break;
```

```
    case constant3:
        statement sequence
        break;
    default:
        statement sequence;
}
```

where, `switch`, `case` and `default` are the keywords and a statement sequence can be simple statement or a compound statement, which need not be enclosed in parentheses. The expression following `switch` must be enclosed in parentheses, and the body of the `switch` must be enclosed within curly braces { }. The data type of the expression to be evaluated and the data type of the case constants specified should be matching. As suggested by the name, case labels can be only an integer or a character constant. It can also be constant expressions provided the expressions do not contain any variable names. All the case labels must be different.

In the `switch` statement, the expression is evaluated, and the value is compared with the case labels in the given order. If a label matches with the value of the expression, the statement mentioned against that case label will be executed. The `break` statement (discussed later) ensures immediate exit from the `switch` statement. If a `break` is not used, the statements in the following case labels are also executed irrespective of whether that case value is satisfied or not. This execution will continue till a `break` is encountered. Therefore, `break` is said to be one of the most important statement while using the `switch` statement.

The statements written against default will be executed, if none of the other cases are satisfied. The default statement is optional. If it is not present, and the value of the expression does not match with any of the cases, then no action will be taken The case labels and default may be in any order .

Consider the following example.

Example 8:

```
#include <stdio.h>
main ()
{
    char ch;
    clrscr();
    printf ("\nEnter a lower cased alphabet (a - z) :" );
    scanf("%c", &ch);
    if(ch < 'a' || ch > 'z')
        printf("\nCharacter not a lower cased alphabet");
    else
        switch(ch)
```

```
        {
            case 'a' :
            case 'e' :
            case 'i' :
            case 'o' :
            case 'u' :
                printf("\nCharacter is a vowel");
                break;
            case 'z' :
                printf ("\nLast Alphabet (z) was entered");
                break;
            default :
                printf("\nCharacter is a consonant");
                break;
        }
}
```

The program will take a lower cased alphabet as an input and display whether it is a vowel, the last alphabet or any of the other alphabets. If any other key, other than lower cased alphabets, is entered, the message "Character not a lower cased alphabet" is displayed.

It is recommended that a `break` be put even after the last case or default even though it is logically unnecessary. This is helpful when another case gets added at the end.

Here's another example where the expression in `switch` is an integer variable and each of the case labels is an integer.

Example 9:

```
/* Integer constants as case labels */
#include <stdio.h>
void main()
{
    int basic;
    printf("\n Please enter your basic:" );
    scanf("%d",&basic);
    switch (basic)
    {
        case 200 : printf("\n Bonus is dollar %d\n", 50);
        break;
```

```
        case 300 : printf("\n Bonus is dollar %d\n", 125);
            break;
        case 400 : printf("\n Bonus is dollar %d\n", 140);
            break;
        case 500 : printf("\n Bonus is dollar %d\n", 175);
            break;
        default : printf("\n Invalid entry");
            break;
    }
}
```

From the above example, it is clear that the `switch` statement is useful if the expression matches one of the **case** labels. But it cannot be used to test whether a value lies within a specified range. For example, it is not possible to test if **basic** lies between 200 and 300, and then to determine the Bonus . In such cases, we have to once again use a sequence of `if-else` statements.

Summary

- Conditional statements enable us to change the flow of the program.
- C supports two types of selection statements: if and switch.
- The following are some of the conditional statements.
  - The if statement – A condition is checked; if the results turn out to be true, the statements following it are executed and joins the main program. If the results are false, it joins the main program straight.
  - The if…else statement – A condition is checked, if the results are true, the statements following the if statement are executed. If the results are false, then the statements following the else part are executed.
  - Nested if statements – Nested if statements comprise of an if within an if statement.
  - The switch statement is a special multi-way decision maker that tests whether an expression matches one of a number of constant values, and branches accordingly.

Check Your Progress

1. _____ statements enable us to change the flow of a program.

   A. Conditional B. Loop

   C. Sequence D. None of the above

2. The `else` statement is optional. **(T/F)**

3. A _____ is an `if` statement, which is placed within another `if` or `else`.

   A. Multi `if` B. Nested `if`

   C. `switch`ed `if` D. None of the above

4. The _____ statement is a multi-way decision maker that tests the value of an expression against a list of integer or character constants.

   A. Sequence B. `if`

   C. `switch` D. None of the above

5.
```
if (expression)
        statement 1
else
        statement 2
```

Which statement will be executed when expression is `false`?

A. `statement 1` B. `statement 2`

Try It Yourself

1. Write a program that accepts two numbers **a** and **b** and checks whether or not **a** is divisible by **b**.

2. Write a program to accept 2 numbers and tell whether the product of the two numbers is equal to or greater than 1000.

3. Write a program to accept 2 numbers. Calculate the difference between the two values.

   If the difference is equal to any of the values entered, then display the following message:

   Difference is equal to value <number of value entered>

   If the difference is not equal to any of the values entered, display the following message:

   Difference is not equal to any of the values entered

4. Montek company gives allowances to its employees depending on their grade as follows:

Grade	Allowance
A	300
B	250
Others	100

   Calculate the salary at the end of the month. (Accept Salary and Grade from the user)

5. Write a program to evaluate the Grade of a student for the following constraints:

   If marks > 75 – grade A

   If 60 < marks < 75 – grade B

   If 45 < marks<60 – grade C

   If 35 < marks<45 - grade D

   If marks < 35 – grade E

Session

8 Condition (Lab)

Objectives

At the end of this session, you will be able to:

- *Use :*
  - *The if statement*
  - *The if – else statement*
  - *The Multi if statement*
  - *The Nested if statement*
  - *The switch statement*

The steps given in this session are carefully thought and explained in great detail so that the understanding of the tool is complete. Follow the steps carefully.

Part I - For the first 1 Hour and 30 minutes:

8.1 The 'if' statement

In this section you will write a program to calculate the commission given to salesmen depending on their sales amount.

Problem

SARA Company gives a `10%` commission to its salesmen if their monthly sales amount to `$10,000` or more. Calculate the commission at the end of the month.

The program declares two '`float`' variables **`sales_amt`** and **`com`**. Note that the variables are declared in the same line of code using a comma (`,`) to separate one from the other.

Consider the following line of code :

```
printf("Enter the Sales Amount : ");
scanf("%f", &sales_amt);
```

In the `printf()` function, we display a message to enter the sales amount, and in the `scanf()` function

we use `%f` to accept a value from the user. The value you are entering goes to the variable **`sales_amt`**.

```
if (sales_amt >= 10000)
    com = sales_amt * 0.1;
```

The above statement is used to check whether the value in **`sales_amt`** variable is greater than or equal to **`10000`** . `>=` is a comparison operator which gives you a **`true`** or **`false`** value. In this case, if you are entering a value **`15000`** the condition (`sales_amt >= 10000`) is **`true`**. If **`true`**, it will execute the statement `com = sales_amt * 0.1` . Now the value of **`com`** will be **`1500`**. If the condition is **`false`**, it will print the commission value as **`0`**. Here we can see that `if` condition has only one statement. If there is more than one statement for an `if` condition, give the statements in curly braces `{ }`.

```
printf("\n Commission = %f",com);
```

The above statement is used to display the value of commission. '`%f`' is used to display the value of the '`float`' variables mentioned after the comma at the end of `printf()`. Thus, `printf()` displays the commission amount of simple interest.

8.1.1 Calculating the commission

1. **Create a new file.**

2. **Type the following code in the 'Edit Window' :**

```
#include <stdio.h>
#include <conio.h>
void main()
{
    float com=0,sales_amt;
    clrscr();
    printf("Enter the Sales Amount : ");
    scanf("%f",&sales_amt);
    if (sales_amt >= 10000)
        com = sales_amt * 0.1;
    printf("\n Commission = %f",com);
}
```

3. **Save the file with the name `comm.c`**

4. **Compile the file `comm.c`**

5. **Execute the program `comm.c`**

6. **Return to the editor.**

OUTPUT :

```
Enter the Sales Amount : 15000
Commision = 1500.0000
```

8.2 The 'if- else' statement

In this section you will write a program that makes use of the `if-else` statement. The program displays the greater of the two given numbers.

Consider the following lines of code :

```
if(num1 > num2)
    printf("\n The greater number is : %d", num1);
else
    printf("\n The greater number is : %d", num2);
```

In this program the first `printf()` is executed only if the value of the variable **`num1`** is greater than the value of the variable **`num2`**, in which case the `else` part is ignored. If the value of the variable **`num1`** is not greater than the value of the variable **`num2`**, the first `printf()` is ignored. In this case the second `printf()`, the one following the `else`, is executed.

In this program, since the value of **`num1`** is greater than **`num2`**, the first `printf()` is executed.

1. **Create a new file.**

2. **Type the following code in the 'Edit Window':**

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int num1,num2,;
    clrscr();
    num1 = 540;
    num2 = 243;
```

```
    if (num1 > num2)
        printf( "\n The greater number is : %d", num1);
    else
        printf("\n The greater number is : %d", num2);
}
```

3. **Save the file with the name `ifelse.c`**

4. **Compile the file `ifelse.c`**

5. **Execute the program `ifelse.c`**

6. **Return to the editor.**

OUTPUT :

```
The greater number is : 540
```

8.3 The 'if – else – if' statement

In this section you will write a program that makes use of the `if – else – if` statement. The program displays the greater of the two given numbers or displays that the numbers are equal.

In the previous program, there are two 'integer' variables **`num1`** and **`num2`** declared. The variables are assigned a value.

Consider the following lines of code :

```
if(num1 == num2)
    printf("\n Numbers are Equal");
else if(num1 < num2)
    printf("\n The Larger Number is : %d", num2);
else
    printf("\n The Larger Number is : %d", num1);
```

In this program, the first '`if`' condition `(num1==num2)` checks if the value of the variable **`num2`** is equal to the value of the variable **`num1`**. In C, the `==` symbol is used to check if the two operands are equal. If the first condition `(num1==num2)` is `true` then, the following `printf()` will be executed and the control will come out of the loop. If the first condition is not true, the `else if` condition will be checked. In case this condition `(num1<num2)` is satisfied, the associated `printf()` will be executed and the control will come out of the loop. If neither the `if` nor the `else if` conditions are satisfied, then the `else` condition

will be executed.

In this program, since the value of **num1** is lesser than **num2**, the second `printf()` statement is executed.

1. **Create a new file.**

2. **Type the following code in the 'Edit Window' :**

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int num1 , num2 ;
    num1 = 77;
    num2 = 90;
    if( num1 == num2)
        printf("\n The Numbers are equal");
    else if (num1 < num2)
        printf("\n The Larger Number is : %d", num2);
    else
        printf("\n The Larger Number is : %d", num1);
}
```

3. **Save the file with the name `ifelseif.c`**

4. **Compile the file `ifelseif.c`**

5. **Execute the program `ifelseif.c`**

6. **Return to the editor.**

OUTPUT :

```
The Larger Number is: 90
```

Lab Guide

8.4 The 'Nested if' statement

In this section you will write a program for understanding the 'Nested if' statement.

Problem:

MONTEK company has decided to give commission to their sales department depending on the sales amount. The commission structure is defined as follows:

Sales	Grade	Commission
> 10,000 $	A	10 %
	—	8%
<=10,000 $	—	5 %

Calculate the commission at the end of each month.

In this program we are calculating the commission based on grade and sales amount.

Consider the following lines of code:

```
printf("Enter the Sales Amount : ");
scanf("%f",&sales_amt);
printf("\n Enter the Grade : ");
scanf("%c",&grade);
```

The first `scanf()` is used to accept the sales amount, and second `scanf()` is used to accept the grade. `%c` format specifier is used to accept a single character from the user.

```
if (sales_amt > 10000)
    if (grade == 'A')
        com = sales_amt * 0.1;
    else
        com = sales_amt * 0.08;
else
    com = sales_amt * 0.05;
```

Suppose we enter the sales amount as **`15000`** and grade as '**`A`**' . It will check the `if` condition (`sales_amt > 10000);` since this condition is **`true`** it will go to the second `if (grade == 'A')`. This condition is also **`true`**, so it will calculate commission `com = sales_amt * 0.1`.

We will see another situation with sales amount **`15000`** and grade as '**`B`**'. It will check the first `if` condition `(sales_amt > 10000)`, in this case it is **`true`**. Then it will go to the second `if` statement, in this case

it is not true, it will go to the respective `else` condition i.e.,

```
else
    com = sales_amt * 0.08;
```

If we enter a value **10000** or less, it will go the last `else` condition and calculate the commission as

`com = sales_amt * 0.05;`

1. **Create a new file.**

2. **Type the following code in the 'Edit Window' :**

```
#include <stdio.h>
#include <conio.h>
void main()
{
    float com=0,sales_amt;
    char grade;
    clrscr();
    printf("Enter the Sales Amount : ");
    scanf("%f", &sales_amt);
    printf("\n Enter the Grade : ");
    scanf("%c", &grade);
    if (sales_amt > 10000)
        if (grade == 'A')
            com = sales_amt * 0.1;
        else
            com = sales_amt * 0.08;
    else
        com = sales_amt * 0.05;
    printf("\n Commission = %f", com);
}
```

3. **Save the file with the name `nestif.c`**

4. **Compile the file `nestif.c`**

5. **Execute the program `nestif.c`**

6. **Return to the editor.**

Lab Guide

OUTPUT :

```
Enter the Sales Amount : 15000
Enter the grade : A
Commission = 1500
```

8.5 Using the 'switch' statement

In this section you will use the 'switch' statement. The program displays appropriate result depending on the mathematical operator used.

In this program there are two integer variables **num1** and **num2** and a character variable **op** declared. Values are assigned to the variables. A mathematical operator is stored in the variable **op**.

The variable **op** is passed in the expression following the 'switch'. The first case compares the value of the variable **op** to be '+'. If the label (+) matches with the value in **op**, then the following lines of code are executed:

```
res = num1 + num2;
printf("\n The sum is : %d", res);
break;
```

The sum of **num1** and **num2** is stored in the variable **res**. The printf() displays the value of the variable **res**. The 'break' statement makes an exit from the 'switch' statement.

In the second case the value of **op** is '–' , then num1 – num2 is executed. Similarly if the value of **op** is '*' and '/' , num1 *num2 and num1 / num2 is executed.

If none of the above cases are satisfied, then the printf() of the 'default' is executed.

1. Create a new file.

2. Type the following code in the 'Edit Window' :

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int num1, num2 ,res;
    char op;
    num1 = 90;
    num2 = 33;
    op = '-';
```

```
    clrscr();
    switch(op)
    {
        case '+':
            res = num1 + num2;
            printf("\n The Sum is : %d", res);
            break;
        case '-':
            res = num1 - num2;
            printf("\n Number after Subtraction : %d", res);
            break;
        case '/':
            res = num1 / num2;
            printf("\n Number after Division : %d", res);
            break;
        case '*':
            res = num1 * num2;
            printf("\n Number after multiplication : %d", res);
            break;
        default:
            printf("\n Invalid");
        break;
    }
}
```

3. Save the file with the name `case.c`

4. Compile the file `case.c`

5. Execute the program `case.c`

6. Return to the editor.

OUTPUT :

```
Number after subtraction: 57
```

Part II : For the next 30 Minutes:

1. A student appears for a test in 3 subjects. Each test is out of 100 marks. The percentage of each student has to be calculated and depending on the percentage calculated, grades are given as under:

Percentage	Grade
>=90	E +
80 - < 90	E
70 - < 80	A +
60 - < 70	A
50 - < 60	B +
< 50	FAIL

To do this :

a. Accept the marks of 3 subjects and store in 3 different variables say **`M1`**, **`M2`** and **`M3`**.

b. Calculate the percentage (`per = (M1 + M2 + M3) / 3`)

c. Check the grade depending on the percentage calculated.

d. Display the grade.

Try It Yourself

1. Declare two variables `x` and `y`. Assign values to these variables. Number `x` should be printed only if it is less than `2000` or greater than `3000`, and number `y` should be printed only if it is between `100` and `500`.

2. Write a program to show your computer's capabilities. The user types in a letter of the alphabet and your program should display the corresponding language or package available. Some sample input and output is given below :

Input	Output
A or a	Ada
B or b	Basic
C or c	COBOL
D or d	dBASE III
f or F	Fortran
p or P	Pascal
v or V	Visual C++

Using the '`switch`' statement to choose and display the appropriate message. Use the `default` label to display a message if the input does not match any of the above letters.

3. Accept values in three variables and print the highest value.

“It is always in season for old men to learn”

Session

9 Loop

Concepts

Objectives

At the end of this session, you will be able to:

- *Understand the 'for' loop in C*
- *Work with the 'comma' operator*
- *Understand nested loops*
- *Understand the 'while' loop and the 'do-while' loop*
- *Work with 'break' and 'continue' statements*
- *Understand the 'exit()' function*

Introduction

One of the greatest advantages of a computer is its capability to execute a series of instructions repeatedly. This is achieved by the loop structures available in a programming language. In this session you will learn the various loop structures in C.

9.1 The Loop Structure

A loop is a section of code in a program which is executed repeatedly, until a specific condition is satisfied. The loop concept is fundamental to structured programming.

The loop structures available in C are:

- The `for` loop
- The `while` loop
- The `do…while` loop

The condition which controls the execution of the loop is coded using the **Relational** and **Logical** operators in C.

Concepts

9.1.1 The 'for' Loop

The general syntax of the `for` loop is:

```
for(initialize counter; conditional test; re-evaluation parameter)
{
    Statement(s);
}
```

The **initialize counter** is an assignment statement that sets the loop control variable, before entering the loop. This statement is executed only once. The **conditional test** is a relational expression, which determines when the loop will exit. The **re-evaluation parameter** defines how the loop control variable changes (mostly, an increment or decrement in the variable set at the start) each time the loop is repeated. These three sections of the `for` loop must be separated by semicolons. The statement, which forms the body of the loop, can either be a single statement or a compound statement (more than one statement).

The `for` loop continues to execute as long as the conditional test turns out to be **`true`**. When the condition becomes **`false`**, the program resumes with the statement following the `for` loop.

Consider the following program:

Example 1:

```
/*This program demonstrates the for loop in a C program */
#include <stdio.h>
main()
{
    int count;
    printf("\tThis is a \n");
    for(count = 1; count <=6 ; count++)
        printf("\n\t\t nice");
    printf("\n\t\t world. \n");
}
```

The sample output is shown below :

```
This is a
    nice
    nice
    nice
    nice
```

```
    nice
    nice
    world.
```

Look at the `for` loop section in the program.

1. The initialization parameter is `count = 1`

   It is executed only once when the loop is entered, and the variable **`count`** is set to **`1`**.

2. The conditional test is `count <=6`

   A test is made to determine whether the current value of **`count`** is less than or equal to **`6`**. If the test is **`true`**, then the body of the loop is executed.

3. The body of the loop consists of a single statement

   ```
   printf("\n\n\t nice");
   ```

   This statement can be enclosed in braces to make the body of the loop more visible.

4. The re-evaluation parameter is **`count++`** , increments the value of **`count`** by **`1`** for the next iteration.

The steps 2,3,4 are repeated until the conditional test becomes **`false`**. The loop will be executed 6 times for value of **`count`** ranging from **`1`** to **`6`**. Hence, the word **`nice`** appears six times on the screen. For the next iteration, **`count`** is incremented to **`7`**. Since this value is greater than **`6`**, the loop is ended and the statement that follows it is executed.

The following program prints even numbers from 1 to 25.

Example 2:

```
#include <stdio.h>
main()
{
    int num;
    printf("The even numbers from 1 to 25 are:\n\n");
    for(num=2; num <= 25; num+=2)
        printf ("%d\n",num);
}
```

The output for the above code will be:

```
The even numbers from 1 to 25 are:
2
4
6
8
10
12
14
16
18
18
20
22
24
```

The `for` loop above initializes the integer variable **`num`** to **`2`** (to get an even number) and increments it by **`2`** every time the loop is executed.

In `for` loops, the conditional test is always performed at the top of the loop. This means that the code inside the loop is not executed if the condition is **`false`** in the beginning itself.

The 'Comma' operator

The scope of the `for` loop can be extended by including more than one initializations or increment expressions in the `for` loop specification. The format is :

```
exprn1 , exprn2
```

The expressions are separated by the 'comma' operator and evaluated from left to right. The order of the evaluation is important if the value of the second expression depends on the newly calculated value of **`exprn1`**. This operator has the lowest precedence among the C operators.

The following example, which prints an addition table for a constant result, will illustrate the concept of the comma operator clearly:

Example 3:

```
/* program illustrates the use of the comma operator */
#include <stdio.h>
```

```
main()
{
    int i, j , max;
    printf("Please enter the maximum value \n");
    printf("for which a table can be printed: ");
    scanf("%d", &max);
    for(i = 0 , j = max ; i <=max ; i++, j--)
        printf("\n%d + %d = %d",i, j, i + j);
}
```

A sample output is shown below :

```
Please enter the maximum value -
for which a table can be printed: 5
0 + 5 = 5
1 + 4 = 5
2 + 3 = 5
3 + 2 = 5
4 + 1 = 5
5 + 0 = 5
```

Note that in the `for` loop, the initialization parameter is

```
i = 0, j = max
```

When it is executed , **`i`** is assigned the value **`0`** and **`j`** is assigned the value present in **`max`**.

The re-evaluation (increment) parameter again consists of two expressions:

```
i++, j--
```

after each iteration, **`i`** is incremented by **`1`** and **`j`** is decremented by **`1`**. The sum of these two variables which is always equal to **`max`** is printed out.

- **'Nested for' Loops**

  A `for` loop is said to be nested when it occurs within another `for` loop. The code will be something like this:

Concepts

```
for(i = 1; i < max1; i++)
{
    .
    .
    for(j = 0; j < = max2; j++)
    {
        .
        .
    }
    .
    .
}
```

Consider the following example,

Example 4:

```
#include <stdio.h>
main()
{
    int i, j, k;
    i = 0;
    printf(”Enter no. of rows :”);
    scanf(”%d”, &i);
    printf(“\n”);
    for(j = 0; j < i ; j++)
    {
        printf(“\n”);
        for (k = 0; k <= j; k++) /*inner for loop */
           printf(“*”);
    }
}
```

This program displays ‘*’ on each line and for each line increments the number of ‘*’ to be printed by `1`. It takes the number of rows for which ‘*’ has to be displayed as input. For example, if the input is `5`, the output will be

```
*
**
***
****
*****
```

➢ **More on 'for' loops**

The `for` loop can be used without one or all of its definitions.

For example,

```
.
.
for(num = 0; num ! = 255;)
{
    printf("Enter no. ");
    scanf("%d", &num);
.
.
}
```

The above will accept a value for **num** until the input is **255**. This loop does not have any re-evaluation factor. The loop terminates when **num** becomes **255**.

Similarly, consider

```
.
.
printf("Enter value for checking :");
scanf("%d", &num);
for(; num < 100; )
{
    .
    .
}
```

This loop does not have an initialization factor or a re-evaluation factor.

The `for` loop, when used without any definitions, gives an infinite loop.

Concepts

```
for( ; ; )
    printf("This loop will go on and on and on ...\n");
```

However, a `break` statement used within this loop will cause an exit from it.

```
.
.
for( ; ; )
{
    printf("This will go on and on");
    i = getchar();
    if(i == 'X' || i == 'x')
        break;
}
.
.
```

The above loop will run until the user enters **`x`** or **`X`**.

The `for` loop (or any other loop) can also be used without the body (statements). This helps to increase the efficiency of some algorithms and to create time delay loops.

```
for(i = 0; i < xyz_value, i++);
```

is an example of a time delay loop.

9.1.2 The 'while' Loop

The second kind of loop structure in C is the `while` loop. Its general syntax is:

```
while(condition is true)
    statement;
```

where, **`statement`** is either an empty statement, a single statement or a block of statements. If a set of statements are present in a `while` loop then they must be enclosed inside curly braces `{ }`. The condition may be any expression. The loop iterates while this condition is true. The program control is passed to the line after the loop code, when the condition becomes false.

The `for` loop can be used provided the number of iterations are known before the loop starts executing, When the number of iterations to be performed is not known before hand, the `while` loop can be used.

Example 5:

```
/* A simple program using the while loop */
#include <stdio.h>
main()
{
    int count = 1;
    while( count <= 10)
    {
        printf("\n This is iteration %d\n",count);
        count++;
    }
    printf("\n The loop is completed. \n");
}
```

A sample output is shown below:

```
This is iteration 1
This is iteration 2
This is iteration 3
This is iteration 4
This is iteration 5
This is iteration 6
This is iteration 7
This is iteration 8
This is iteration 9
This is iteration 10
The loop is completed.
```

The program initially sets the value of **`count`** to **`1`** in the declaration statement itself. The next statement executed is the `while` statement. The condition is first checked. The current value of **`count`** is **`1`**, which is less than **`10`**. The condition test results is **`true`**, and therefore the statements in the body of the `while` loop are executed. Therefore, they are enclosed within curly braces `{ }`. The value of **`count`** becomes **`2`** after 1st iteration. Then the condition is checked again. This process is repeated until the value of **`count`** becomes greater than **`10`**. When the loop is exited, the second `printf()` statement is executed.

Like `for` loops, `while` loops check the condition at the top of the loop. This means that the loop code is not executed, if the condition is **`false`** at the start.

The conditional test in the loop may be as complex as required. The variables in the conditional test may be reassigned values within the loop, but a point to remember is that eventually the condition test must

become false otherwise the loop will never end. The following is an example of an infinite `while` loop.

Example 6:

```
#include <stdio.h>
main ()
{
    int count = 0;
    while(count < 100)
    {
        printf("This goes on forever, HELP!!!\n");
        count+=10;
        printf("\t%d", count);
        count-=10;
        printf("\t%d", count);
        printf("\nCtrl-C will help");
    }
}
```

In the above, **`count`** is always **`0`**, which is less than **`100`** and so the expression always returns a **`true`** value. Hence the loop never ends.

If more than one condition is to be checked to end a `while` loop, the loop will end if at least one of the conditions become false. The following example illustrates this.

Example 7:

```
#include <stdio.h>
main()
{
    int i, j;
    i = 0;
    a = 10;
    while(i < 100 && a > 5)
    {
        .
        .
        i++;
        a-= 2;
    }
```

```
    .
    .
}
```

This loop will perform **3** iterations, the first time **a** will be **10**, the next time it will be **8** and the third time it will be **6**. After this though **i** is still less than **100** (**i** is **3**), **a** takes the value **4**, and the condition a>5 turns out to be false, so the loop ends.

Let us write a useful program. It accepts input from the console and prints it on the screen.

The program ends when you press **^Z (Ctrl+Z)**.

Example 8:

```
/* ECHO PROGRAM */
/* A program to accept input data from the console and print it on the
screen */
/* End of input data is indicated by pressing '^Z' */
# include <stdio.h>
main()
{
    char ch;
    while((ch = getchar()) != EOF)
    {
        putchar(ch);
    }
}
```

A sample output is shown below :

```
Have
Have
a
a
good
good
day
day
^ z
```

Concepts

The user input is highlighted. How does this program work? After entering a set of characters, the contents will get echoed back to the screen only when you press **<Return>**. This is because the characters you enter through the keyboard gets stored in the keyboard buffered input and the `putchar()` statement fetches it from the buffer when you press **<Return>**. Notice how the input is terminated with a **^Z**, which is the end of file character on MS DOS operating system.

9.1.3 The 'do .... while' Loop

The `do...while` loop is sometimes referred to as the `do` loop in C. Unlike `for` and `while` loops, this loop checks its condition at the end of the loop, that is after the loop has been executed. This means that the `do.... while` loop will execute at least once, even if the condition is false at first.

The general syntax of the `do... while` loop is:

```
do {
    statement;
} while (condition);
```

The curly brackets are not necessary when only one statement is present within the loop, but it is a good habit to use them. The `do...while` loop iterates until the condition becomes false. In the `do...while` loop the statement (block of statements) is executed first, then the condition is checked. If it is **true**, control is transferred to the `do` statement. When the condition becomes **false**, control is transferred to the statement after the loop.

Consider the following program :

Example 9:

```
/*accept only int values */
#include <stdio.h>
main ()
{
    int num1, num2;
    num2 = 0;
    do{
        printf( "\nEnter a number : ");
        scanf("%d",&num1);
        printf( " No. is %d",num1);
        num2++;
    }while(num1 != 0);
    printf("\nThe total numbers entered were %d",--num2);
```

```
    /* num2 is decremented before printing because count for last integer (0)
is not to be considered */
}
```

A sample output is shown below:

```
Enter a number : 10
No. is 10
Enter a number : 300
No. is 300
Enter a number : 45
No. is 45
Enter a number : 0
No. is 0
The total numbers entered were 3
```

The above code will accept integers and display them until **`zero`** (**`0`**) is entered. It will then, exit from the **`do...while`** loop and print the number of integers entered.

➢ **'Nested while' and 'do...while' Loops**

Like `for` loops, `while` and `do...while` loops can also be nested. An example is given below:

Example 10:

```
#include <stdio.h>
main()
{
    int x;
    char i, ans;
    i = ' ';
    do
    {
        clrscr ();
        x = 0;
        ans = 'y';
        printf("\nEnter sequence of character:");
        do {
            i = getchar ();
            x++;
```

```
        }while (i != '\n');
        i = ' ';
        printf("\nNumber of characters entered is: %d", --x);
        printf("\nMore sequences (Y/N) ?");
        ans = getch ();
    }while(ans == 'Y' || ans == 'y');
}
```

A sample output is shown below:

```
Enter sequence of character:Good Morning !
Number of characters entered is: 14
More sequences (Y/N) ? N
```

This program code first asks the user to enter a sequence of characters till the enter key is hit (nested while). Once the **Enter** key is pressed, the program exits from the inner `do…while` loop. The program then asks the user if more sequences of characters are to be entered. If the user types '**y**' or '**Y**', the outer `while` condition is true and the program prompts the user to enter another sequence. This goes on till the user hits any other key except '**y**' or '**Y**'. The program then ends.

9.2 Jump Statements

C has four statements that perform an unconditional branch: `return`, `goto`, `break`, and `continue`. Unconditional branching means the transfer of control from the point where it is, to a specified statement. Of the above jump statements, `return` and `goto` can be used anywhere in the program, whereas `break` and `continue` statements are used in conjunction with any of the loop statements.

9.2.1 The 'return' Statement

The `return` statement is used to return from a function. It causes execution to return to the point at which the call to the function was made. The `return` statement can have a value with it, which it returns to the program. The general syntax of the `return` statement is:

```
return expression ;
```

The expression is optional. More than one return can be used in a function. However the function will return when the first return is met. The `return` statement will be clear after a discussion on functions.

9.2.2 The 'goto' Statement

Though C is a structured programming language, it contains the following unstructured forms of program control:

- `goto`
- `label`

A `goto` statement transfers control not only to any other statement within the same function in a C program, but it allows jumps in and out of blocks. Therefore, it violates the rules of a strictly structured programming language.

The general syntax of the `goto` statement is,

```
goto label;
```

where **`label`** is an identifier which must appear as a prefix to another C statement in the same function. The semicolon(`;`) after the label identifier marks the end of the `goto` statement. `goto` statements in a program make it difficult to read. They reduce program reliability and make the program difficult to maintain. However, they are used because they can provide useful means of getting out of deeply nested loops. Consider the following code:

```
for(...){
    for(...) {
        for(...) {
            while(..) {
                if(...) goto error1;
                    .
                    .
                    .
                }
            }
        }
    }
error1: printf("Error !!!");
```

As seen, the label appears as a prefix to another statement in the program.

```
label: statement
```

or

Concepts

```
label: {
    statement sequence
}
```

Example 11:

```
# include <stdio.h>
#include <conio.h>
main()
{
    int num ;
    clrscr();
    label1:
        printf("\nEnter a number (1) :");
        scanf("%d",&num);
        if(num==1)
            goto Test;
        else
            goto label1:
    Test:
        printf("All done…");
}
```

A sample output is given below:

```
Enter a number : 4
Enter a number : 5
Enter a number : 1
All done...
```

9.2.3 The 'break' Statement

The `break` statement has two uses. It can be used to terminate a case in the `switch` statement and/or to force immediate ending of a loop, bypassing the normal loop conditional test.

When the `break` statement is met inside a loop, the loop is immediately ended and the program control is passed to the statement following the loop. For example,

Example 12:

```
#include <stdio.h>
main ()
{
    int count1, count2;
    for(count1 = 1, count2 = 0; count1 <=100; count1++)
    {
        printf("Enter %d Count2 : ", count1);
        scanf("%d", &count2);
        if(count2 == 100) break;
    }
}
```

A sample output is shown below:

```
Enter 1 count2 : 10
Enter 2 count2 : 20
Enter 3 count2 : 100
```

In the above code, the user can enter **100** values for **j**. However, if **100** is entered, the loop ends and control is passed to the next statement.

Another point to be remembered while using a `break` is that it causes an exit from an inner loop. This means that if a `for` loop is nested within another `for` loop, and a `break` statement is encountered in the inner loop, the control is passed back to the outer `for` loop.

9.2.4 The 'continue' Statement

The `continue` statement causes the next iteration of the enclosing loop to being. When this statement is met in the program, the remaining statements in the body of the loop are skipped and the control is passed to the re-initialization step.

In case of the `for` loop, `continue` causes the increment portions of the loop and then the conditional test to be executed. In case of the `while` and `do...while` loops, program control passes to the conditional tests. For example:

Example 13:

```
#include <stdio.h>
main ()
```

```
{
    int num;
    for(num = 1; num <=100; num++)
    {
        if (num % 9 == 0) continue;
        printf("%d\t", num);
    }
}
```

The above prints all numbers from **1** to **100**, which are not divisible by **9**. The output will look somewhat like this.

```
1 2 3 4 5 6 7 8 10 11 12 13 14 15 16 17
19 20 21 22 23 24 25 26 28 29 30 31 32 33 34 35
37 38 39 40 41 42 43 44 46 47 48 49 50 51 52 53
55 56 57 58 59 60 61 62 64 65 66 67 68 69 70 71
73 74 75 76 77 78 79 80 82 83 84 85 86 87 88 89
91 92 93 94 95 96 97 98 100
```

9.3 The 'exit()' function

The `exit()` function is a standard C library function. Its working is similar to the working of a jump statement, the major difference being that the jump statements are used to break out of a loop, whereas `exit()` is used to break out of the program. This function causes immediate ending of the program and control is transferred back to the operating system. The `exit()` function is usually used to check if a mandatory condition for a program execution is satisfied or not. The general syntax of the `exit()` function is

```
exit(int return_code);
```

where, **`return_code`** is optional. **`Zero`** is generally used as a **`return_code`** to indicate normal program ending. Other values indicate some sort of error.

Summary

- The loop structures available in C are:
  - The for loop
  - The while loop
  - The do...while loop
- The for loop enables repeated execution statements in C. It uses three expressions, separated by semicolons, to control the looping process. The statement part of the loop can be simple statement or a compound statement.
- The ‘comma’ operator is occasionally useful in the for statements . Of all the operators in C it has the lowest priority.
- The body of a do statement is executed at least once.
- C has four statements that perform an unconditional branch : return, goto, break, and continue.
- The break statement enables early exit from a simple or a nesting of loops. The continue statement causes the next iteration of the loop to begin.
- A goto statement transfers control to any other statement within same function in a C program, but it allows jumps in and out of blocks.
- The exit() function causes immediate termination of the program and control is transferred back to the operating system.

Check Your Progress

1. _____ allows a set of instructions to be performed until a certain condition is reached.

   A. Loop B. Structure

   C. Operator D. None of the above

2. _____ loops check the condition at the top of the loop which means the loop code is not executed, if the condition is false at the start.

   A. `while` loop B. `for` loop

   C. `do..while` loop D. None of the above

3. A _____ is used to separate the three parts of the expression in a `for` loop.

   A. comma B. semicolon

   C. hyphen D. None of the above

4. The _____ loop checks its condition at the end of the loop, that is after the loop has been executed.

   A. `while` loop B. `for` loop

   C. `do..while` loop D. None of the above

5. The _____ statement causes execution to return to the point at which the call to the function was made.

   A. `exit` B. `return`

   C. `goto` D. None of the above

6. The _____ statement violates the rules of a strictly structured programming language.

   A. `exit` B. `return`

Check Your Progress

C. `goto` D. None of the above

7. The _____ function causes immediate termination of the program and control is transferred back to the operating system

A. `exit` B. `return`

C. `goto` D. None of the above

Try It Yourself

1. Write a program to print the series 100, 95 , 90, 85,…….., 5.

2. Accept two numbers num1 and num2. Find the sum of all odd numbers between the two numbers entered.

3. Write a program to generate the Fibonacci series. (1,1,2,3,5,8,13,……).

4. Write a program to display the following patterns.

```
(a) 1
    12
    123
    1234
    12345
```

```
(b) 12345
    1234
    123
    12
    1
```

5. Write a program to generate the following pattern.

```
*******
******
*****
****
***
**
*
```

Session

10 Loop (Lab)

Objectives

At the end of this session, you will be able to:

- *Use the loop structure*
- *Write Some Programs*
  - Using 'for' Loop
  - Using 'while' loop
  - Using 'do…while' loop

The steps given in this session are detailed, comprehensive and carefully thought through. This has been done so that the learning objectives are met and the understanding of the tool is complete. Follow the steps carefully.

Part I – For the first 1 Hour and 30 Minutes :

10.1 Using the 'for' Loop

In this section you will write a program using the '`for`' loop. The program displays even numbers from 1 to 30.

In the program an 'integer' variable, **`num`**, is declared. The '`for`' loop is used to display the even numbers till 30. The first argument of the '`for`' loop, initializes the variable **`num`** to 2. The second argument of the '`for`' loop, checks whether the value of the variable is less than or equal to 30. If this condition is satisfied the statement in the loop is executed. The '`printf()`' statement is used to display the value of the variable **`num`**.

In the third argument, the value of the variable **`num`** is incremented by 2. In C, **`num += 2`** is like **`num = num + 2`**. The '`printf()`' is executed until the second argument is satisfied. Once the value of the variable becomes greater than 30 the condition does not get satisfied and hence the loop is not executed. The curly brackets are not necessary when only one statement is present within the loop, but it is a good programming practice to use them.

Lab Guide

1. **Create a new file.**

2. **Type the following code in the 'Edit window' :**

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int num;
    clrscr();
    printf("\n The even Numbers from 1 to 30 are \n ");
    for (num = 2 ; num <= 30 ; num += 2)
       printf("%d\n",num);
}
```

3. **Save the file with the name, `for.c`**

4. **Compile the file, `for.c`**

5. **Execute the program, `for.c`.**

6. **Return to the editor.**

OUTPUT :

```
The even Numbers from 1 to 30 are
2
4
6
8
10
12
14
16
18
20
22
24
26
28
30
```

10.2 Using the 'while' Loop

In this section you will write a program using the '`while`' loop. The program displays numbers from 10 to 0 in the reverse order.

In this program there is an integer variable **`num`**. The variable is initialized to a value.

Consider the following lines of code:

```
while (num >= 0)
{
    printf("\n%d",num);
    num--;
}
```

The 'while' statement checks, if the value of the variable **`num`** is greater than 0. If the condition is satisfied the 'printf()' is executed and the value of the variable **`num`** is reduced by 1. In C, **`num--`** woks as **`num = num - 1`**. The 'while' loop continues till, the value of the variable is greater than 1 or equal to 0.

1. Create a new file.

2. Type the following code in the 'Edit Window':

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int num;
    clrscr();
    num = 10;
    printf("\n Countdown");
    while(num >= 0)
    {
      printf("\n%d",num);
      num--;
    }
}
```

3. **Save the file with the name, `while.c`.**

4. **Compile the file, `while.c` .**

5. **Execute the program, `while.c`**

6. **Return to the editor.**

OUTPUT:

```
Countdown
10
9
8
7
6
5
4
3
2
1
0
```

10.3 Using do-while Loop

In this section you will write a program that makes use of the 'do-while' loop. The `do` loop differs from the `while` loop only in that it executes the statement before evaluating the expression. An important consequence of this, which you should keep in mind, is that unlike a `while` loop, the contents of a `do` loop will be executed at least once. Because the `while` loop evaluates the expression before executing the statement, if the condition is `false` (zero) right at the beginning, the statement will never be executed.

The program will accept integers and display them until zero (0) is entered. It will then exit from the `do…while` loop and print the number of integers entered.

The program declares two variables **`cnt`** and **`cnt1`**. Inside the `do-while` loop we are entering the number by giving the code below:

```
printf("\nEnter a Number : ");
scanf("%d", &cnt);
```

The code below will display the number entered.

```
printf("No. is %d",cnt);
```

cnt1++ will increment the value of **cnt1** by 1. Suppose if we are entering the number as 0 ,first it will print the value and then check the condition . In this case the condition is `true`. It will come out of the loop and print the value of **cnt1**. **cnt1** is decremented before printing because count for last integer (0) is not to be considered.

1. **Create a new file.**

2. **Type the following code in the 'Edit Window':**

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int cnt, cnt1 ;
    clrscr();
    cnt = cnt1 = 0;
    do
    {
      printf("\nEnter a Number : ");
      scanf("%d", &cnt);
      printf("No. is %d", cnt);
      cnt1++;
    } while ( cnt != 0 );
    printf("\n The total numbers entered were %d", --cnt1);
}
```

3. **Save the file with the name , `dowhile.c`.**

4. **Compile the file, `dowhile.c`.**

5. **Execute the program, `dowhile.c`.**

6. **Return to the editor.**

OUTPUT:

```
Enter a number 11
No is 11

Enter a number 50
No is 50

Enter a number 0
No is 0

The total numbers entered were 2
```

10.4 Using the break statement

The `break` causes immediate exit from a `for`, `while`, `do/while`, or `switch` statement.

The following program demonstrates the use of `break` statement.

Consider the following code:

```
for(cnt = 1; cnt <= 10 ; cnt++)
{
    if (cnt == 5)
      break;
    printf("%d\n",cnt);
}
```

The above code, uses a `for` loop to print the values from 1 to 10. The value of **`cnt`** is initialized to 1 in the beginning, then it will check the condition. If the condition is `true` it will execute the statements inside the `for` loop.

In this case the program prints only 1, 2, 3 and 4 when the value of **`cnt`** becomes 5, the if condition becomes `true` and control will come out of the loop.

1. **Create a new file.**

2. **Type the following code in the 'Edit Window':**

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int cnt ;
    clrscr();
    for(cnt =1 ; cnt <=10 ; cnt++)
    {
      if(cnt==5)
        break;
      printf("%d\t", cnt);
    }
}
```

3. **Save the file with the name, `breakex.c`.**

4. **Compile the file, `breakex.c`.**

5. **Execute the program, `breakex.c`.**

6. **Return to the editor.**

OUTPUT:

```
1    2    3    4
```

10.5 Using the continue statement

The `continue` statement when executed in a `while`, `for`, or `do/while` causes all other statements in that control statement to be skipped and the next iteration to be performed

The following program demonstrates the use of `continue` statement.

Consider the following code:

```
for(cnt = 1; cnt <= 10; cnt++)
{
    if(cnt == 5)
      continue;
    printf("%d\n", cnt);
}
```

The above code uses a `for` loop to print the values from 1 to 10. The value of **`cnt`** is initialized to 1 in the beginning , then it will check the condition. If the condition is `true` it will execute the statements inside the `for` loop.

In this case, the program prints only 1, 2, 3, 4, 6, 7, 8, 9 and 10. When the value of **`cnt`** becomes 5, the if condition becomes `true` and it goes back to the `for` loop again without printing the value 5.

1. Create a new file.

2. Type the following code in the 'Edit Window':

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int cnt ;
    clrscr();
    for(cnt = 1; cnt <= 10; cnt++)
    {
      if(cnt==5)
        continue;
      printf("%d\t", cnt);
    }
}
```

3. Save the file with the name, `breakex.c`.

4. Compile the file, `breakex.c`.

5. Execute the program, `breakex.c`.

6. Return to the editor.

OUTPUT:

```
1    2    3    4    6    7    8    9    10
```

Part II – For the next 30 Minutes:

1. Find the factorial of a number.

   For example,

   - $n! = n * (n-1) * (n-2) * ..... * 1$
   - $4! = 4 * 3 * 2 * 1$
   - $1! = 1$
   - $0! = 1$

   Hint:

   1. Get the number.
   2. To begin, set the factorial of the number to be one.
   3. While the number is greater than one.
   4. Set the factorial to be the factorial multiplied by the number.
   5. Decrement the number.
   6. Print out the factorial.

Try It Yourself

1. Declare a variable which has the age of the person. Print the user's name as many times as his age.

2. Write a program to generate the following pattern:

   1

   1 2

   1 2 3

   1 2 3 4

   1 2 3 4 5

   1 2 3 4 5 6

   1 2 3 4 5 6 7

   1 2 3 4 5 6 7 8

   1 2 3 4 5 6 7 8 9

3. Write a program to print a multiplication table for a given number.

Session

11 Arrays

Concepts

Objectives

At the end of this session, you will be able to:

- *Explain array elements and indices*
- *Define an array*
- *Explain array handling in C*
- *Explain how an array is initialized*
- *Explain string / character arrays*
- *Explain two dimensional arrays*
- *Explain initialization of multidimensional arrays*

Introduction

It may be difficult to store a collection of similar data elements in different variables. For example, consider that the scores for all the 11 players of a soccer team have to be recorded for a match. Storing the score of each player in variables having unique names is certainly more tiresome than having a common variable for them. This is what an array does .An array is a collection of data elements of the same type. Each element of the array has the same data type, same storage class and same characteristics. Each element is stored in successive locations of the main store. These elements are known as **members** of the array.

11.1 Array Elements and Indices

Each member of an array is identified by an unique **index** or **subscript** assigned to it. The **dimension** of an array is determined by the number of indices needed to uniquely identify each element. An index is a positive integer enclosed in square brackets [ ] placed immediately after the array name, without a space in between. An **index** holds integer values starting with zero. Thus, an array **`player`** with 11 elements will be represented as,

```
player[0], player[1], player[2],.... player[10].
```

As is seen, the array element starts with **`player[0]`**, and so the last element is **`player[10]`** and not **`player[11]`**. This is because in C, an array index starts from 0; and so for an array of **`N`** elements, the last element has an index of **`N-1`**. The limits of the allowed index values are called the **bounds** of the

array index, the **Lower** bound and the **Upper** bound. A valid index must have an integer value either between the bounds or equal to one of them. The term **valid** is used for a very specific reason. In C, if the user tries to access an element outside the legal index range (like **`player[11]`** in the above example of an array), it will not generate an error as far as the C compiler is concerned. However, it may access some value which can lead to unpredictable results. There is also a danger of overwriting data or program code. Hence the programmer has to ensure that all indices are within the valid bounds.

Defining an Array

An array has some particular characteristics and has to be defined using them. These characteristics include:

- The **storage** class
- The **data type** of the elements of the array
- The **array name** which indicates the location of the first member of the array
- Array size, a constant which is an integer expression evaluating to a positive value

An array is defined in the same way as a variable is defined, except that the array name is followed by one or more expressions, enclosed within square brackets, [ ], specifying the array dimension. The general syntax of an array definition is:

```
storage_class data_type array_name[size_expr]
```

Here, **`size_expr`** is an expression denoting the number of members in the array and must evaluate to a **positive integer**. Storage class is optional. The array defaults to class **automatic** for arrays defined within a function or a block and **external** for arrays defined outside a function. The array **`player`** will therefore, be declared as

```
int player[11];
```

Remember that while defining the array, the array size will be 11, though the indices of individual members of the array will range from 0 to 10.

The rules for forming array names are the same as those for variable names. An **array name** and a **variable name** cannot be the same as it leads to ambiguity. If such a declaration is given in a program, the compiler displays an error message.

- **Some Norms with Arrays**

  - All elements of an array are of the same type. This means that if an array is declared of type `int`, it cannot contain elements of any other types.

- Each element of an array can be used wherever a variable is allowed or required.
- An element of an array can be referred using a variable or an integer expression. The following are valid references:

```
player[i]; /* Where i is a variable, though care has to be
taken that i is within the range of the
defined subscript for the array player.*/
player[3] = player[2] + 5;
player[0] += 2;
player[i/2+1];
```

- Arrays can have their data type as `int`, `char`, `float`, or `double`.

11.2 Array handling in C

An array is treated differently from a variable in C. Two arrays, even if they are of the same type and size, cannot be tested for equality. Moreover, it is not possible to assign one array directly to another. Instead each array element has to be assigned separately to the corresponding element of another array. Values cannot be assigned to an array as a whole, except at the time of initialization. However, individual elements can not only be assigned values but can also be compared.

```
int player1[11], player2[11];
for( i=0; i<11; i++)
    player1[i] = player2[i];
```

The same can also be attained by using separate assignment statements as follows:

```
player1[0] = player2[0];
player1[1] = player2[1];
.
.
player1[10] = player2[10];
```

The `for` construct is the ideal way of manipulating arrays.

Example 1:

```
/* Program demonstrates a single dimensional array */
#include <stdio.h>
void main()
```

Concepts

```
{
    int num[5];
    int i;
    num[0] = 10;
    num[1] = 70;
    num[2] = 60;
    num[3] = 40;
    num[4] = 50;
    for(i=0;i<5;i++)
        printf("\n Number at [%d] is %d" ,i, num[i]);
}
```

The output is shown below:

```
Number at [0] is 10
Number at [1] is 70
Number at [2] is 60
Number at [3] is 40
Number at [4] is 50
```

The example given below accepts values into an array of size 10 and displays the highest and average values.

Example 2:

```
/* Input values are accepted from the user into the array ary[10]*/
#include <stdio.h>
void main()
{
    int ary[10];
    int i, total, high;
    for(i=0; i<10; i++)
    {
        printf("\n Enter value: %d : ", i+1);
        scanf("%d",&ary[i]);
    }
    /* Displays highest of the entered values */
    high = ary[0];
    for(i=1; i<10; i++)
```

```
    {
        if(ary[i] > high)
        high = ary[i];
    }
    printf("\nHighest value entered was %d", high);
    /* prints average of values entered for ary[10] */
    for(i=0,total=0; i<10; i++)
        total = total + ary[i];
    printf("\nThe average of the elements of ary is %d",total/i);
}
```

A sample output is shown below:

```
Enter value: 1 : 10
Enter value: 2 : 20
Enter value: 3 : 30
Enter value: 4 : 40
Enter value: 5 : 50
Enter value: 6 : 60
Enter value: 7 : 70
Enter value: 8 : 80
Enter value: 9 : 90
Enter value: 10 : 10
Highest value entered was 90
The average of the elements of ary is 46
```

- **Array Initialization**

Automatic arrays cannot be initialized, unless each element is given a value separately. Automatic arrays should not be used without proper initialization, as the results in such cases are unpredictable. This is because the allocated storage locations assigned to the array are not automatically initialized. Whenever elements of such an array are used in arithmetic expressions, the previously existing values will be used, which are not guaranteed to be the same type as the array definition, unless the array elements are very clearly initialized. This is true not only for arrays but also for ordinary variables.

In the following code snippet, the array elements have been assigned values using a `for` loop.

```
int ary[20], i;
for(i=0; i<20; i++)
    ary[i] = 0;
```

Initializing an array using the `for` loop can be done using a constant value or values which are in arithmetic progression.

The `for` loop can also be used to initialize an array of alphabets as follows:

Example 3:

```
#include <stdio.h>
void main()
{
    char alpha[26];
    int i, j;
    for(i=65,j=0; i<91; i++,j++)
    {
        alpha[j] = i;
        printf("The character now assigned is %c \n", alpha[j]);
    }
    getchar();
}
```

The partial output of the above code is:

```
The character now assigned is A
The character now assigned is B
The character now assigned is C
.
.
```

The above program assigns ASCII character codes to the elements of the character array **`alpha`**. This, when printed using the `%c` specifier, results in a series of characters printed one after the other. Arrays can be initialized when they are defined. This can be done by assigning a list of values separated by commas and enclosed in curly braces { } to the array name. The values within the curly braces are assigned to the elements of the array in the order of their appearance.

For example,

```
int deci[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
static float rates[4] = {0.0, -2.5, 13.75, 18.0};
char company[5] = {'A', 'P', 'P', 'L', 'E'};
int marks[100] = {15, 13, 11, 9}
```

Arrays **deci**, **company** and **marks** are assumed to be external arrays by virtue of their position in the program. The initializing values must be constants and cannot be variables or expressions. The first few elements of the array will be initialized if the number of initializing values is fewer than the defined array dimension. The remaining elements will be initialized to zero. For example, in array **marks** after the above initialization, the first four elements (0 through 3) are initialized to 15, 13, 11 and 9 respectively. The rest of the elements are initialized to the value zero. It is not possible to initialize only elements 1 through 4, or 2 through 4, or 2 through 5 when initialization is done at the time of declaration. There is no facility in C to specify repetition of an initializing value.

In case of an explicit initialization, `class extern` or `static`, the array elements are guaranteed (unlike class `auto`) to be initialized to zero.

It is not necessary to declare the size of the array that is being initialized. If the size of the array is omitted, the compiler will compute the array size by counting the initializing values. For example, the following **external** array declaration will assess the dimension of the array **ary** to be 5 as there are five initializing values.

```
int ary[] = {1, 2, 3, 4, 5};
```

- **Strings / Character Arrays**

A **string** can be defined as a character type array, which is terminated by a NULL character. Each character of a string occupies one byte and the last character of a string is always the character '\0'. This `'\0'` is called the **null character**. It is an escape sequence like `'\n'` and stands for a character with a value of 0 (zero). As '\0' is always the last character of a string, character arrays have to be one character longer than the longest string they are required to hold. For example, an array, for example **ary**, that holds a 10- character string should be declared as:

```
char ary[11];
```

The additional location is used for storing the null character. Remember that the ending character (null character) is very important.

The string values can be input using the `scanf()` function. For the string **ary** declared above, the code statement to accept the value will be as follows:

```
scanf("%s",ary);
```

In the above statement, **ary** defines the location where the successive characters of the string will be stored.

Concepts

Example 4:

```
#include <stdio.h>
void main()
{
    char ary[5];
    int i;
    printf("\n Enter string :" );
    scanf("%s",ary);
    printf("\n The string is %s \n\n", ary);
    for (i=0; i<5; i++)
        printf("\t%d", ary[i]);
}
```

The outputs for various inputs are as follows:

If the entered string is **appl**, the output will be:

```
The string is appl
97 112 112 1 08 0
```

The input for the above is of 4-characters (**appl**) and the 5th character is the null character. This is clear from the ASCII codes for the characters printed in the second line. The fifth character is printed as 0, which is the value of the null character.

If the entered string is **apple**, the output will be:

```
The string is apple
97 112 112 108 101
```

The above output is for an input of 5 characters namely a, p, p, l, & e. This is not considered a string because the 5th character of this array is not \0. Again, this is clear from the line giving the ASCII codes of the characters a, p, p, l, e.

If the entered string is **ap**, the output will be as shown below.

```
The string is ap
97 112 0 6 100
```

In the above example, when only two characters are entered the third character is the null character. This actually indicates that the string has ended. The remaining characters are unpredictable

characters.

In the above case, the importance of the null character becomes clear. The null character actually specifies the end of the string and is the only way in which functions that work with the string will know where the end of string is.

Although C does not have a string data type, it allows string constants. A string constant is a list of characters enclosed in double quotes(“ ”). Like any other constant, it cannot be altered in the program. An example is,

```
"Hi Aptechite!"
```

The `null` character is added automatically at the end of the string by the C compiler.

C supports a wide range of string functions, which are found in the standard header file `string.h`. A few of these functions are given in Table 11.1. The working of these functions will be discussed in Session 17.

Name	Function
`strcpy(s1, s2)`	Copies **s2** into **s1**
`strcat(s1, s2)`	Joins **s2** onto the end of **s1**
`strlen(s1)`	Returns the length of **s1**
`strcmp(s1, s2)`	Returns 0 if **s1** and **s2** are the same; less than 0 if **s1<s2**; greater than 0 if **s1>s2**
`strchr(s1, ch)`	Returns a pointer to the first occurrence of **ch** in **s1**
`strstr(s1, s2)`	Returns a pointer to the first occurrence of **s2** in **s1**

Table 11.1: A few String functions in C

11.3 Two Dimensional Arrays

So far, we have dealt with arrays that were **single-dimensional** arrays. This means that the arrays had only one subscript. Arrays can have more than one dimension. **Multidimensional** arrays make it easier to represent multidimensional objects, such as a graph with rows and columns or the screen of the monitor. Multidimensional arrays are defined in the same way as single-dimensional arrays, except that a separate pair of square brackets [ ], is required in case of a two dimensional array. A three dimensional array will require three pairs of square brackets and so on. In general terms, a multidimensional array can be represented as

```
storage_class data_type ary[exp1][exp2]....[expN];
```

where **ary** is an array having a storage class **storage_class**, data type **data_type**, and **exp1**, **exp2**, up to **expN** are positive integer expressions that indicate the number of array elements associated with

Concepts

each subscript.

The simplest and most commonly used form of multidimensional arrays is the two-dimensional array. A two dimensional array can be thought of as an array of two 'single-dimensional' arrays. A typical two-dimensional array is the airplane or railroad timetable. To locate the information, the required row and column is determined, and information is read from the location at which they (the row and column) meet. In the same way, a two-dimensional array is a grid containing rows and columns in which each element is uniquely specified by means of its row and column coordinates. A two-dimensional array **tmp** of type `int` with 2 rows and 3 columns can be defined as,

```
int tmp[2][3];
```

This array will contain 2 X 3 (6) elements and they can be represented as:

		Column		
		0	**1**	**2**
Row	**1**	e1	e2	e3
	2	e4	e5	e6

where **e1 – e6** represent the elements of the array. Both rows and columns are numbered from 0 onwards. The element **e6** is designated by row 1 and column 2. To access this element the representation will be:

```
tmp[1][2];
```

➢ **Initialization of Multidimensional Arrays**

A multidimensional array definition can include the assignment of initial values. Care must be taken regarding the order in which the initial values are assigned to the array elements (only external and static arrays can be initialized). The elements of the first row of two-dimensional array will be assigned values first, and then the elements of the second row, and so on. Consider the following array definition:

```
int ary[3][4] ={1,2,3,4,5,6,7,8,9,10,11,12};
```

The result of the above assignment will be as follows:

```
ary[0][0] = 1    ary[0][1] = 2    ary[0][2] = 3    ary[0][3] = 4
ary[1][0] = 5    ary[1][1] = 6    ary[1][2] = 7    ary[1][3] = 8
ary[2][0] = 9    ary[2][1] = 10   ary[2][2] = 11   ary[2][3] = 12
```

Note that the first subscript ranges from 0 to 2 and the second subscript ranges from 0 to 3. A point to remember is that array elements will be stored in neighbouring (side by side) locations in memory. The above array **ary** can be thought of as an array of 3 elements, each of which is an

array of 4 integers, and will appear as,

Row 0	Row 1	Row 2
1 2 3 4	5 6 7 8	9 10 11 12

The natural order in which the initial values are assigned can be altered by forming groups of initial values enclosed within braces. Consider the following initialization.

```
int ary [3][4] ={
{1, 2, 3},
{4, 5, 6},
{7, 8, 9}
};
```

The array will be initialized as follows

```
ary[0][0]=1    ary[0][1]=2    ary[0][2]=3    ary[0][3]=0
ary[1][0]=4    ary[1][1]=5    ary[1][2]=6    ary[1][3]=0
ary[2][0]=7    ary[2][1]=8    ary[2][2]=9    ary[2][3]=0
```

An element of a multidimensional array can be used as a variable in C provided the proper number of subscripts is given with the array element.

Example 5:

```
/* Program to accept numbers in a two dimensional array. */
#include <stdio.h>
void main()
{
    int arr[2][3];
    int row,col;
    for(row=0;row<2;row++)
    {
        for(col=0;col<3;col++)
        {
             printf("\nEnter a Number at [%d][%d] : ",row,col);
             scanf("%d",&arr[row][col]);
         }
    }
```

Concepts

```
    for(row=0;row<2;row++)
    {
        for(col=0;col<3;col++)
        {
            printf("\nThe Number at [%d][%d] is %d", row, col,
arr[row][col];
        }
    }
}
```

A sample output is given below :

```
Enter a Number at [0][0] : 10
Enter a Number at [0][1] : 100
Enter a Number at [0][2] : 45
Enter a Number at [1][0] : 67
Enter a Number at [1][1] : 45
Enter a Number at [1][2] : 230
The Number at [0][0] is 10
The Number at [0][1] is 100
The Number at [0][2] is 45
The Number at [1][0] is 67
The Number at [1][1] is 45
The Number at [1][2] is 230
```

- **Two-dimensional Arrays and Strings**

As seen earlier, a string can be represented as a single-dimensional, character-type array. Each character within the string will be stored within one element of the array. This array of the string can be created using a two-dimensional character array. The left index or subscript determines the number of strings and the right index specifies the maximum length of each string. For example, the following declares an array of 25 strings, each with a maximum length of 80 characters including the null character.

```
char str_ary[25][80];
```

- **Example demonstrating the use of a Two-dimensional Array**

The following example demonstrates the use of two-dimensional arrays as strings.

Consider the problem of organizing a list of names in an alphabetical order. The following example

accepts a list of names and then arranges them in alphabetical order.

Example 6:

```
#include <stdio.h>
#include <string.h>
#include <conio.h>
void main ()
{
    int i, n = 0;
    int item;
    char x[10][12];
    char temp[12];
    clrscr();
    printf("Enter each string on a separate line \n\n");
    printf("Type 'END' when over \n\n");
    /* read in the list of strings */
    do
    {
         printf("String %d : ", n+1);
         scanf("%s", x[n]);
    } while (strcmp(x[n++], "END"));
    /*reorder the list of strings */
    n = n - 1;
    for(item=0; item<n-1; ++item)
    {
         /* find lowest of remaining strings */
         for(i=item+1; i<n; ++i)
         {
             if(strcmp (x[item], x[i]) > 0)
             {
                 /*interchange two stings */
                 strcpy(temp, x[item]);
                 strcpy(x[item], x[i]);
                 strcpy(x[i], temp);
             }
          }
    }
```

```
    /* Display the arranged list of strings */
    printf("Recorded list of strings : \n");
    for(i = 0; i < n ; ++i)
    {
        printf("\nString %d is %s", i+1, x[i]);
    }
}
```

The program accepts strings till the user types END. When END is typed, the program sorts the list of strings and prints it out in sorted order. The program checks two consecutive elements. If they are not in proper order, then they are interchanged. The comparison of two strings is done with the help of the `strcmp()` function whereas the interchanging is done with the `strcpy()` function.

A sample output of the above example is as follows:

```
Enter each string on a separate line
Type 'END' when over
String 1 : has
String 2 : seen
String 3 : alice
String 4 : wonderland
String 5 : END
Record list of strings:
String 1 is alice
String 2 is has
String 3 is seen
String 4 is wonderland
```

Summary

- An array is a collection of data elements of the same type that are referenced by a common name.
- Each element of the array has the same data type, same storage class and common characteristics.
- Each element is stored in successive locations of the main store. The data elements are known as the members of the array.
- The dimension of an array is determined by the number of indices needed to uniquely identify each element.
- Arrays can have the data type as int, char, float, or double.
- The element of an array can be referred using a variable or an integer expression.
- Automatic arrays cannot be initialized, unless each element is given a value separately.
- The extern and static arrays can be initialized when they are defined.
- A two-dimensional array can be thought of as an array of single-dimensional arrays.

Concepts

Check Your Progress

1. An _____ is a collection of data elements of the same type that are referred by a common name.

   A. Loop B. Array

   C. Structure D. None of the above

2. Each member of an array is identified by the unique _____ or _____ assigned to it.

   A. Index, Subscript B. Bound, Index

   C. None of the above

3. An array name and a variable name can be the same. (T/F)

4. Each element of an array cannot be used where a variable is allowed or required. **(T/F)**

5. Two arrays, even if they are of the same type and size, cannot be tested for _____.

   A. Condition B. Negation

   C. Equality D. None of the above

6. String can be defined as a character type array, which is terminated by a _____ character.

   A. semicolon B. comma

   C. NULL D. None of the above

7. Arrays can have more than one dimension. **(T/F)**

8. The comparison of two strings is done with the help of _____ whereas the interchanging is done by _____.

   A. `strcmp`,`strcpy` B. `strcat`,`strcpy`

   C. `strlen`,`strcat` D. None of the above

Try It Yourself

1. Write a program to arrange the following names in alphabetical order.

   George

   Albert

   Tina

   Xavier

   Roger

   Tim

   William

2. Write a program to count the number of vowels in a line of text.

3. Write a program that accepts the following numbers in an array and reverses the array.

   34

   45

   56

   67

   89

“The only way to predict the future is to invent it”

Session

12 Arrays (Lab)

Objectives

At the end of this session, you will be able to:

- *Use a Single Dimensional Array*
- *Use a Two Dimensional Array*

Introduction

The steps given in this session are detailed, comprehensive and carefully thought through. This has been done so that the learning objectives are met and the understanding of the tool is complete. Please follow the steps carefully.

Part I – For the first 1 Hour and 30 Minutes:

12.1 Arrays

Arrays can be classified into two types based on dimensions: Single dimensional or Multi dimensional. In this session, let us focus on how to create and use arrays.

12.1.1 Sorting a single dimensional array

A single dimensional array can be used to store a single set of values of same data type. Consider a set of student marks in a particular subject. Let us sort these marks in descending order.

The steps to sort a single dimensional array in descending order are:

1. Accept the total number of marks to be entered.

To do this, a variable has to be declared and the value for the same has to be accepted. The code will be:

```
int n;
printf("\n Enter the total number of marks to be entered : ");
scanf("%d",&n);
```

2. Accept the set of marks.

To accept a set of values for an array the array should be declared. The code for the same is,

```
int num[100];
```

The number of elements in the array is determined by the value entered for the variable **n**. The **n** elements of the array should be initialized with values. To accept **n** values, a `for` loop is required. An integer variable needs to be declared as the subscript of the array. This variable helps in accessing individual elements of the array. The values of the array elements are then initialized by accepting values from the user. The code for the same is:

```
int l;
for(l=0;l<n;l++)
{
    printf("\n Enter the marks of student %d : ", l+1);
    scanf("%d",&num[l]);
}
```

As subscripts of the array always start from 0 and we need to initialize 1 to 0. Each time the `for` loop is executed, an integer value is assigned to the element of the array.

3. Make a copy of the array.

Before sorting the array, it is always safer to preserve the original array. Hence another array can be declared and the elements of the first array can be copied into this new array. The following lines of code are used to do this:

```
int desnum[100],k;

for(k=0;k<n;k++)
    desnum[k] = num[k];
```

4. Sort the array in descending order.

To sort an array, the elements in the array need to be compared with each other. The best way of sorting an array, in descending order, is to pick the highest value of the array and exchange it with the first element. Once this is done, the second highest element from the remaining elements can be exchanged with the second element of the array. While doing so, the first element of the array can be ignored as it is the highest of all. Similarly, the elements of the array can be eliminated one by one till the nth highest element is found. In case the array needs to be sorted in ascending order the highest value can be exchanged with the last element of the array.

Let us consider a numerical example to understand the same. Figure 12.1 represents an array of integers that need to be sorted.

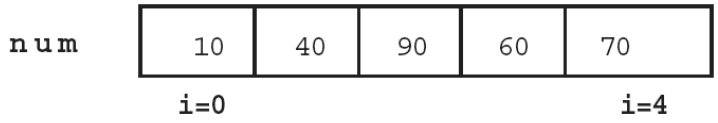


Figure 12.1: Array num with subscript i (5 elements)

To sort this array in descending order,

a. We need to find the first highest element and place it in the first position. This can be referred to as the first pass. To get the highest element to the first position, we need to compare the first element with the rest of the elements. In case the element being compared is greater than the first element then the two elements need to be exchanged.

To start with, in the first pass, the element in the first place is compared with the element in the second place. Figure 12.2 represents the exchange of the element in the first place.

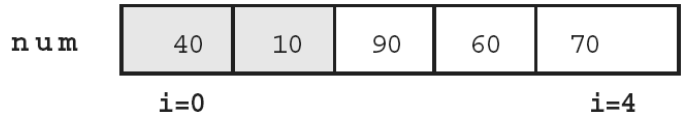


Figure 12.2: Swapping the first element with the second element

The first element is then compared with the third element. Figure 12.3 represents the exchange of the first and the third elements.

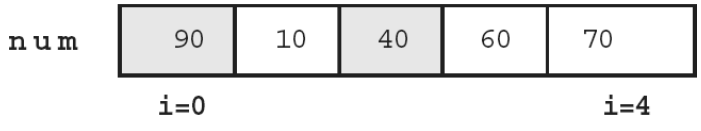


Figure 12.3: Swapping the first element with the third element

This process is repeated until the first element is compared with the last element of the array. The resultant array after the first pass will be as shown in figure 12.4.

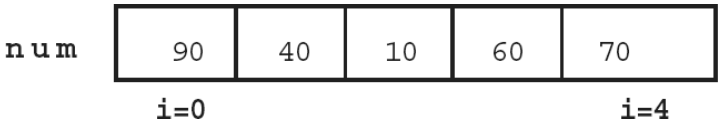


Figure 12.4: Array after the first pass

b. Leaving the first element intact, we need to find the second highest element and swap it with the second element of the array. Figure 12.5 represents the array after doing so.

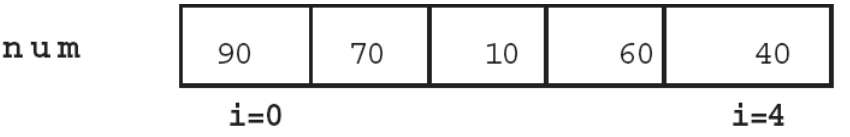


Figure 12.5: Array after second pass

c. The third element has to be swapped with the third highest element of the array. Figure 12.6 represents the array after swapping the third highest element.

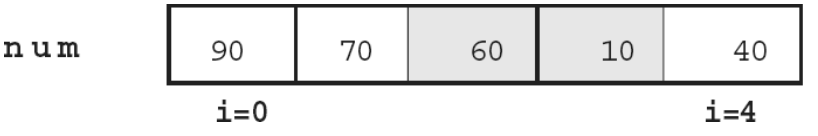


Figure 12.6: Array after third pass

d. The fourth element has to be swapped with the fourth highest element of the array. Figure 12.7 represents the array after swapping the fourth highest element.

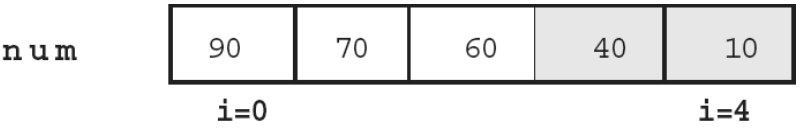


Figure 12.7: Array after fourth pass

e. Figure 12.7 also represents the sorted array.

To do this programmatically, we need two loops one to find the highest element in an array and another to repeat the process **n** times. To be more specific the process has to be repeated **n-1** times `for` an n element array because the last element will not have any other element to be compared with. Hence we declare two variables named **i** and **j** to work through two `for` loops. The `for` loop with index **i** is used to repeat the process of locating highest element in the remaining portion of the array. The `for` loop with index **j** is used to find the **i**th highest element of the array between the **i+1**th and the last element of the array. Thus the **i**th highest element in the remaining part of the array will be pushed to the **i**th position.

The code for declaring the subscripts and looping through the array **n-1** times with **i** as index is,

```
int i,j;
for(i=0;i<n-1;i++)
{
```

The code for looping from the **i+1**th element to the nth element of the array is,

```
    for(j=i+1;j<n;j++)
    {
```

To swap two elements in an array we need to use a temporary variable. This is because the moment the one element of the array is copied onto another element, the value in the second element is lost. To avoid the loss of value in the second element, the value needs to be preserved in a temporary variable. The code for swapping the **i**th element with the greatest element in the remaining part of the array is,

```
      if(desnum[i] < desnum[j])
      {
        temp = desnum[i];
        desnum[i] = desnum[j];
        desnum[j] = temp;
      }
   }
}
```

The for loops need to be closed and hence two extra closing parenthesis are given in the above code.

5. Display the sorted array.

The same subsrcipt **i** can be used to display the values of the array as shown in the statements below:

```
for(i=0;i<n;i++)
    printf("\n Number at [%d] is %d", i, desnum[i]);
```

Thus an array of elements can be sorted. Let us look at the complete program.

1. Invoke the editor in which you can type the C program.

2. Create a new file.

Lab Guide

3. Type the following code :

```
void main()
{
    int n;
    int num[100];
    int l;
    int desnum[100],k;
    int i,j,temp;
    printf(”\n Enter the total number of marks to be entered : “);
    scanf(“%d”,&n);
    clrscr();
    for(l=0;l<n;l++)
    {
        printf(“\n Enter the marks of student %d : ”, l+1);
        scanf(“%d”,&num[l]);
    }

    for(k=0;k<n;k++)
        desnum[k] = num[k];

    for(i=0;i<n-1;i++)
    {
        for(j=i+1;j<n;j++)
        {
            if(desnum[i] < desnum[j])
            {
                temp= desnum[i];
                desnum[i] = desnum[j];
                desnum[j] = temp;
            }
        }
    }
    for(i=0;i<n;i++)
        printf(“\n Number at [%d] is %d”, i, desnum[i]);
}
```

To see the output, follow the steps listed below:

4. **Save the file with the name arrayI.C**
5. **Compile the file, arrayI.C**
6. **Execute the program, arrayI.C**
7. **Return to the editor.**

The sample output of the above program is shown in Figure 12.8 and 12.9.

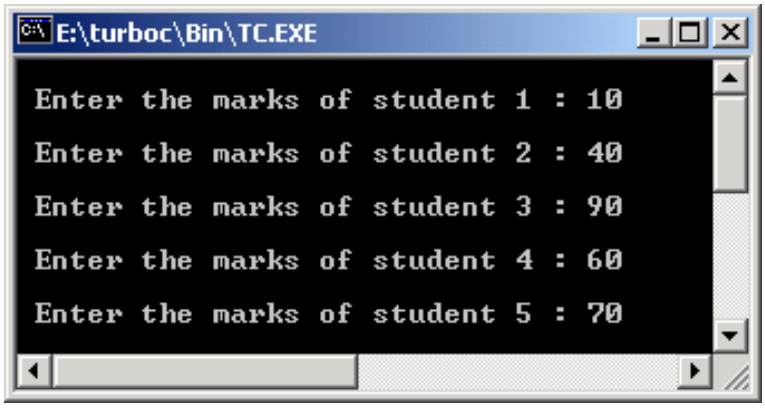


Figure 12.8: Output I of arrayI.C – Input Values

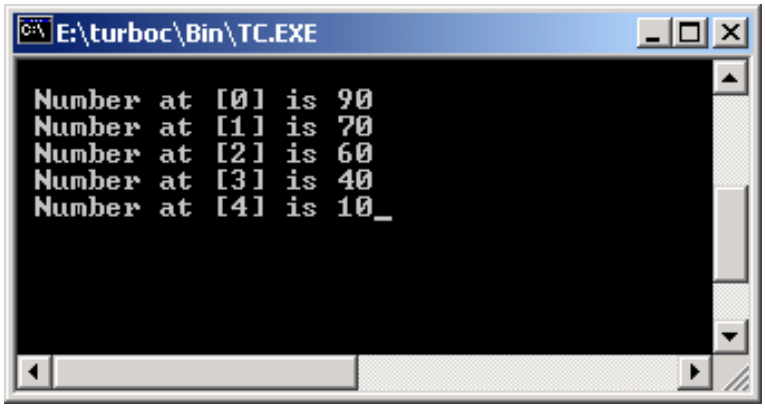


Figure 12.9 : Output II of arrayI.C – Output Values

12.1.2 Matrix addition using two dimensional arrays

Arrays can have multiple dimensions. A typical example of a two dimensional array is a matrix. A matrix is a rectangular number pattern that is made up of rows and columns. The intersection of each row and

column has a value. Figure 12.10 represents a matrix.

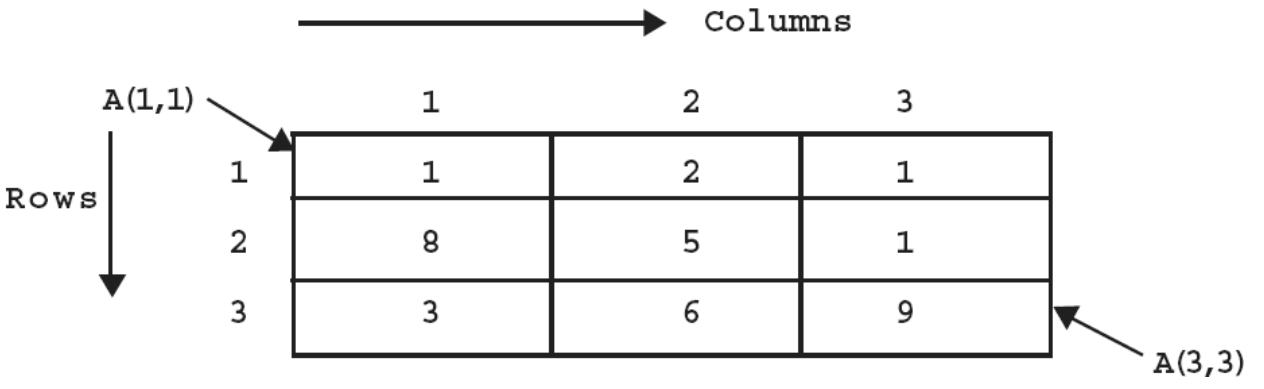


Figure 12.10 : Matrix named A

The number of rows and columns in a matrix represented as (number of rows) x (number of columns) is called the dimension of the matrix. The dimension of the matrix in Figure 12.10 is 3x3.

Let us consider the example of matrix addition for understanding the use of two dimensional arrays. Consider the following matrices A and B given in Figure 12.11.

A	1	2	3
1	1	2	1
2	8	5	1
3	3	6	9

B	1	2	3
1	2	1	4
2	3	4	2
3	8	9	0

Figure 12.11: Matrix A and B

The sum of these two matrices is another matrix. This is created by adding the values from each of the corresponding rows and columns. For example, the first element `C(1,1)` in matrix `C` will be the sum of `A(1,1)` and `B(1,1)`. The second element `C(1,2)` will be the sum of `A(1,2)` and `B(1,2)` and so on. An important rule in summing matrices is that the dimension of the matrices that are being added should be same. That is, a 2x3 matrix can be added with another 2x3 matrix only. Figure 12.12 represents the matrices `A`, `B` and `C`.

A	1	2	3
1	1	2	1
2	8	5	1
3	3	6	9

B	1	2	3
1	2	1	4
2	3	4	2
3	8	9	0

C	1	2	3
1	3	3	5
2	11	9	3
3	11	15	9

Figure 12.12 : Matrix A, B and C

To do this programmatically,

1. Declare three two dimensional arrays. The code for the same is,

```
int A[10][10], B[10][10], C[10][10];
```

2. Accept the dimension of the matrices. The code is,

```
int row,col;
printf("\n Enter the dimension of the matrix : ");
scanf("%d %d",&row,&col);
```

3. Accept the values of the matrix A and B.

The values of a matrix are accepted row wise. First the values of the first row are accepted. Then the values of the second row are accepted and so on. Within a row the values of each column are accepted sequentially. Hence two for loops are necessary to accept the values of a matrix. The first for loop traverses through the rows one by one, whereas the inner for loop will traverse through the columns one by one.

The code for the same is,

```
printf("\n Enter the values of the matrix A and B : \n");
int i, j;
for(i=0;i<row;i++)
    for(j=0;j<col;j++)
    {
        print("A[%d,%d],B[%d,%d]:",row,col,row,col);
        scanf("%d %d",&A[i][j],&B[i][j]);
    }
```

4. Add the two matrices. The two matrices can be added using the following code,

```
C[i][j] = A[i][j] + B[i][j];
```

Note this line needs to be inside the inner for loop of the code given previously. Alternatively, the two for loops can be re-written to add the matrices.

5. Display the three matrices. The code for the same will be,

```
for(i=0;i<row;i++)
    for(j=0;j<col;j++)
    {
      printf("\nA[%d,%d]=%d,B[%d,%d]=%d,C[%d,%d]=%d \n",i,j,A[i][j],i,j
,B[i][j],i,j,C[i][j]);
    }
```

Let us look at the complete program.

1. Create a new file.

2. Type the following code :

```
void main()
{
    int A[10][10], B[10][10], C[10][10];
    int row,col;
    int i,j;
    printf("\n Enter the dimension of the matrix : ");
    scanf("%d %d",&row,&col);
    printf("\nEnter the values of the matrix A and B: \n");

    for(i=0;i<row+i++)
        for(j=0;j<col;j++)
        {
            print("\n A[%d,%d] , B[%d,%d] : ",i,j,i,j);
            scanf("%d %d",&A[i][j],&B[i][j]);
            C[i][j] = A[i][j] + B[i][j];
        }
```

```
    for(i=0;i<row+i++)
        for(j=0;j<col;j++)
        {
            printf("\n A[%d,%d]=%d,B[%d,%d]=d,C[%d,%d]=%d\n",i,j,A[i][j
],i,j,B[i][j],i,j,C[i][j]);
        }
}
```

3.	**Save the file with the name arrayII.C**
4.	**Compile the file, arrayII.C**
5.	**Execute the program, arrayII.C**
6.	**Return to the editor.**

The sample output of the above program will be as shown in Figure 12.13.

```
C:\WINNT\System32\cmd.exe - tc
Enter the dimension of the matrix : 3 3
Enter the values of the matrix A and B :
A[0,0] , B[0,0] : 1 2
A[0,1] , B[0,1] : 2 1
A[0,2] , B[0,2] : 1 4
A[1,0] , B[1,0] : 8 3
A[1,1] , B[1,1] : 5 4
A[1,2] , B[1,2] : 1 2
A[2,0] , B[2,0] : 3 8
A[2,1] , B[2,1] : 6 9
A[2,2] , B[2,2] : 9 0
A[0,0] = 1 , B[0,0] = 2 , C[0,0] = 3
A[0,1] = 2 , B[0,1] = 1 , C[0,1] = 3
A[0,2] = 1 , B[0,2] = 4 , C[0,2] = 5
A[1,0] = 8 , B[1,0] = 3 , C[1,0] = 11
A[1,1] = 5 , B[1,1] = 4 , C[1,1] = 9
A[1,2] = 1 , B[1,2] = 2 , C[1,2] = 3
A[2,0] = 3 , B[2,0] = 8 , C[2,0] = 11
A[2,1] = 6 , B[2,1] = 9 , C[2,1] = 15
A[2,2] = 9 , B[2,2] = 0 , C[2,2] = 9
```

Figure 12.13 : Output I of arrayII.C – Input Values

Part II – For the next 30 Minutes :

1. Write a C program that accepts a set of numbers in an array and reverses the array.

   To do this,

   a. Declare two arrays.

   b. Accept the values of one array.

   c. Loop through the array in the reverse order to copy the values into the second array. While doing so have a different subscript to the second array which will move forward.

Try It Yourself

1. Write a C program to find the minimum and the maximum value in an array.

2. Write a C program to count the number of vowels and the number of consonants in a word.

Session

13 Pointers

Concepts

Objectives

At the end of this session, you will be able to:

- *Explain what a pointer is and where it is used*
- *Explain how to use pointer variables and pointer operators*
- *Assign values to pointers*
- *Explain pointer arithmetic*
- *Explain pointer comparisons*
- *Explain how pointers can be passed as arguments to functions*
- *Explain pointers and single dimensional arrays*
- *Explain Pointer and multidimensional arrays*
- *Explain how allocation of memory takes place*

Introduction

A pointer provides a way of accessing a variable without referring to the variable directly. It provides a symbolic way of using addresses. This session deals with the concept of pointer and its usage in C. Also, we shall discuss a few concepts which are associated with pointers.

13.1 What is a Pointer?

A **Pointer** is a variable, which contains the address of a memory location of another variable, rather than the stored value of that variable. If one variable contains the address of another variable, the first variable is said to **point** to the second. A pointer provides an indirect way of accessing the value of a data item. Consider two variables **`var1`** and **`var2`** such that **`var1`** has a value of `500` and is stored in the memory location `1000`. If **`var2`** is declared as a pointer to the variable **`var1`**, the representation will be as follows:

Memory location	Value stored	Variable name
1000	500	var1
1001		
1002		
.		
.		
1108	1000	var2

Here, **var2** contains the value `1000`, which is nothing but the address of the variable **var1**.

Pointers can point to variables of other fundamental data type variables like `int, char`, or `double` or data aggregates like `arrays`.

13.1.1 Why are Pointers used?

Few of the situations where pointers can be used are:

- To return more than one value from a function
- To pass arrays and strings more conveniently from one function to another
- To manipulate arrays easily by moving pointers to them (or to parts of them), instead of moving the arrays themselves
- To allocate memory and access it (dynamic memory allocation)

13.2 Pointer Variables

If a variable is to be used as a pointer, it must be declared. A pointer declaration consists of a base type, an `*`, and a variable name. The general syntax for declaring a pointer variable is:

```
type *name;
```

where `type` is any valid data type, and `name` is the name of the pointer variable. The declaration tells the compiler that `name` is used to store the address of a value corresponding to the data type `type`. In the declaration statement, `*` indicates that a pointer variable is being declared.

In the above example of **var1** and **var2**, since **var2** is a pointer, which holds the value of an `int` type variable **var1**, it will be declared as:

```
int *var2;
```

Now, **var2** can be used in a program to indirectly access the value of **var1**. Remember, **var2** is not of type `int` but is a pointer to a variable of type `int`.

The base type of the pointer defines what type of variables the pointer can point to. Technically, any type of pointer can point anywhere in the memory. However, all pointer arithmetic is done relative to its base type, so it is important to declare the pointer correctly.

13.3 The Pointer Operators

There are two special operators which are used with pointers: `*` and `&`. The `&` operator is a unary operator and it returns the memory address of the operand. For example,

```
var2 = &var1;
```

places the memory address of **var1** into **var2**. This address is the computer's internal location of the variable **var1** and has nothing to do with the value of **var1**. The `&` operator can be thought of as returning "the address of". Therefore, the above assignment means "**var2** receives the address of **var1**". Referring back, the value of **var1** is `500` and it uses the memory location `1000` to store this value. After the above assignment, **var2** will have the value `1000`.

The second pointer operator, `*`, is the complement of `&`. It is a unary operator and returns the value contained in the memory location pointed to by the pointer variable's value.

Consider the previous example, where **var1** has the value `500` and is stored in the memory location `1000`, after the statement

```
var2 = &var1;
```

var1 contains the value `1000`, and after the assignment

```
temp = *var2;
```

temp will contain `500` and not `1000`. The `*` operator can be thought of as "at the address".

Both `*` and `&` have a higher precedence than all other arithmetic operators except the unary minus. They share the same precedence as the unary minus.

The following program prints the value of an integer variable, its address, which is stored in a pointer variable, and also the address of the pointer variable.

Concepts

Example 1:

```
#include <stdio.h>
void main()
{
    int var = 500, *ptr_var;
    /* var is declared as an integer and ptr_var as a pointer pointing to an
integer */
    ptr_var = &var; /*stores address of var in ptr_var*/
    /*Prints value of variable (var) and address where var is stored */
    printf("The value %d is stored at address %u:",var,&var);
    /*Prints value stored in ptr variable (ptr_var) and address where ptr_var
is stored */
    printf("\nThe value %u is stored at address: %u", ptr_var, &ptr_var);
    /* Prints value of variable (var) and address where var is stored, using
pointer to variable */
    printf("\nThe value %d is stored at address:%u", *ptr_var, ptr_var);
}
```

A sample output for the above will be:

```
The value 500 is stored at address: 65500
The value 65500 is stored at address: 65502
The value 500 is stored at address: 65500
```

In the above, **`ptr_var`** contains the address 65500, which is a memory location where the value of **`var`** is stored. The contents of this memory location (65500) can be obtained by using `*`, as **`*ptr_var`**. Now **`*ptr_var`** represents the value 500, which is the value of **`var`**. Since **`ptr_var`** is also a variable, its address can be printed by prefixing it with `&`. In the above case, **`ptr_var`** is stored at location 65502. The `%u` conversion specifier prints the arguments as unsigned integers.

Recollect that an integer occupies 2 bytes of memory. Hence, value of **`var`** is stored at 65500 and the compiler allots the next memory allocation 65502 to **`ptr_var`**. Similarly, a floating point number will require four bytes and a double precision number may require eight bytes. Pointer variables store an integer value. For most programs using pointers, pointer types can be considered to be 16-bit values that occupy 2 bytes.

Note that the following two statements give the same output.

```
printf("The value is %d", var);
printf("The value is %d", *(&var));
```

Assigning Values to Pointers

Values can be assigned to pointers through the `&` operator. The assignment statement will be:

```
ptr_var = &var;
```

where the address of **`var`** is stored in the variable **`ptr_var`**. It is also possible to assign values to pointers through another pointer variable pointing to a data item of the same type.

```
ptr_var = &var;
ptr_var2 = ptr_var;
```

A NULL value can also be assigned to a pointer using zero as follows:

```
ptr_var = 0;
```

Variables can be assigned value through their pointers as well.

```
*ptr_var = 10;
```

will assign `10` to the variable **`var`** if **`ptr_var`** points to **`var`**.

In general, expressions involving pointers follow the same rules as other C expressions. It is very important to assign values to pointer variables before using them; else they could be pointing to any unpredictable values.

Pointer Arithmetic

Addition and subtraction are the only operations, which can be performed on pointers. The following example demonstrates this:

```
int var, *ptr_var;
ptr_var = &var;
var = 500;
```

In the above example, let us assume that **`var`** is stored at the address `1000`. Then, **`ptr_var`** has the value `1000` stored in it. Since integers are 2 bytes long, after the expression:

```
ptr_var++ ;
```

`ptr_var` will contain `1002` and NOT `1001`. This means that **`ptr_var`** is now pointing to the integer stored at the address `1002`. Each time **`ptr_var`** is incremented, it will point to the next integer and since

integers are 2 bytes long, **ptr_var** will be incremented by 2. The same is true for decrements also.

Here are few more examples.

`++ptr_var or ptr_var++`	points to next integer after **var**
`--ptr_var or ptr_var--`	points to integer previous to **var**
`ptr_var + i`	points to the ith integer after **var**
`ptr_var - i`	points to the ith integer before **var**
`++*ptr_var or (*ptr_var)++`	Will increment **var** by 1
`*ptr_var++`	Will fetch the value of the next integer after **var**

Each time a pointer is incremented, it points to the memory location of the next element of its base type. Each time it is decremented, it points to the location of the previous element. With pointers to characters, this appears normal, because generally characters occupy 1 byte per character. However, all other pointers will increase or decrease depending on the length of the data type they are pointing to.

As seen in the above examples, in addition to the increment and decrement operators, integers can be added and subtracted to or from pointers. Besides addition and subtraction of a pointer and an integer, none of the other arithmetic operations can be performed on pointers. To be specific, pointers cannot be multiplied or divided. Also, float or double type cannot be added or subtracted to or from pointers.

Pointer Comparisons

Two pointers can be compared in a relational expression. However, this is possible only if both these variables are pointing to variables of the same type. Consider that **ptr_a** and **ptr_b** are two pointer variables, which point to data elements **a** and **b**. In this case, the following comparisons are possible:

`ptr_a < ptr_b`	Returns true provided **a** is stored before **b**
`ptr_a > ptr_b`	Returns true provided **a** is stored after **b**
`ptr_a <= ptr_b`	Returns true provided **a** is stored before **b** or **ptr_a** and **ptr_b** point to the same location
`ptr_a >= ptr_b`	Returns true provided **a** is stored after **b** or **ptr_a** and **ptr_b** point to the same location
`ptr_a == ptr_b`	Returns true provided both pointers **ptr_a** and **ptr_b** points to the same data element
`ptr_a != ptr_b`	Returns true provided both pointers **ptr_a** and **ptr_b** point to different data elements but of the same type
`ptr_a == NULL`	Returns true if **ptr_a** is assigned NULL value (zero)

Also, if **ptr_begin** and **ptr_end** point to members of the same array then,

```
ptr_end - ptr_begin
```

will give the difference in bytes between the storage locations to which they point.

13.4 Pointers and Single-dimensional Arrays

An array name is truly a pointer to the first element in that array. Therefore, if **ary** is a single-dimensional array, the address of the first array element can be expressed as either **&ary[0]** or simply as **ary**. Similarly, the address of the second array element can be written as **&ary[1]** or as **ary+1**, and so on. In general, the address of the (i + 1)th array element can be expressed as either **& ary[i]** or as **(ary + i)**. Thus, the address of an array element can be expressed in two ways:

- By writing the actual array element preceded by the ampersand sign (&)
- By writing an expression in which the subscript is added to the array name

Remember that in the expression **(ary + i), ary** represents an address, whereas **i** represents an integer quantity. Moreover, **ary** is the name of an array whose elements can be both integers, characters, floating point, and so on (of course, all elements have to be of the same type). Therefore, the above expression is not a mere addition; it is actually specifying an address, which is a certain number of memory cells beyond the first. The expression **(ary + i)** is in true sense, a symbolic representation for an address specification rather than an arithmetic expression.

As said before, the number of memory cells associated with an array element will depend on the data type of the array as well as the computer's architecture. However, the programmer can specify only the address of the first array element that is the name of the array (**ary** in this case) and the number of array elements beyond the first, that is, a value for the subscript. The value of **i** is sometimes referred to as an **offset** when used in this manner.

The expressions **&ary[i]** and **(ary + i)** both represent the address of the ith element of ary, and so it is only logical that **ary[i]** and ***(ary + i)** both represent the contents of that address, that is, the value of the **i**th element of **ary**. Both terms are interchangeable and can be used in any particular application as desired by the programmer.

The following program shows the relationship between array elements and their addresses.

Example 2:

```
#include<stdio.h>
void main()
{
    static int ary[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int i;
    for (i = 0; i < 10; i ++)
    {
```

Concepts

```
        printf("\n i = %d , ary[i] = %d , *(ary+i)= %d ", i, ary[i], *(ary
+ i));
        printf("&ary[i] = %X , ary + i = %X", &ary[i], ary+i);
        /* %X gives unsigned hexadecimal */
    }
}
```

The above program defines a single dimensional, 10-element integer array **ary**, whose elements are assigned the values `1`, `2`, ..`10`. The `for` loop is used to display the value and the corresponding address of each array element. Note that the value of each element is specified in two different ways, as **`ary[i]`** and as **`*(ary + i)`**, to illustrate their equivalence. Similarly, the address of each array element is also displayed in two ways. The output of the program will be as follows:

```
i=0 ary[i]=1  *(ary+i)=1  &ary[i]=194  ary+i = 194
i=1 ary[i]=2  *(ary+i)=2  &ary[i]=196  ary+i = 196
i=2 ary[i]=3  *(ary+i)=3  &ary[i]=198  ary+i = 198
i=3 ary[i]=4  *(ary+i)=4  &ary[i]=19A  ary+i = 19A
i=4 ary[i]=5  *(ary+i)=5  &ary[i]=19C  ary+i = 19C
i=5 ary[i]=6  *(ary+i)=6  &ary[i]=19E  ary+i = 19E
i=6 ary[i]=7  *(ary+i)=7  &ary[i]=1A0  ary+i = 1A0
i=7 ary[i]=8  *(ary+i)=8  &ary[i]=1A2  ary+i = 1A2
i=8 ary[i]=9  *(ary+i)=9  &ary[i]=1A4  ary+i = 1A4
i=9 ary[i]=10 *(ary+i)=10 &ary[i]=1A6  ary+i = 1A6
```

This output clearly shows the difference between **`ary[i]`**, which represents the value of the **`i`**th array element, and **`&ary[i]`**, which represents its address.

When assigning a value to an array element such as **`ary[i]`**, the left side of the assignment statement can be written as either **`ary[i]`** or as **`*(ary + i)`**. Thus, a value may be assigned directly to an array element or it may be assigned to the memory area whose address is that of the array element. It is sometimes necessary to assign an address to an identifier. In such situations, a pointer must appear on the left side of the assignment statement. It is not possible to assign an arbitrary address to an array name or to an array element. Thus, expressions such as **`ary, (ary + i)`** and **`&ary[i]`** cannot appear on the left side of an assignment statement. Moreover, the address of an array cannot be arbitrarily altered, so expressions such as **`ary++`** are not allowed. The reason for this is: **ary** is the address of the array **`ary`**. When the array is declared, the linker has decided where this array will go, for example, say an address `1002`. Once it is given this address, it stays there. Trying to increment this address has no meaning, its like saying

```
x = 5++;
```

Since a constant cannot be incremented, the compiler will flag an error.

In case of the array **ary**, **ary** is also known as a **Pointer Constant**. Remember, **(ary + 1)** does not move the array **ary** to the **(ary + 1)**th position, it just points to that position, whereas **ary++** actually tries to move **ary** by 1 position.

The address of one element cannot be assigned to some other array element, though the value of one array element can be assigned to another through pointers.

```
&ary[2] =  &ary[3];   /* not allowed */
ary[2]  =  ary[3];    /* allowed */
```

Recall that the `scanf()` function required that variables of the basic data types be preceded by ampersands (`&`), whereas array names were exempted from this requirement. This will be easy to understand now. The `scanf()` requires that the address of the data items, being entered into the computer's memory, be specified. As said before, the ampersand (`&`) actually gives the address of the variable and so it is required that an ampersand precede a single valued variable. Ampersands are not required with array names because array names themselves represent addresses. However, if a single element of an array is to be read, it will require an ampersand to precede it.

```
scanf("d", *ary) /* For first element of array */
scanf("%d", &ary[2]) /* For an array element */
```

13.4.1 Pointers and Multidimensional Arrays

The way a single dimensional array can be represented in terms of a pointer (the array name) and an offset (the subscript), a multidimensional array can also be represented with an equivalent pointer notation. This is because a multidimensional array is actually a collection of single dimensional arrays. For example, a two-dimensional array can be defined as a pointer to a group of contiguous one-dimensional arrays. A two-dimensional array declaration can be written as:

```
data_type (*ptr_var)[expr 2];
```

instead of

```
data_type array[expr 1][expr 2];
```

This concept can be generalized to higher dimensional arrays, that is,

```
data_type (*ptr_var)[exp 2] .... [exp N];
```

can be written instead of

```
data_type array[exp 1][exp 2] ... [exp N];
```

In these declarations, **data_type** refers to the data type of the array, **ptr_var** is the name of the pointer variable, array is the corresponding array name, and **exp 1**, **exp 2**, **exp 3**, ... **exp N** are positive valued integer expressions that indicate the maximum number of array elements associated with each subscript.

Note the parentheses that surround the array name and the preceding asterisk in the pointer version of each declaration. These parentheses must be present; else the definition would represent an array of pointers rather than a pointer to a group of arrays.

For example, if **ary** is a two dimensional array having 10 rows and 20 columns, it can be declared as

```
int (*ary)[20];
```

instead of

```
int ary[10][20];
```

In the first definition, **ary** is defined to be a pointer to a group of contiguous, single-dimensional, 20-element integer arrays. Thus, **ary** points to the first element of the array, which is actually the first row (row 0) of the original two-dimensional array. Similarly, **(ary + 1)** points to the second row of the original two-dimensional array, and so on.

A three-dimensional floating-point array **fl_ary** can be defined as:

```
float (*fl_ary)[20][30];
```

rather than

```
float fl_ary[10][20][30];
```

In the first declaration, **fl_ary** is defined as a group of contiguous, two-dimensional, 20 x 30 floating point arrays. Hence, **fl_ary** points to the first 20 x 30 array, **(fl_ary + 1)** points to the second 20 x 30 array, and so on.

In the two-dimensional array **ary**, the item in row 4 and column 9 can be accessed using the statement:

```
ary[3][8];
```

or

```
*(*(ary + 3) + 8);
```

The first form is the usual way in which an array is referred to. In the second form, **(ary + 3)** is a pointer

to the row 4. Therefore, the object of this pointer, **`*(ary + 3)`**, refers to the entire row. Since row 3 is a one-dimensional array, **`*(ary + 3)`** is actually a pointer to the first element in row 3, `8` is then added to this pointer. Hence, **`*(*(ary + 3) + 8)`** is a pointer to element 8 (the 9th element) in row 4. The object of this pointer, **`*(*(ary + 3) + 8)`**, therefore refers to the item in column 9 of row 4, which is **`ary [3][8]`**.

There are different ways to define arrays, and different ways to process the individual array elements. The choice of one method over another generally depends on the user's preference. However, in applications involving numerical arrays, it is often easier to define the arrays in the conventional manner.

Pointers and Strings

Strings are nothing but single-dimensional arrays, and as arrays and pointers are closely related, it is only natural that strings too will be closely related to pointers. Consider the case of the function `strchr()`. This function takes as arguments a string and a character to be searched for in that string, that is,

```
ptr_str = strchr(strl, 'a');
```

the pointer variable **`ptr_str`** will be assigned the address of the first occurrence of the character '`a`' in the string **`str`**. This is not the position in the string, from 0 to the end of the string but the address, from where the string starts to the end of the string.

The following program uses `strchr()` in a program which allows a user to enter a string and a character to be searched for. The program prints out the address of the start of the string, the address of the character, and the character's position relative to the start of the string (0 if it is the first character, 1 if it is the second and so on). This relative position is the difference between the two addresses, the address of start of the string and the address where the character's first occurrence is found.

Example 3:

```
#include <stdio.h>
#include <string.h>
void main ()
{
    char a, str[81], *ptr;
    printf("\nEnter a sentence:");
    gets(str);
    printf("\nEnter character to search for:");
    a = getche();
    ptr = strchr(str,a);
    /* return pointer to char */
    printf( "\nString starts at address: %u",str);
    printf("\nFirst occurrence of the character is at address: %u", ptr);
```

Concepts

```
    printf("\n Position of first occurrence (starting from 0)is: %d", ptr-
str);
}
```

A sample run will be as follows:

```
Enter a sentence: We all live in a yellow submarine
Enter character to search for: Y
String starts at address: 65420.
First occurrence of the character is at address: 65437.
Position of first occurrence (starting from 0) is: 17
```

In the declaration statement, a pointer variable **ptr** is set aside to hold the address returned by `strchr()`, since this is an address of a character (**ptr** is of type `char`).

The function `strchr()` does not need to be declared if the include file `string.h` is included.

13.5 Allocating Memory

Till this point of time it has been established that an array name is actually a pointer to the first element of the array. Also, it is possible to define the array as a pointer variable rather than the conventional array. However, if an array is declared conventionally, it results in a fixed block of memory being reserved at the beginning of the program execution, whereas this does not occur if the array is represented as a pointer variable. As a result, the use of a pointer variable to represent an array requires some sort of initial memory assignment before the array elements are processed. Such memory allocations are generally done using the `malloc()` library function.

Consider an example. A single dimensional integer array **ary** having 20 elements can be defined as:

```
int *ary;
```

instead of

```
int ary[20];
```

However, **ary** will not be automatically assigned a memory block when it is defined as a pointer variable, though a block of memory enough to store 10 integer quantities will be reserved in advance if **ary** is defined as an array. If **ary** is defined as a pointer, sufficient memory can be assigned as follows:

```
ary = malloc(20 *sizeof(int));
```

This will reserve a block of memory whose size (in bytes) is equivalent to the size of an integer. Here,

a block of memory for 20 integers is allocated. The number 20 assigns 20 bytes (one for each integer) and this is multiplied by `sizeof(int)`, which will return 2, if the computer uses 2 bytes to store an integer. If a computer uses 1 byte to store an integer, the `sizeof()` function is not required. However, it is preferable to use this always as it facilitates the portability of code. The function `malloc()` returns a pointer which is the address location of the starting point of the memory allocated. If enough memory space does not exist, `malloc()` returns a NULL. The allocation of memory in this manner, that is, **as and when required in a program** is known as **Dynamic memory allocation.**

Before proceeding further, let us discuss the concept of **Dynamic Memory allocation**. A C program can store information in the main memory of the computer in two primary ways. The first method involves global and local variables – including arrays. In the case of global and static variables, the storage is fixed throughout the program's run time. These variables require that the programmer knows the amount of memory needed for every situation in advance. The second way in which information can be stored is through C's **Dynamic Allocation System**. In this method, storage for information is allocated from the pool of free memory as and when needed.

The `malloc()` function is one of the most commonly used functions which permit allocation of memory from the pool of free memory. The parameter for `malloc()` is an integer that specifies the number of bytes needed.

As another example, consider a two-dimensional character array **`ch_ary`** having 10 rows and 20 columns. The definition and allocation of memory in this case would be as follows:

```
char (*ch_ary)[20];
ch_ary = (char*)malloc(10*20*sizeof(char));
```

As said earlier, `malloc()` returns a pointer to type void. However, since **`ch_ary`** is a pointer to type `char`, type casting is necessary. In the above statement, `(char*)` casts `malloc()` so as to return a pointer to type `char`.

However, if the declaration of array has to include the assignment of initial values then an array has to be defined in the conventional manner rather than as a pointer variable as in:

```
int ary[10] = {1,2,3,4,5,6,7,8,9,10};
```

or

```
int ary[] = {1,2,3,4,5,6,7,8,9,10};
```

The following example creates a single dimensional array dynamically and sorts the array in ascending order. It uses pointers and the `malloc()` function to assign memory.

Concepts

Example 4:

```
#include<stdio.h>
#include<malloc.h>
void main()
{
    int *p,n,i,j,temp;
    printf("\n Enter number of elements in the array :");
    scanf("%d",&n);
    p=(int*)malloc(n*sizeof(int));
    for(i=0;i<n;++i)
    {
        printf("\nEnter element no. %d:",i+1);
        scanf("%d",p+i);
    }
    for(i=0;i<n-1;++i)
        for(j=i+1;j<n;++j)
            if(*(p+i)>*(p+j))
            {
                temp=*(p+i);
                *(p+i)=*(p+j);
                *(p+j)=temp;
            }
    for(i=0;i<n;++i)
        printf("%d\n",*(p+i));
}
```

Note the `malloc()` statement,

```
p = (int*)malloc(n*sizeof(int));
```

Here, `p` is declared as a pointer to an array and assigned an amount of memory using `malloc()`.

Data is read, using `scanf()`.

```
scanf("%d",p+i);
```

In `scanf()`, the pointer variable is used to store data into the array.

Sorted array elements are displayed using `printf()`.

```
printf("%d\n",*(p+i));
```

Note the asterisk in this case. This is because the value stored in that particular location has to be displayed. Without the asterisk, the `printf()` will display the address where the marks are stored and not the marks stored.

- **free()**

This function can be used to de-allocate (frees) memory when it is no longer needed.

The general format of `free()` function:

```
voidfree( void *ptr );
```

The `free()` function de-allocates the space pointed to by **`ptr`**, freeing it up for future use. **`ptr`** must have been used in a previous call to `malloc()`, `calloc()`, or `realloc()`. `calloc()` and `realloc()` have been discussed later.

The example given below will ask you how many integers you'd like to store in an array. It'll then allocate the memory dynamically using `malloc()` and store a certain number of integers, print them out, then releases the used memory using **free**.

Example 5:

```
#include <stdio.h>
#include <stdlib.h> /* required for the malloc and free functions */
int main()
{
    int number;
    int *ptr;
    int i;
    printf("How many ints would you like store? ");
    scanf("%d", &number);
    ptr = (int *) malloc (number*sizeof(int)); /* allocate memory */
    if(ptr!=NULL)
    {
        for(i=0 ; i<number ; i++)
        {
            *(ptr+i) = i;
        }
```

```
        for(i=number ; i>0 ; i--)
        {
             printf("%d\n",*(ptr+(i-1))); /* print out in reverse order
*/
        }
        free(ptr); /* free allocated memory */
        return 0;
    }
    else
    {
        printf("\nMemory allocation failed - not enough memory.\n");
        return 1;
    }
}
```

Output if entered 3:

```
How many ints would you like store? 3
2
1
0
```

- **calloc()**

`calloc` is similar to `malloc`, but the main difference is that the values stored in the allocated memory space is zero by default. With `malloc`, the allocated memory could have any value.

`calloc` requires two arguments. The first is the number of variables you'd like to allocate memory for. The second is the size of each variable.

```
void *calloc( size_t num, size_t size );
```

Like `malloc`, `calloc` will return a void pointer if the memory allocation was successful, else it will return a NULL pointer.

The example given below shows you how to call `calloc` and reference the allocated memory using an array index. The initial value of the allocated memory is printed out in the `for` loop.

Example 6:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    float *calloc1, *calloc2;
    int i;
    calloc1 = (float *) calloc(3, sizeof(float));
    calloc2 = (float *)calloc(3, sizeof(float));
    if(calloc1!=NULL && calloc2!=NULL)
    {
        for(i=0 ; i<3 ; i++)
        {
            printf(”calloc1[%d] holds %05.5f ”, i, calloc1[i]);
            printf(”\ncalloc2[%d] holds %05.5f ”, i, *(calloc2+i));
        }
        free(calloc1);
        free(calloc2);
        return 0;
    }
    else
    {
        printf(”Not enough memory\n”);
        return 1;
    }
}
```

Output:

```
calloc1[0]   holds   0.00000
calloc2[0]   holds   0.00000
calloc1[1]   holds   0.00000
calloc2[1]   holds   0.00000
calloc1[2]   holds   0.00000
calloc2[2]   holds   0.00000
```

On all machines, the `calloc1` and `calloc2` arrays should hold zeros. `calloc` is especially useful when you’re using multi-dimensional arrays. Here is another example to demonstrate the use of `calloc()` function.

Concepts

Example 7:

```
/* This program gets the number of elements, allocates spaces for the
elements, gets a value for each element, sum the values of the elements,
and print the number of the elements and the sum.
*/
#include <stdio.h>
#include <stdlib.h>
main()
{
    int *a, i, n, sum = 0;
    printf ( "\n%s%s", "An array will be created dynamically. \n\n",
"Input an array size n followed by integers : " );
    scanf( "%d", &n); /* get the number of elements */
    a = (int *) calloc (n, sizeof(int) ); /* allocate space */
    /* get a value for each element */
    for( i = 0; i < n; i++ )
    {
        printf("Enter %d values : ",n);
        scanf( "%d", a + i );
    }
    /* sum the values */
    for(i = 0; i < n; i++ )
        sum += a[i];
    free(a); /* free the space */
    /* print the number and the sum */
    printf ( "\n%s%7d\n%s%7d\n\n", "Number of elements: ", n, "Sum of
the elements: ", sum );
}
```

- **realloc()**

Suppose you've allocated a certain number of bytes for an array but later find that you want to add values to it. You could copy everything into a larger array, which is inefficient, or you can allocate more bytes using `realloc`, without losing your data.

`realloc()` takes two arguments. The first is the pointer referencing the memory. The second is the total number of bytes you want to reallocate.

```
void *realloc( void *ptr, size_t size );
```

Passing zero as the second argument is the equivalent of calling **free**.

Once again, `realloc` returns a void pointer if successful, else a NULL pointer is returned.

This example uses `calloc` to allocate enough memory for an `int` array of five elements. Then `realloc` is called to extend the array to hold seven elements.

Example 8:

```
#include<stdio.h>
#include <stdlib.h>
int main()
{
    int *ptr;
    int i;
    ptr = (int *)calloc(5, sizeof(int *));
    if(ptr!=NULL)
    {
        *ptr = 1;
        *(ptr+1) = 2;
        ptr[2] = 4;
        ptr[3] = 8;
        ptr[4] = 16;
        /* ptr[5] = 32; wouldn't assign anything */
        ptr = (int *)realloc(ptr, 7*sizeof(int));
        if(ptr!=NULL)
        {
            printf("Now allocating more memory... \n");
            ptr[5] = 32; /* now it's legal! */
            ptr[6] = 64;
            for(i=0;i<7;i++)
            {
                printf("ptr[%d] holds %d\n", i, ptr[i]);
            }
            realloc(ptr,0); /* same as free(ptr); - just fancier! */
            return 0;
        }
        else
        {
```

```
            printf("Not enough memory - realloc failed.\n");
            return 1;
        }
    }
    else
    {
        printf("Not enough memory - calloc failed.\n");
        return 1;
    }
}
```

Output :

```
Now allocating more memory...

ptr[0] holds  1
ptr[1] holds  2
ptr[2] holds  4
ptr[3] holds  8
ptr[4] holds  16
ptr[5] holds  32
ptr[6] holds  64
```

Notice the two different methods that used when initializing the `array:` `ptr[2] = 4`; is the equivalent to `*(ptr+2) = 4;` (just easier to read!).

Before using **`realloc`**, assigning a value to **`ptr[5]`** wouldn't cause a compile error. The program would still run, but **`ptr[5]`** wouldn't hold the value you assigned.

Summary

- A pointer provides a way of accessing a variable without referring to the variable directly.
- A pointer is a variable, which contains the address of a memory location of another variable, rather than its stored value.
- A pointer declaration consists of a base type, an *, and the variable name.
- There are two special operators which are used with pointers: * and &.
- The & operator returns the memory address of the operand.
- The second operator, *, is the complement of &. It returns the value contained in the memory location pointed to by the pointer variable's value.
- Addition and subtraction are the only operations, which can be performed on pointers.
- Two pointers can be compared in a relational expression only if both these variables are pointing to variable(s) of the same type.
- Pointers are passed to a function as arguments, enabling data items within the called routine of the program to access variables whose scope does not extend beyond the calling function.
- An array name is truly a pointer to the first element in that array.
- A pointer constant is an address; a pointer variable is a place to store addresses.
- Memory can be allocated as and when needed by using the malloc(), calloc(), and realloc() functions. Allocating memory in this way is known as Dynamic Memory Allocation.

Check Your Progress

1. A _____ provides a way of accessing a variable without referring to the variable directly.

   A. Array B. Pointer

   C. Structure D. None of the above

2. Pointers cannot point to arrays. (T/F)

3. The _____ of the pointer defines what type of variables the pointer can point to.

   A. Type B. Size

   C. Content D. None of the above

4. The two special operators used with pointers are ____ and _____.

   A. ^ and % B. ; and ?

   C. * and & D. None of the above

5. _____ and _____ are the only operations, which can be performed on pointers.

   A. Addition, Subtraction B. Multiplication, Division

   C. Division, Addition D. None of the above

6. Two pointers can be compared only if both these variables are pointing to variables of different types. **(T/F)**

7. The allocation of memory in this manner, that is, as and when required in a program is known as _____ .

   A. Dynamic Memory Allocation B. Static Memory Allocation

   C. Content Memory Allocation D. None of the above

Try It Yourself

1. Write a program to accept a string and find out if it is a palindrome.

2. Write a program using pointer to strings that accepts the name of an animal and a bird and returns the names in plural.

“Who dares to teach must never cease to learn”

Session

14 Pointers (Lab)

Objectives

At the end of this session, you will be able to:

- *Use Pointers*
- *Use Pointers with Arrays*

The steps given in this session are detailed comprehensive and carefully thought through. This has been done so that the learning objectives are met and the understanding of the tool is complete. Please follow the steps carefully.

Part I – For the first 1 Hour and 30 Minutes:

14.1 Pointers

Pointer variables in C can hold the address of a variable of any basic data type. That is, pointers can be of integer or char data type. An integer pointer variable will hold the address of an integer variable. A character pointer will hold the address of a character variable.

14.1.1 Counting the number of vowels in a string using pointers

Pointers can be used instead of subscripts to traverse through elements in an array. For example, a string pointer can be used to point to the starting memory address of a word. Hence such a pointer could be used to read the characters in that word. To demonstrate this, let us write a C program to count the number of vowels in a word using pointers. The steps are listed below:

1. Declare a character pointer variable.

The code will be,

```
char *ptr;
```

2. Declare a character array and accept value for the same.

The code will be,

```
char word[10];
printf("\n Enter a word : ");
scanf("%s", word);
```

3. Assign the character pointer to the string.

The code will be,

```
ptr = &word[0];
```

The address of the first character of the character array, word, will be stored in the pointer variable, ptr. In other words, the pointer ptr will be point to the first character in the character array word.

4. Traverse through the characters in the word to find out whether they are vowels or not. In case a vowel is found, increment the count of vowels.

The code for the same is,

```
int i, vowcnt;
for(i=0;i<strlen(word);i++)
{
    if((*ptr=='a')||(*ptr=='e')||(*ptr=='i')||(*ptr=='o')||(*ptr=='u')|
|(*ptr=='A')||(*ptr=='E')|| (*ptr =='I')||(*ptr=='O')||(*ptr=='U'))
        vowcnt++;
    ptr++;
}
```

5. Display the word and the number of vowels in the word.

The code for the same will be,

```
printf("\n The word is : %s \n The number of vowels in the word is :
%d ", word,vowcnt);
```

Let us look at the complete program.

1. **Invoke the editor in which you can type the C program.**
2. **Create a new file.**
3. **Type the following code :**

```
void main()
{
    char *ptr;
    char word[10];
    int i, vowcnt=0;
    printf("\n Enter a word : ");
    scanf("%s",word);
    ptr = &word[0];
    for(i=0;i<strlen(word);i++)
    {
        if((*ptr=='a')||(*ptr=='e')||(*ptr=='i')||(*ptr=='o')||
        (*ptr=='u')|| (*ptr=='A')||(*ptr=='E')||(*ptr=='I')||
        (*ptr=='O')||(*ptr=='U'))
            vowcnt++;
        ptr++;
    }
    printf("\n The word is : %s \n The number of vowels in the word
is : %d ", word,vowcnt);
}
```

To see the output, follow these steps:

4. **Save the file with the name `pointerI.C`.**
5. **Compile the file, `pointerI.C`.**
6. **Execute the program, `pointerI.C`.**
7. **Return to the editor.**

The sample output of the above program will be as shown in Figure 14.1.

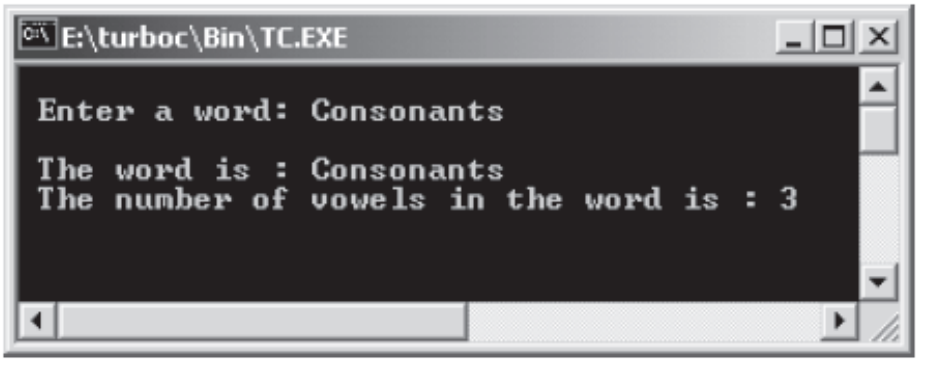


Figure 14.1: Output of pointerl.C

14.1.2 Sort an array in alphabetical order using pointers

Pointers could be used to swap the contents of two memory addresses. To demonstrate this, let us write a C program to sort a set of strings in alphabetical order.

There are many ways of solving this program. Let us use an array of character pointers to understand the use of array of pointers.

To do this programmatically,

1. Declare an array of character pointers to hold 5 strings.

The code for the same is,

```
char *ptr[5];
```

The array looks as depicted in figure 14.2.

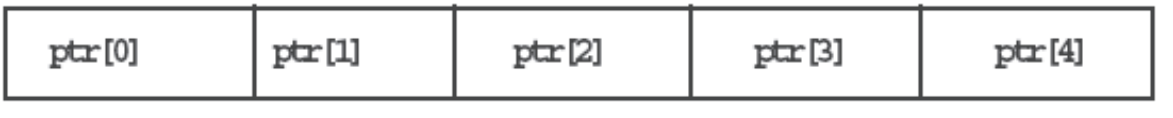


Figure 14.2: Character Pointer Array

2. Accept 5 strings and assign the pointers in the pointer array to the strings.

The code is,

```
int i;
char cpyptr1[5][10];
for (i=0;i<5;i++)
{
    printf("\n Enter a string : ");
    scanf("%s",cpyptr1[i]);
    ptr[i]=cpyptr1[i];
}
```

3. Preserve the array of strings before sorting.

To do this, we need to create a copy of the array of strings. The code for the same will be,

```
char cpyptr2[5][10];
for (i=0;i<5;i++)
    strcpy(cpyptr2[i],cpyptr1ptr[i]);
```

Here, the `strcpy()` function is used to copy the strings into another array.

4. Sort the array of strings in alphabetical order.

The code is,

```
char *temp;
for(i=0;i<4;i++)
{
    for(j=i+1;j<5;i++)
    {
        if (strcmp(ptr[i],ptr[j])>0)
        {
            temp=ptr[i];
            ptr[i]=ptr[j];
            ptr[j]=temp;
        }
    }
}
```

Lab Guide

5. Display the original and the sorted strings.

The code for the same will be,

```
print("\n The Original list is ");
for(i=0;i<5;i++)
    printf("\n%s",cpyptr2[i]);
printf("\n The Sorted list is ");
for(i=0;i<5;i++)
    printf("\n%s",ptr[i]);
```

Let us look at the complete program.

1. Create a new file.

2. Type the following code :

```
void main()
{
    char *ptr[5];
    int i;
    int j;
    char cpyptr1[5][10],cpyptr2[5][10];
    char *temp;
    for (i=0;i<5;i++)
    {
        printf("\n Enter a string : ");
        scanf("%s",cpyptr1[i]);
        ptr[i]=cpyptr1[i];
    }
    for (i=0;i<5;i++)
        strcpy(cpyptr2[i],cpyptr1[i]);
    for(i=0;i<4;i++)
    {
        for(j=i+1;j<5;j++)
        {
            if (strcmp(ptr[i],ptr[j])>0)
            {
                temp=ptr[i];
```

```
                ptr[i]=ptr[j];
                ptr[j]=temp;
            }
        }
    }
    printf("\n The Original list is ");
    for(i=0;i<5;i++)
        printf("\n%s",cpyptr2[i]);
    printf("\n The Sorted list is ");
    for(i=0;i<5;i++)
        printf("\n%s",ptr[i]);
}
```

To see the output, follow these steps:

3. **Save the file with the name `pointII.C`.**

4. **Compile the file, `pointII.C`.**

5. **Execute the program, `pointII.C`.**

6. **Return to the editor.**

The sample output of the above program will be as shown in Figure 14.3.

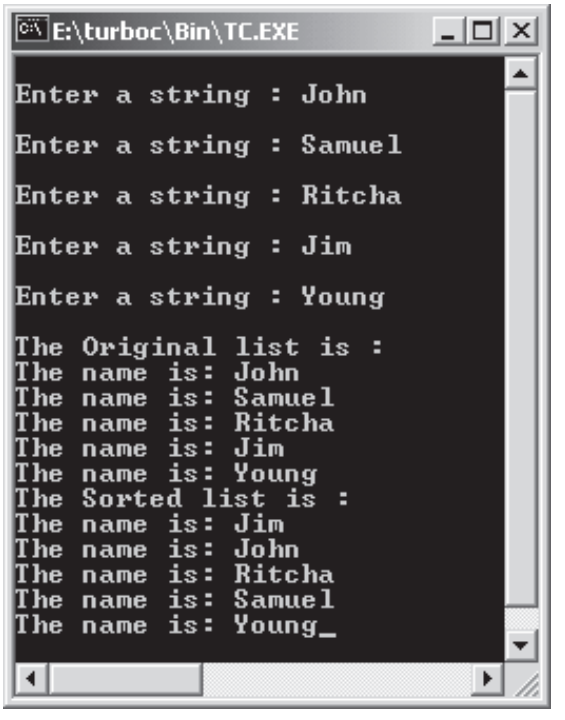


Figure 14.3: Output of pointII.C

Part II – For the next 30 Minutes:

1. Write a C Program to concatenate two strings using pointers.

   To do this,

   a. Declare three string variables

   b. Declare three character pointers

   c. Accept the values of two strings

   d. Make the three pointers to point to the three strings variables respectively. The third string will not have any value right now

   e. Loop through the first string and copy the contents of that string to the third string. Use the pointer variables to copy the values

   f. After copying the first string, loop through the second string and copy the contents of that string to the end of the third string. Use the pointer variables to copy the values

   g. Print the three strings

Try It Yourself

1. Write a C Program to reverse a character array using pointers.

2. Write a C Program to add two matrices using pointers.

Session

15 Functions

Concepts

Objectives

At the end of this session, you will be able to:

- *Explain the use of functions*
- *Explain the structure of a function*
- *Explain function declaration and function prototypes*
- *Explain the different types of variables*
- *Explain how to call functions*
  - *Call by Value*
  - *Call by Reference*
- *Explain the scope rules for a function*
- *Explain functions in multifile programs*
- *Explain Storage classes*
- *Explain function pointers*

Introduction

A function is a self-contained program segment that carries out a specific, well-defined task. They are actually the smaller segments, which help solve the larger problem.

15.1 The use of Functions

Functions are generally used in **C** to execute a series of instructions. However, usage of functions is not similar to that of loops. Loops can repeat a series of instructions only if every iteration immediately follows the previous one. But calling a function causes a series of instructions to be executed at any given point within a program. The functions can be called as many times as required. Suppose a section of code in a program calculates the percentage of some given numbers. If later, in the same program, the same calculation has to be done using different numbers, instead of writing the same instructions all over again, a function can be written to calculate the percentage of any given numbers. The program can then jump

Concepts

to that function, perform the calculations (in the function) and jump back to the place from where it was called. This will become clear as the discussion proceeds.

Another important aspect is that functions are easier to write and understand. Simple functions can be written to do specific tasks. Also, debugging the program is easier as the program structure is more readable, due to its simplified form. Each function can individually be tested for all possible inputs, for valid as well as invalid data. Programs containing functions are also easier to maintain, because modifications, if required, can be restricted to certain functions within the program. Not only can a function be called from different points in a program, but also they can be put in a library of related functions and used by many programs, thus saving on coding time.

15.2 The Function Structure

The general syntax of a function in C is:

```
type_specifier function_name (arguments)
{
    body of the function
    return statement
}
```

The **`type_specifier`** specifies the data type of the value, which will be returned by the function. If no type is specified, the function is assumed to return an integer result. The arguments are separated by commas. A pair of empty parentheses must follow the function name even if the it does not include any arguments. Arguments appearing in the parentheses are also termed as **formal parameters** or **formal arguments**. The body of the function may consist of one or many statements. A function should return a value and hence at least one return statement should be there in a function.

15.2.1 Arguments of a function

Before discussing arguments in details, consider the following example,

Example 1:

```
#include <stdio.h>
main()
{
    int i;
    for(i=1; i<=10; i++)
        printf("\nSquare of %d is %d ", i, squarer(i));
}
squarer(int x)
```

```
/* int x */
{
    int j;
    j = x * x;
    return(j);
}
```

The above program calculates the square of numbers from 1 to 10. This is done by calling the function **squarer**. The data is passed from the calling routine (`main()` in the above case) to the called function **squarer** through arguments. The arguments are known as **actual arguments** in the calling routine and as **formal arguments** in the called function definition (`squarer()`). The data type of the **actual arguments** should be the same as the **formal arguments**. Also the number and order of the **actual arguments** should be the same as the **formal arguments.**

When a function is invoked, the control is passed to the called function where the **formal arguments** are replaced by **actual arguments**. The function is then executed and, on encountering the return statement, control is transferred back to the calling program.

The function `squarer()` is called by passing the number whose square is required. The argument **`x`** can be declared in one of the following ways, while defining the function.

Method 1:

```
squarer(int x)
/* x defined with its type in parentheses */
```

Method 2:

```
squarer(x)
int x;
/* x is in parentheses, and its type is defined immediately after the */
/* function name */
```

Note that in the last case, **`x`** has to be defined immediately after the function name, before the code block. This is helpful when many parameters of the same data type are passed. In such cases the type has to be mentioned only once at the beginning.

When the arguments are declared within the parentheses, each and every argument has to be defined individually, irrespective of whether they are of the same type. For example, if **`x`** and **`y`** are two arguments of a function `abc()`, then `abc(char x, char y)` is right declaration and `abc(char x, y)` is wrong.

Concepts

15.2.2 Returning from the Function

The return statement has two purposes:

- It immediately transfers the control from the function back to the calling program

Whatever is inside the parentheses following the `return` is returned as a value to the calling program.

In the function `squarer()`, a variable **`j`** of the type `int` is defined, which stores the square of the passed argument. This value of this variable is returned to the calling routine through the `return` statement. A function can do a special task and return control to the calling routine without returning any value. In such a case, the `return` function can be written as `return(0)` or `return`. Note that if a function is supposed to return a value and it does not do that then it will return some garbage value.

In the program to calculate squares of numbers, the program passes data to the function squarer through arguments. There can be functions that can be called without any arguments. Here, the function performs a sequence of statements and returns the value, if required.

Note that the function `squarer()` can also be written as:

```
squarer(int x)
{
   return(x*x);
}
```

This is because an expression is valid in `return` statement as it is in an argument. As a matter of fact, the `return` function can be used in any of the following ways:

```
return;
return(constant);
return(variable);
return(expression);
return(statement on evaluation); for example: return(a>b?a:b);
```

However, a limitation of `return` is that it can return only one value.

15.2.3 The type of a Function

The **`type-specifier`** is used to specify the data type of the return argument of a function. In the explained example, the **`type-specifier`** is not written besides the function `squarer()`, because `squarer()` returns a value of `int` type. The **`type-specifier`** is not compulsory if an integer value is returned or if no value is returned. However, it is better to specify `int` if an integer value is to be returned and similarly

`void` if the function returns nothing.

15.3 Invoking a Function

A function can be invoked or called from the main program simply by using its name, followed by parentheses. The parentheses are necessary to inform the compiler that a function is being referred to. When a function name is used in the calling program it is either part of a statement or a statement itself. Hence the statement always ends with a semi-colon. However, when defining the function, a semi-colon is not used in the end. The absence of the semi-colon indicates the compiler that a function is being defined and not called.

Some points to remember:

- A semicolon is used at the end of the statement when a function is called, but not after the function definition.
- Parentheses are compulsory after the function name, irrespective of whether the function has arguments or not.
- The **function**, which calls another **function**, is known as the **calling routine** or **calling function** and the **function**, which is being called, is known as the **called routine** or **called function**.
- Functions not returning an integer value have to specify the type of value being returned.
- Only one value can be returned by a function.
- A program can have one or more functions.

15.4 Function Declaration

A function should be declared in the `main()` function, before it is defined or used. This has to be done in case the function is called before it is defined.

Consider the following code snippet:

```
#include <stdio.h>
main()
{
    .
    .
    address();
```

Concepts

```
    .
    .
}
address()
{
    .
    .
    .
}
```

The `main()` function calls the function `address()` and the `address()` function is called before it is defined. Also it is not declared in the `main()` function. This is possible in some C compilers, because the function `address()` is called through a statement which contains nothing else but the function call. This is referred to as **implicit declaration** of a function.

15.5 Function Prototypes

A function prototype is a function declaration that specifies the data types of the arguments. Normally, functions are declared by specifying the type of value to be returned by the function, and the function name. However, the ANSI C standard allows the number and types of the function's arguments to be declared. A function `abc()` having two arguments **`x`** and **`y`** of type `int`, and returning a `char` type, can be declared as:

```
char abc();
```

or

```
char abc(int x, int y);
```

The later definition is called **function prototype**. When prototypes are used, C can find and report any illegal type conversions between the arguments used to call a function and the type definition of its parameters. An error will be reported even if there is a difference between the number of arguments used to call a function and define a function.

The general syntax of a function prototype definition is:

```
type function_name(type parm_namel, type parm_name2,..type parm_nameN);
```

When the function is declared without any prototype information, the compiler assumes that no information about the parameters is given. A function having no arguments can be mistaken to be declared without prototype information. To avoid this, when a function has no parameters, its prototype uses `void` inside the parentheses. As said earlier, `void` explicitly declares a function as returning no value.

For example, if a function called `noparam()` returns `char` type and has no parameters, it can be declared as

```
char noparam(void);
```

This indicates that the function has no parameters, and any call to that function, which passes parameters to the function is erroneous.

When a non-prototyped function is called all characters are converted to integers and all floats are converted to doubles. However, if a function is prototyped, the types mentioned in the prototype are maintained and no type promotions occur.

15.6 Variables

As discussed earlier, variables are named locations in memory that are used to hold a value that may or may not be modified by a program or a function. Variables are basically of three kinds: **local variables**, **formal parameters**, and **global variables**.

1. **Local variables** are those that are declared inside a function.

2. **Formal parameters** are declared in the definition of function as parameters.

3. **Global variables** are declared outside all functions.

15.6.1 Local Variables

Local variables are also known as **automatic variables**, as the keyword `auto` can be used to declare them. They can be referred to only by statements that are inside the code block, which declares them. To be precise, a local variable is created upon entry into a block and destroyed upon exit from the block. The most common code block in which a local variable is declared is a function.

Consider the following code snippet:

```
void blkl(void) /* void denotes no value returned */
{
    char ch;
    ch = 'a';
    .
    .
}
```

```
void blk2(void)
{
    char ch;
    ch = 'b';
    .
    .
}
```

The variable `ch` is declared twice, in **`blk1()`** and **`blk2()`**. The variable **`ch`** in **`blk1()`** has no bearing on or no relation to **`ch`** in **`blk2()`** because each **`ch`** is known only to the code where it is declared.

As local variables are created and destroyed within the block in which they are declared, their contents are lost outside the block scope. This implies that they cannot retain their values between calls to functions.

Though the keyword `auto` can be used to declare local variables, it is hardly ever used because all non-global variables are, by default, considered to be local.

Local variables, which are to be used by functions, are generally declared immediately after the function's opening curly brace and before any other statement. These declarations can, however, be made within a block in the function. For example,

```
void blk1(void)
{
    int t;
    t = 1;
    if(t > 5)
    {
        char ch;
        .
        .
    }
    .
}
```

In the above example the variable **`ch`** is created and will be valid only within the '`if`' code block. It cannot be referenced even in the other parts of the function **`blk1()`**.

One of the advantages of declaring a variable in this way is that memory will be allocated to it only if the condition to enter the '`if`' block is satisfied. This is because local variables are declared only after the block they are defined in is entered into.

Note: The important point to remember is that all local variables have to be declared at the start of the block in which they are defined, prior to any executable program statement.

The following code may not work with most of the compilers:

```
void blk1(void)
{
    int len;
    len = 1;
    char ch; /* This will cause an error */
    ch = 'a';
    .
    .
}
```

15.6.2 Formal Parameters

A function using arguments has to declare variables to accept values of the arguments. These variables are called **formal parameters** of the function and act like any local variable inside a function.

These variables are declared inside the parentheses that follows the function's name. Consider this example:

```
blk1(char ch, int i)
{
    if(i > 5)
        ch = 'a';
    else
        i = i + 1;
    return;
}
```

The function **`blk1()`** has two parameters : **`ch`** and **`i`**.

The formal parameters have to be declared along with their types. Like in the above example, **`ch`** is of type `char` and **`i`** is of type `int`. These variables can be used inside the function like normal local variables. They are destroyed upon exit from the function. Care needs to be taken that the formal parameters declared have the same data-types as the arguments that are used to call the function. In case of a type mismatch, C may not display any error but may give unexpected results. This is because, C generally gives some result even in unusual circumstances. The programmer has to ensure that there are no type mismatch errors.

Concepts

As with local variables, assignments can be made to a function's formal parameters and they can also be used in any allowable C expression.

15.6.3 Global Variables

Global variables are visible to the entire program, and can be used by any piece of code. They are declared outside any function of a program and hold their value throughout the execution of the program. These variables can be declared either outside the `main()` or declared anywhere before its first use. However, it is best to declare global variables at the top of the program, that is, before the `main()` function.

```
int ctr; /* ctr is global */
void blk1(void);
void blk2(void);
void main(void)
{
    ctr = 10;
    blk1 ();
    .
    .
}
void blk1(void)
{
    int rtc;
    if (ctr >8)
    {
        rtc = rtc + 1;
        blk2();
    }
}
void blk2(void)
{
    int ctr;
    ctr = 0;
}
```

In the above code, **`ctr`** is a global variable and even though it is declared outside `main()` and **`blk1()`**, it can be referenced within them. The variable **`ctr`** in **`blk2()`**, though, is a local variable and has no connection with the global variable **`ctr`**. If a global and local variable have the same name, all references to that name within the block defining the local variable will be associated with the local variable and not the global variable.

Storage for global variables is in fixed region of memory. Global variables are useful when many functions in a program use the same data. However, unnecessary use of global variables should be avoided mainly because they take up memory for the entire time that the program is executing. Also, using a global variable where a local variable would be enough makes the function using it, less general. This will be clear from the following program code:

```
void addgen(int i, int j)
{
    return(i + j);
}
int i, j;
void addspe(void)
{
    return(i + j);
}
```

Both **`addgen()`** and **`addspe()`** return the sum of the variables **`i`** and **`j`**. The **`addgen()`** function, however, is used to return the sum of any two numbers; while the **`addspe()`** function returns only the sum of the global variables **`i`** and **`j`**.

15.7 Storage Classes

Every C variable has a feature called as **storage class**. The storage class defines two aspects of the variable: its **lifetime** and its **visibility** (**or scope**). The **lifetime** of a variable is the length of time it retains a particular value. The **visibility** of a variable defines which parts of a program will be able to recognize it (the variable). A variable may be visible in a block, a function, a file, a group of files, or an entire program.

From the C compiler's point of view, a variable name identifies some physical location within the computer, where the string of bits representing the variable's value is stored. There are basically two kinds of locations in a computer where such a value may be kept: the memory or the CPU register. The variable's storage class determines whether the variable should be stored in the memory or in a register. There are four storage classes in C. These are:

- `automatic`
- `external`
- `static`
- `register`

Keywords in bold indicate their usage as a storage specifier. The storage specifier leads the rest of the

variable declaration. Its general syntax is:

```
storage_specifier type var_name;
```

15.7.1 Automatic Variables

These are nothing but the local variables, which have been discussed previously. The scope of an automatic variable can be smaller than the entire function, if it is declared from within a compound statement. The scope is then restricted to that compound statement. They can be declared using the specifier `auto`, though the declaration is not necessary. Any variable declared within a function or a code block is by default of the class auto and the system sets aside the required memory area for that variable.

15.7.2 Extern

In C, a large program can be split into smaller modules, which can be compiled separately and then linked together. This is done to speed up the compilation process for large projects. However, when the modules are linked, all the files have to be told of the global variables required by the program. A global variable can be declared only once. If two global variables having the same name are declared inside the same file, an error message like '**duplicate variable name**' may be displayed or the C compiler might simply choose one variable. The same problem occurs if all global variables required by the program are included in each and every file. Although the compiler does not issue any error message at compile time, the fact remains that copies of the same variable are being made. At the time of linking the files, the linker displays an error message such as '**duplicate label**' because it does not know which variable to use. The `extern` class is used, in a case like this. All global variables are declared in one file and the same variables are declared as `extern` in all other files. Consider the following code:

```
File1                  File2
int i, j;              extern int i, j;
char a;                extern char a;
main()                 xyz()
{                      {
   .                     i = j * 5
   .                     .
   .                     .
}                      }
abc()                  pqr()
{                      {
   i = 123;              j = 50;
    .                    .
    .                    .
}                      }
```

File2 has the same global variables as **File1**, except for the fact that these variables have the `extern` specifier added to their declaration. This specifier tells the compiler the types and names of the global variables being used without actually creating storage for them again.

When the two modules are linked, all references to the external variables are solved. If a variable has not been declared within a function, the compiler checks whether it matches any of the global variables. If a match is found, the compiler assumes that a global variable is being referred to.

15.7.3 Static Variables

The static variables are permanent variables within their own functions or files. Unlike global variables they are not known outside their function or file, but they maintain their values between calls. This means that, if a function terminates and is then re-entered later, the static variables defined within that function would still retain their values. The variable declaration begins with the static storage class designation.

It is possible to define static variables having the same names as leading external variables. The local variables (static as well as auto) take precedence over the external variables and the value of the external variables will be unaffected by any changes of the local variables. External variables having the same names as the local variables in a function cannot be accessed in that function directly.

Initial values can be assigned to variables within static variable declarations but these values must be expressed as constants or expressions. The compiler automatically assigns a default value zero to any static variable, which is not initialized. Initialization takes place at the beginning of the program.

Consider the following two programs. The difference between `auto` and `static` local variables will be quite apparent.

Automatic variables

Example 2:

```
#include <stdio.h>
main()
{
    incre();
    incre();
    incre();
}
incre()
{
    char var = 65; /* var is automatic variable */
    printf("\nThe character stored in var is %c", var++);
}
```

The output for the above program will be:

```
The character stored in var is A
The character stored in var is A
The character stored in var is A
```

Static variables

Example 3:

```
#include<stdio.h>
main()
{
    incre();
    incre();
    incre();
}
incre()
{
    static char var = 65; /* var is static variable */
    printf("\nThe character stored in var is %c", var++);
}
```

The output for this program will be:

```
The character stored in var is A
The character stored in var is B
The character stored in var is C
```

Both the programs call **`incre()`** thrice. In the first program, each time **`incre()`** is called, the variable **`var`** with storage class `auto` (default storage class) is re-initialized to **`65`** (which is the ASCII equivalent of the character A). Thus when the function terminates, the new value of **`var`** (**`66`**) is lost **(ASCII character B)**.

In the second program, **`var`** is of `static` storage class. Here, **`var`** is initialized to **`65`** only once the program is compiled. At the end of the first function call, **`var`** has a value of **`66`** **(ASCII B)** and similarly in the next function call **`var`** has a value of **`67`** **(ASCII C)**. At the end of the last function call **`var`** is incremented during the execution of the `printf()` statement. This value is lost when the program terminates.

Concepts

15.7.4 Register Variables

Computers have registers in their **Arithmetic Logic Unit** (**ALU**), which are used to temporarily store data that has to be accessed repeatedly. Intermediate results of calculations are also stored in **registers**. Operations performed upon the data stored in **registers** are faster than the data stored in memory. In assembly language, a programmer has access to these **registers** and can move frequently used data into them thus making the program run faster. A high level programmer usually does not have access to the computer's registers. In C, the option of choosing a storage location for a value has been left to the programmer. If a particular value is to be used often (for example, the value that controls a loop), its storage class can be named as **register**. Then if the compiler finds a free register, and if the machine's **registers** are big enough to hold the variable, it (the variable) is placed in that register. Otherwise, the compiler treats register variables as any other automatic variables, i.e. it stores them in the memory. The keyword `register` must be used to define the `register` variables.

The **scope** and **initialization** of the **register** variables is the same as that of **automatic** variables, except for the location of storage. Register variables are local to a function. That is, they come into existence, when the function is invoked and the value is lost once the function is exited from. Initialization of these variables is the responsibility of the programmer.

As the number of registers available is limited, a programmer has to determine which variables used in the program are used repeatedly and then declare them as register variables.

The usefulness of register variables varies from one machine to another and from one C compiler to another. Sometimes, **register** variables are not supported at all – the keyword `register` is accepted but treated just like the keyword `auto`. In other cases, if **register** variables are supported and if the programmer uses it with care, a program can be made to run twice as fast.

The `register` variables are declared as shown below:

```
register int x;
register char c;
```

The register declaration can only be applied to automatic variables and formal arguments. In the latter case, the declaration looks like the following:

```
f(c, n)
register int c, n;
{
    register int i;
    .
    .
    .
}
```

Concepts

Consider an example, where the sum of the cubes of the digits of a number is equal to the number itself need to be displayed. For example, 370 is such a number because

$3^3 + 7^3 + 0 = 27 + 343 + 0 = 370$

The following program prints all such numbers in the range from **1** to **999**.

Example 4:

```
#include <stdio.h>
main()
{
    register int i;
    int no, digit, sum;
   printf(" \nThe numbers whose Sum of Cubes of Digits is Equal to the number
itself are :\n\n");
    for(i=1;i<999;i++)
    {
        sum = 0;
        no = i;
        while(no)
        {
            digit = no%10;
            no = no/10;
            sum = sum + digit * digit * digit;
        }
        if(sum==i)
            printf("t%d\n", i);
    }
}
```

The output for the above program is:

```
The numbers whose Sum of Cubes of Digits is Equal to the number itself are:
1
153
370
371
407
```

In the above program, the value of `i` ,varies from 1 to `999`. For each of these value, the cubes of individual digits are added and the resultant sum is compared to `i`. If these two values are equal, `i` is displayed. Since `i` is used to control the looping, (the most essential part of the program), it is declared to be of the storage class register. This declaration increases the efficiency of the program.

15.8 Scope Rules for a Function

Scope rules are rules that decide whether one piece of code knows about or has access to another piece of code or data. In C, each function of a program is a separate block of code. The code within a function is private or local to that function and cannot be accessed by any statement in any other function except through a call to that function. The code within a function is hidden from the rest of the program and, unless it uses global variables or data, it can neither affect or be affected by other parts of the program. To be precise, the code and data defined within one function cannot interact with the code or data defined in another function because the two functions have different scopes.

In C, all functions are at the same scope level. This means, that a function cannot be defined within another function. Because of this aspect, C is technically not a block-structured language.

15.9 Calling the Function

In general, functions communicate with each other by passing arguments. Arguments can be passed in one of the following two ways:

- Call by value
- Call by reference.

15.9.1 Call by Value

In C, by default, all function arguments are passed by value. This means that, when arguments are passed to the called function, the values are passed through temporary variables. All manipulations are done on these temporary variables only. The called function cannot change its value. Consider the following example.

Example 5:

```
#include <stdio.h>
main()
{
    int a, b, c;
    a = b = c = 0;
```

Concepts

```
    printf("\nEnter 1st integer : ");
    scanf("%d", &a);
    printf("\nEnter 2nd integer : ");
    scanf("%d", & b);
    c = adder(a, b);
    printf("\n\na & b in main() are : %d, % d", a, b);
    printf("\n\nc in main() is : %d", c);
    /* c gives the addition of a and b */
}
adder(int a, int b)
{
    int c;
    c = a + b;
    a *= a;
    b += 5;
    printf("\n\na & b within adder function are: %d, %d ", a, b);
    printf("\nc within adder function is : %d", c);
    return(c);
}
```

The sample output for an input of **2** and **4** will be:

```
a & b in main() are : 2, 4
c in main() is : 6
a & b within adder function are : 4, 9
c within adder function is : 6
```

The above program accepts two integers, which are passed to the function **adder()**. The function `adder()` does the following : it takes the two integers as its arguments, adds them, squares the first integer, adds **5** to the second integer, prints the result and returns the sum of the actual arguments. The variables which are used in the `main()` and the **adder()** functions have the same name. However nothing else is common between them. They are stored in different memory locations. This is clear from the output of the above program. The variables **a** and **b** in the function **adder()** are altered from **2** and **4** to **4** and **9** respectively. However this change does not affect the values of the **a** and **b** in the `main()` function. The variables must be stored in different memory locations. The variable **c** in `main()` is different from the variable **c** in **adder()**.

So, arguments are said to be passed using call by value when the value of the variables are passed to the called function and any alterations on this value has no effect on the original value of the passed variable.

Concepts

15.9.2 Call by Reference

When arguments are passed using call by value, the values of the arguments of the calling routine are not changed. However, there may be cases, where the values of the arguments have to be changed. In such cases, **call by reference** could be used. In **call by reference**, the function is allowed access to the actual memory locations of the arguments and therefore can change the value of the arguments of the calling routine.

For example, consider a function, which takes two arguments, interchanges their values and returns them. If a program like the one given below is written for this purpose, it will never work.

Example 6:

```
#include <stdio.h>
main()
{
    int x, y;
    x = 15; y = 20;
    printf("x = %d, y = %d\n", x, y);
    swap(x, y);
    printf("\nAfter interchanging x = %d, y = %d\n", x, y);
}
swap(int u, int v)
{
    int temp;
    temp = u;
    u = v;
    v = temp;
    return;
}
```

The output for the above will be as follows:

```
x = 15, y = 20
After interchanging x = 15, y = 20
```

The function **`swap()`** interchanges the values of **`u`** and **`v`**, but these values are not passed back to the `main()`. This is because the variables **`u`** and **`v`** in **`swap()`** are different from the variables **`u`** and **`v`** used in `main()`. A call by reference can be used to achieve the desired result, because it will change the values of the actual arguments. Pointers are used when a call by reference has to be made.

Concepts

Pointers are passed to a function as arguments to enable the called routine of the program to access variables whose scope does not extend beyond the calling function. When a pointer is passed to a function, the address of a data item is passed to the function making it possible to freely access the contents of that address from within the function. The function as well as the calling routine recognizes any change made to the contents of the address. In this way, function arguments permit data-items to be altered in the calling routine, enabling a two-way transfer of data between the calling routine and the function. When the function arguments are pointers or arrays, a **call by reference** is made to the function as opposed to a **call by value** for the variable arguments.

Formal arguments of a function, which are pointers, are preceded by an asterisk `(*)`, just like pointer variable declarations, indicating them to be pointers. Actual pointer arguments in a function call must either be declared as pointers or as referenced variables `(&var)`.

For example, the function definition

```
getstr(char *ptr_str, int *ptr_int)
```

indicates that the arguments **`ptr_str`** points to type `char` and **`ptr_int`** points to type `int`. The function can be called by the statement,

```
getstr(pstr, &var)
```

where, **`pstr`** is declared as a pointer and the address of the variable **`var`** is passed. Assigning a value through,

```
*ptr_int = var;
```

the function, can now assign values to the variable **`var`** in the calling routine, enabling a two way transfer to and from the function.

```
char *pstr;
```

Consider the same example of **`swap()`** as shown in Example 7. This problem will work when pointers are passed instead of variables.

Example 7:

```
#include <stdio.h>
void main()
{
    int x, y, *px, *py;
    /* Storing address of x in px */
```

```
    px = &x;
    /* Storing address of y in py */
    py = &y;
    x = 15; y = 20;
    printf("x = %d, y = %d \n", x, y);
    swap (px, py);
    /* Passing addresses of x and y */
    printf("\n After interchanging x = %d, y = %d\n", x, y);
}
swap(int *u, int *v)
/* Accept the values of px and py into u and v */
{
    int temp;
    temp = *u;
    *u = *v;
    *v = temp;
    return;
}
```

The output of the above example will be:

```
x = 15, y = 20
After interchanging x = 20, y = 15
```

Two pointer type variables **px** and **py** are declared, and the addresses of the variables **x** and **y** are assigned to them. These pointer variables are then passed to the function **swap()**, which interchanges the values stored in **x** and **y** through the pointers.

15.10 Nesting of Function Calls

Calling one function from another is said to be **nesting** of **function calls**. A program which checks whether a string is a palindrome or not, could be considered as an example for nested function calls. A palindrome is a string of characters, which is the same when read forward or backwards. Consider the following code:

```
main()
{
    .
    .
```

```
    palindrome();
    .
    .
}
palindrome()
{
    .
    .
    getstr();
    reverse();
    cmp();
    .
    .
}
```

In the above program, the function `main()` calls the function `palindrome()`. The function **`palindrome()`** calls three other functions **`getstr()`**, **`reverse()`** and **`cmp()`**. The **`getstr()`** function obtains a string of characters from the user, the function **`reverse()`** reverses the input string and the function **`cmp()`** compares the input string and the reversed string.

Since `main()` calls **`palindrome()`**, which in turn calls the **`getstr()`**, **`reverse()`** and **`cmp()`** functions, the function calls are said to be nested within **`palindrome()`**.

Nesting of function calls as shown above is allowed, whereas defining one function within another function is not allowed by C.

15.11 Functions in Multifile Programs

Programs can be composed of multiple files. Such programs could make use of lengthy functions, where each function may occupy a separate file. As variables in multifile programs, functions can also be defined as `static` or `external`. The scope of the `external` function is through all the files of the program and it is the default storage class for functions. `Static` functions are recognized only within the program file and their scope does not extend outside the program file. The function header will look like,

```
static fn _type fn_name (argument list)
```

or

```
extern fn_type fn_name (argument list)
```

The keyword `extern` is optional as it is the default storage class.

15.12 Pointers to Functions

A confusing yet powerful feature of C is the **function pointer**. Even though a function is not a variable, it has a physical location in memory that can be assigned to a pointer. A function's address is the entry point of the function and function pointer can be used to call a function.

To know how function pointers work, it is necessary to be very clear as to how a function is compiled and called in C. As each function is compiled, source code is transformed into object code and the entry point is established. When a call is made to a function, a machine language call is made to this entry point. Therefore, if a pointer contains the address of a function's entry point, it can be used to call that function. The address of a function can be obtained by using the function's name without any parentheses or arguments. The following program will demonstrate the concept of function pointer.

Example 8:

```
#include <stdio.h>
#include <string.h>
void check(char *a, char *b, int (*cmp)());
main()
{
    char sl[80];
    int (*p)();
    p = strcmp;
    gets(s1);
    gets(s2);
    check(s1, s2, p);
}
void check(char *a, char *b, int (*cmp)())
{
    printf("testing for equality \n");
    if(!(*cmp)(a,b))
        printf("Equal");
    else
        printf("Not Equal");
}
```

The function **`check()`** is called by passing two character pointers and one function pointer. Inside **`check()`**, the arguments are declared as character pointers and a function pointer. Notice how the function pointers are declared. A similar syntax could be used when declaring other function pointers irrespective of the return type of the function. The parentheses around the **`*cmp`** are necessary for the compiler to interpret this statement correctly.

Concepts

Summary

- Functions are generally used in C to execute a series of instructions more than once.
- The `type _ specifier` specifies the data type of the value that will be returned by the function.
- The arguments to a function can be constants, variables, expressions or functions.
- The arguments are known as actual arguments in the calling routine and as formal arguments in the called function.
- A function has to declared in main(), before it is defined or used.
- In C, by default, all function arguments are passed by value.
- Variables are basically of three kinds: local variables, formal parameters, and global variables.
    - Local variables are declared inside a function.
    - Formal parameters are declared in the definition of function parameters.
    - Global variables are declared outside all functions.
- The storage class defines two features of the variable; its **lifetime** and its **visibility** or **scope**).
- Automatic variables are the same as local variables.
- All global variables are declared in one file and the same variables are declared as **extern** in all other files using them.
- The static variables are permanent variables within their own functions or file.
- Unlike global variables, static variables are not known outside their function or file, but they maintain their values between calls.
- If a particular value is to be used often, its storage class can be named as register.

Summary

- As with variables in multifile programs, functions can also be defined as static or **external**.
- The code and data defined within one function cannot interact with the code or data defined in another function because the two function have different scopes.
- A function cannot be defined within another function.
- A function prototype is a function declaration that specifies the data types of the arguments.
- Calling one function from within another is said to be nesting of **function calls**.
- A function pointer can be used to call a function.

Check Your Progress

1. A _____ is a self-contained program segment that carries out a specific, well defined task.

2. Arguments appearing in the parentheses are termed as _____.

3. If the return is ignored, control passes to the calling program when the closing braces of the code block are encountered. This is termed as _____.

4. The function, which calls another function, is known as the _____ and the function, which is being called, is known as the _____.

5. A _____ is a function declaration that specifies the data types of the arguments.

6. _____ can be referred to only by statements that are inside the code block, which declares them.

7. _____ are visible to the entire program, and can be used by any piece of code.

8. _____ govern whether one piece of code knows about or has access to another piece of code or data

9. Arguments are said to be passed _____ when the value of the variables are passed to the called function

10. In_____, the function is allowed access to the actual memory location of the argument.

Session

16 Functions (Lab)

Objectives

At the end of this session, you will be able to:

- *Define and call function*
- *Use of parameters in function*

Part I – For the first 1 Hour and 30 Minutes:

16.1 Functions

As we already know, a function is a self-contained block of statements that perform a task of some kind. In this session, let us focus on how to create and use functions.

16.1.1 Define a function

A function is defined with a function name, is followed by a pair of curly braces in which one or more statements may be present. For example,

```
argentina()
{
    statement 1;
    statement 2;
    statement 3;
}
```

16.1.2 Calling a function

A function can be called from a main program by stating its name followed by a pair of braces and a semi-colon. For example,

```
argentina();
```

Now, let us look at the complete program.

1. Invoke the editor in which you can type the C program.

Lab Guide

2. **Create a new file.**

3. **Type the following code :**

```
#include<stdio.h>
void main()
{
    printf("\n I am in main");
    italy();
    brazil();
    argentina();
}
italy()
{
    printf("\n I am in italy");
}
brazil()
{
    printf("\n I am in brazil");
}
argentina()
{
    printf("\n I am in argentina");
}
```

To see the output, follow these steps:

4. **Save the file with the name `functionI.C`.**

5. **Compile the file, `functionI.C`.**

6. **Execute the program, `functionI.C`.**

7. **Return to the editor.**

The sample output of the above program will be as shown in Figure 16.1.

Figure 16.1: Output of functionI.C

16.2 Use of parameters in functions

Parameters are used to convey information to the function. The format strings and the list of variables used inside the parenthesis in the functions are parameters.

16.2.1 Define a parameterized function

A function is defined with a function name is followed by an opening brace followed by a parameter(s) and finally closing brace. Inside the function, one or more statements may be present. For example,

```
calculatesum (int x, int y, int z)
{
    statement 1;
    statement 2;
    statement 3;
}
```

Now, Let us look at the complete program.

1. **Create a new file.**

2. **Type the following code :**

```
#include<stdio.h>
void main()
{
    int a, b, c, sum;
```

```
    printf("\n Enter any three numbers: ");
    scanf("%d %d %d", &a, &b, &c);
    sum = calculatesum(a, b, c);
    printf("\n Sum = %d", sum);
}
calculatesum(int x, int y, int z)
{
    int d;
    d = x + y + z;
    return (d);
}
```

To see the output, follow these steps:

3. **Save the file with the name `functionII.C`.**

4. **Compile the file, `functionII.C`.**

5. **Execute the program, `functionII.C`.**

6. **Return to the editor.**

The sample output of the above program will be as shown in Figure 16.2.

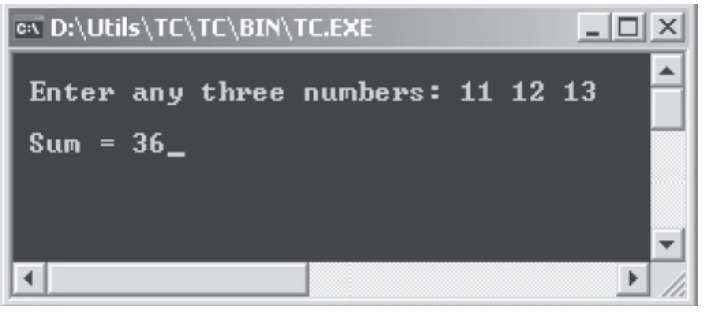


Figure 16.2: Output of functionII.C

Part II – For the next 30 Minutes:

1. Write a C program that accepts a number and square the number with the help of a function.

   To do this,

   a. Declare a function.

   b. Accept the number.

   c. Pass the number to the function and return the square of that number.

Try It Yourself

1. Write a C program to find the area and perimeter of a circle.

2. Write a C program to calculate the factorial of an integer.

Session

17 Strings

Concepts

Objectives

At the end of this session, you will be able to:

- *Explain string variables and constants*
- *Explain pointers to strings*
- *Perform string input/output operations*
- *Explain the various string functions*
- *Explain how arrays can be passed as arguments to functions*
- *Describe how strings can be used as function arguments*

Introduction

Strings in C are implemented as arrays of characters terminated by the NULL ('\0') character. This session discusses the usage and manipulation of strings.

17.1 String variables and constants

String variables are used to store a series of characters. Like any other variable, these variables must be declared before they are used. A typical string variable declaration is:

```
char str[10];
```

`str` is a character array variable that can hold a maximum of 10 characters. Consider that **`str`** is assigned a string constant,

```
"WELL DONE"
```

A string constant is a sequence of characters surrounded by double quotes. Every character in the string is stored as an array element. In memory, the string is stored as follows:

'W'	'E'	'L'	'L'	' '	'D'	'O'	'N'	'E'	'\0'

The '\0' (null) character is automatically added in the internal representation so that the program can locate the end of the string. So, while declaring a string variable, allow one extra element space for the

Concepts

null terminator.

17.1.1 Pointers to strings

Strings can also be stored and accessed using character pointers. A character pointer to a string is declared as follows:

```
char *pstr = "WELCOME";
```

pstr is a pointer that is initialized to point to a string constant. The pointer may be modified to point elsewhere. However, the modification would cause the string to be inaccessible.

17.1.2 String I/O operations

String input/output (I/O) operations in C are carried out using function calls. These functions are part of the standard I/O library called `stdio.h`. A program that uses the string I/O functions must have the following statement at the beginning:

```
#include <stdio.h>;
```

When the program containing this statement is compiled, the contents of the file `stdio.h` become a part of the program.

- **Simple string I/O operations**

  The `gets()` function is the simplest method of accepting a string through standard input. Input characters are accepted till the **Enter** key is pressed. The `gets()` function replaces the terminating `'\n'` new line character with the `'\0'` character. The syntax is as follows:

  ```
  gets(str);
  ```

  where, **str** is a character array that has been declared.

  Similarly, the `puts()` function is used to display a string on the standard output device. The newline character terminates the string output. The function syntax is:

  ```
  puts(str);
  ```

  where, **str** is a character array that has been declared and initialized. The following program accepts a name and displays a message.

Example 1:

```
#include <stdio.h>
void main()
{
    char name[20];
    /* name is declared as a single dimensional character array */
    clrscr(); /* Clears the screen */
    puts("Enter your name:"); /* Displays a message */
    gets(name); /* Accepts the input */
    puts("Hi there:");
    puts(name); /* Displays the input */
    getch();
}
```

If the name **`Lisa`** is entered, a sample output for the above program will be:

```
Enter your name:
Lisa
Hi there:
Lisa
```

➢ **Formatted string I/O operations**

The `scanf()` and `printf()` functions can also be used to accept and display string values. These functions are used to accept and display mixed data types with a single statement. The syntax to accept a string is as follows:

```
scanf("%s", str);
```

where `%s` is the format specifier that states that a string value is to be accepted. **`str`** is a character array that has been declared. Similarly, to display a string, the syntax is:

```
printf("%s", str);
```

where the `%s` format specifier states that a string value is to be displayed and **`str`** is a character array that has been declared and initialized. The `printf()` function can also be used without the format specifier to display messages.

The earlier program can be modified to accept and display a name using `scanf()` and `printf()`.

Concepts

Example 2:

```
#include <stdio.h>
void main()
{
    char name[20];
    /* name is declared as a single dimensional character array */
    clrscr(); /* Clears the screen */
    printf("Enter your name:"); /* Displays a message */
    scanf("%s",name); /* Accepts the input */
    printf("Hi there: %s",name); /* Displays the input */
    getch();
}
```

If the name **Brendan** is entered, a sample output for the above program will be:

```
Enter your name: Brendan
Hi there: Brendan
```

17.2 String functions

C supports a wide range of string functions, which are found in the standard header file `string.h`. Few of the operations performed by these functions are:

- Concatenating strings
- Comparing strings
- Locating a character in a string
- Copying one string to another
- Calculating the length of a string

17.2.1 The 'strcat()' function

The `strcat()` function is used to join two string values into one. The syntax of the function is:

```
strcat(str1, str2);
```

where, **str1** and **str2** are two character arrays that have been declared and initialized. The value in

str2 is attached at the end of **str1**.

The following program accepts a first name and a last name. It concatenates the last name to the first name and displays the concatenated name.

Example 3:

```
#include<stdio.h>
#include<string.h>
void main()
{
    char firstname[15];
    char lastname[15];
    clrscr();
    printf("Enter your first name:");
    scanf("%s",firstname);
    printf("Enter your last name:");
    scanf("%s",lastname);
    strcat(firstname, lastname);
    /* Attaches the contents of lastname at the end of firstname */
    printf("%s", firstname);
    getch();
}
```

A sample output for the above program will be:

```
Enter your first name: Carla
Enter your last name: Johnson
CarlaJohnson
```

17.2.2 The 'strcmp()' function

The equality (or inequality) of two numbers can be verified using relational operators. However, to compare strings, a function call has to be made. The function `strcmp()` compares two strings and returns an integer value based on the results of the comparison. The syntax of the function is:

```
strcmp(str1, str2);
```

where, **str1** and **str2** are two character arrays that have been declared and initialized. The function returns a value:

- Less than zero if **str1** < **str2**
- Zero if **str1** is same as **str2**
- Greater than zero if **str1** > **str2**

The following program compares one name with three other names and displays the results of the comparisons.

Example 4:

```
#include <stdio.h>
#include<string.h>
void main()
{
    char name1[15] = "Geena";
    char name2[15] = "Dorothy";
    char name3[15] = "Shania";
    char name4[15] = "Geena";
    int i;
    clrscr();
    i = strcmp(name1,name2);
    printf("%s compared with %s returned %d\n", name1, name2, i);
    i=strcmp(name1,name3);
    printf("%s compared with %s returned %d\n", name1, name3, i);
    i=strcmp(name1,name4);
    printf("%s compared with %s returned %d\n", name1, name4, i);
    getch();
}
```

A sample output for the above program will be:

```
Geena compared with Dorothy returned 3
Geena compared with Shania returned -12
Geena compared with Geena returned 0
```

Note the value returned for each comparison. It is the difference between the ASCII values of the first different characters encountered in the two strings.

Concepts

17.2.3 The 'strchr()' function

The `strchr()` function determines the occurrence of a character in a string. The syntax of the function is:

```
strchr(str, chr);
```

where, **`str`** is a character array or string. **`chr`** is a character variable containing the value to be searched. The function returns a pointer to the value located in the string, or NULL if it is not present.

The following program determines whether the character 'a' occurs in two specified city names.

Example 5:

```
#include<stdio.h>
#include<string.h>
void main()
{
    char str1[15] = "New York";
    char str2[15] = "Washington";
    char chr = 'a', loc;
    clrscr();
    loc = strchr(str1,chr);
    /* Checks for the occurrence of the character value held by chr */
    /* in the first city name */
    if(loc != NULL)
        printf("%c occurs in %s\n", chr, str1);
    else
        printf("%c does not occur in %s\n", chr, str1);

    loc = strchr(str2,chr);
    /* Checks for the occurrence of the character in the second city name */
    if(loc != NULL)
        printf("%c occurs in %s\n", chr, str2);
    else
        printf("%c does not occur in %s\n", chr, str2);

    getch();
}
```

The output for the above program is:

```
a does not occur in New York
a occurs in Washington
```

17.2.4 The 'strcpy()' function

There are no operators in C for handling a string as a single unit. So, the assignment of one string value to another requires the use of the function `strcpy()`. The syntax of the function is:

```
strcpy(str1, str2);
```

where, **`str1`** and **`str2`** are character arrays that have been declared and initialized. The function copies the value in **`str2`** onto **`str1`** and returns **`str1`**.

The following program demonstrates the use of the `strcpy()` function. It changes the name of a hotel and displays the new name.

Example 6:

```
#include <stdio.h>
#include<string.h>
void main()
{
    char hotelname1[15] = "Sea View";
    char hotelname2[15] = "Sea Breeze";
    clrscr();
    printf("The old name is %s\n", hotelname1);
    strcpy(hotelname1, hotelname2);
    /* Changes the hotel name */
    printf("The new name is %s\n", hotelname1);
    /* Displays the new name */
    getch();
}
```

The output of the above program is:

```
The old name is Sea View
The new name is Sea Breeze
```

17.2.5 The 'strlen()' function

The `strlen()` function returns the length of a string. The length of a string can be useful in programs accessing each character of a string in a loop. The syntax of the function is:

```
strlen(str);
```

where, **`str`** is a character array that has been declared and initialized. The function returns the length of **`str`**.

The following program shows a simple implementation of the `strlen()` function. It determines the length of a company name and displays the name along with additional characters.

Example 7:

```
#include <stdio.h>
#include<string.h>
void main()
{
    char compname[20] = "Microsoft";
    int len, ctr;
    clrscr();
    len = strlen(compname);
    /* Determines the length of the string */
    for(ctr = 0; ctr < len; ctr++)
        /* Accesses and displays each character of the string */
        printf("%c * ", compname[ctr]);
    getch();
}
```

The output of the above program is:

```
M * i * c * r * o * s * o * f * t *
```

17.3 Passing Arrays to Functions

In C, when an array is passed as an argument to a function, only the address of the array is passed. The array name without the subscripts refers to the address of the array. The following code snippet passes the address of the array **`ary`** to the function **`fn_ary()`**:

```
void main()
{
    int ary[10];
    ...
    fn_ary(ary);
    ...
}
```

If a function receives a single-dimensional array, the formal parameters can be declared in one of the following ways.

```
fn_ary (int ary [10]) /* sized array */
{
    :
}
```

or

```
fn_arry (int ary []) /* unsized array */
{
    :
}
```

Both the above declarations produce the same results. The first method employs the standard array declaration. In the second version, the array declaration simply specifies that an array of type `int` of some length is required.

The following program accepts numbers into an integer array. The array is then passed to a function **`sum_arr()`**. The function computes and returns the sum of the numbers in the array.

Example 8:

```
#include <stdio.h>
void main()
{
    int num[5], ctr, sum=0;
    int sum_arr(int num_arr[]); /* Function declaration */
    clrscr();
    for(ctr = 0; ctr < 5; ctr++) /* Accepts numbers into the array */
    {
        printf("\nEnter number %d: ", ctr+1);
```

```
        scanf("%d", &num[ctr]);
    }
    sum = sum_arr(num); /* Invokes the function */
    printf("\nThe sum of the array is %d", sum);
    getch();
}
    int sum_arr(int num_arr[]) /* Function definition */
    {
        int i, total;
        for(i=0,total=0;i<5;i++) /* Calculates the sum */
           total+=num_arr[i];
        return total; /* Returns the sum to main() */
    }
```

A sample output of the above program is:

```
Enter number 1: 5
Enter number 2: 10
Enter number 3: 13
Enter number 4: 26
Enter number 5: 21
The sum of the array is 75
```

17.4 Passing Strings to Functions

Strings, or character arrays, can also be passed to functions. For example, the following program accepts strings into a two-dimensional character array. Then, the array is passed to a function that determines the longest string in the array.

Example 9:

```
#include <stdio.h>
void main()
{
    char lines[5][20];
    int ctr, longctr=0;
    int longest(char lines_arr[][20]);
    /* Function declaration */
    clrscr();
    for(ctr = 0; ctr < 5; ctr++) /* Accepts string values into the array */
```

Concepts

```
    {
        printf("\nEnter string %d: ", ctr+1);
        scanf("%s", lines[ctr]);
    }
    longctr = longest(lines);
    /* Passes the array to the function */
    printf("\nThe longest string is %s", lines[longctr]);
    getch();
}
int longest(char lines_arr[][20]) /* Function definition */
{
    int i=0, l_ctr=0, prev_len, new_len;
    prev_len = strlen(lines_arr[i]);
    /* Determines the length of the first element */
    for(i++; i<5; i++)
    {
        new_len=strlen(lines_arr[i]);
        /* Determines the length of the next element */
        if(new_len > prev_len)
            l_ctr=i;
        /* Stores the subscript of the longer string */
            prev_len = new_len;
    }
    return l_ctr;
    /* Returns the subscript of the longest string */
}
```

A sample output of the program is given below.

```
Enter string 1: The
Enter string 2: Sigma
Enter string 3: Protocol
Enter string 4: Robert
Enter string 5: Ludlum
The longest string is Protocol
```

Concepts

Summary

- Strings in C are implemented as arrays of characters terminated by the NULL ('\0') character.
- String variables are used to store a series of characters.
- A string constant is a sequence of characters surrounded by double quotes.
- Strings can be stored and accessed using character pointers.
- String I/O operations in C are carried out using functions that are part of the standard I/O library called stdio.h.
- The gets() and puts() functions are the simplest method of accepting and displaying strings respectively.
- The scanf() and printf() functions can be used to accept and display strings along with other data types.
- C supports a wide range of string functions, which are found in the standard header file string.h.
- The strcat() function is used to join two string values into one.
- The function strcmp() compares two strings and returns an integer value based on the results of the comparison.
- The strchr() function determines the occurrence of a character in a string.
- The strcpy() function copies the contents of one string onto another.
- The strlen() function returns the length of a string.
- In C, when an array is passed as an argument to a function, only the address of the array is passed.
- The array name without the subscripts refers to the address of the array.

Concepts

Check Your Progress

1. Strings are terminated by the _____ character.

2. The number of characters that can be input into `char arr[15]` is _____.

3. Modification of the string pointer can lead to data loss. **(True / False)**

4. The character is used to print a new line in `printf()`.

5. To use the `strcat()` function, the _____ header file must be included in the program.

6. Two pointers can be compared only if both these variables are pointing to variables of different types. **(True / False)**

7. `strcmp()` returns _____ if two strings are identical.

8. When an array is passed to a function, only its _____ is passed.

Try It Yourself

1. Write a program that accepts two strings. The program should determine whether the first string occurs at the end of the second string.

2. Write a program that accepts an array of integers and displays the average. Use a function to calculate the average.

“Woe to him who teaches men faster than they can learn”

Session

18 Strings (Lab)

Objectives

At the end of this session, you will be able to:

- *Use strings functions*
- *Pass arrays to functions*
- *Pass strings to functions*

The steps given in this session are detailed comprehensive and carefully thought through. This has been done so that the learning objectives are met and the understanding of the tool is complete. Please follow the steps carefully.

Part I – For the first 1 Hour and 30 Minutes:

18.1 Strings Functions

The string functions in C are found in the standard header file `string.h`. This file must be included in each program that uses the string functions.

18.1.1 Sorting strings using library functions

String functions are useful for manipulating character arrays. For example, the length of a string can be determined using the `strlen()` function. Let us write a C program to sort 5 strings in descending order of their length. The steps are listed below:

1. As we learnt in the theory session, in C, to use the string functions from the library we need to include the two header files: `stdio.h, string.h`.

The code for the same will be,

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

2. Declare a character array to hold the 5 strings.

The code will be,

```
char str_arr[5][20];
```

3. Accept the five strings in a `for` loop.

The code will be,

```
for(i=0;i<5;i++)
{
  printf("\nEnter string %d: ",i+1);
  scanf("%s", str_arr[i]);
}
```

4. Compare the length of each string with the others. Swap if the string length is less than the other.

The code will be,

```
for(i=0;i<4;i++)
  for(j=i+1;j<5;j++)
  {
    if(strlen(str_arr[i]) < strlen(str_arr[j]))
    {
       strcpy(str, str_arr[i]);
       strcpy(str_arr[i], str_arr[j]);
       strcpy(str_arr[j], str);
    }
  }
```

An array `str` is being used to aid the swap operation.

5. Display the strings in the sorted array.

The code for the same will be,

```
printf("\nThe strings in descending order of length are:");
for(i=0;i<5;i++)
   printf("\n%s", str_arr[i]);
```

Let us look at the complete program.

1. **Invoke the editor in which you can type the C program.**

2. **Create a new file.**

3. **Type the following code:**

```
#include <stdio.h>
#include <string.h>
void main()
{
    int i, j;
    char str_arr[5][20], str[20];
    clrscr();
    for(i=0;i<5;i++)
    {
        printf("\nEnter string %d: ",i+1);
        scanf("%s", str_arr[i]);
    }
    for(i=0;i<4;i++)
        for(j=i+1;j<5;j++)
        {
            if(strlen(str_arr[i]) < strlen(str_arr[j]))
            {
                strcpy(str, str_arr[i]);
                strcpy(str_arr[i], str_arr[j]);
                strcpy(str_arr[j], str);
            }
        }
    printf("\nThe strings in descending order of length are:");
    for(i=0;i<5;i++)
        printf("\n%s", str_arr[i]);
    getch();
}
```

To see the output, follow these steps:

4. **Save the file with the name `stringI.C`.**

Lab Guide

5. **Compile the file, `stringI.C`.**

6. **Execute the program, `stringI.C`.**

7. **Return to the editor.**

The sample output of the program is given below:

```
Enter string 1: This
Enter string 2: sentence
Enter string 3: is
Enter string 4: not
Enter string 5: sorted
The strings in descending order of length are:
sentence
sorted
This
not
is
```

18.1.2 Convert a character array to upper case using functions

Strings can be passed to functions for manipulation. When strings, or character arrays, are passed to functions, actually the address is passed. To demonstrate this, let us write a C program to convert a set of strings to upper case. The conversion to upper case will be achieved using a function.

The steps are listed below:

1. **Include the required header files.**

The code will be,

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

2. **Declare an array to hold 5 strings.**

The code for the same is,

```
char names[5][20];
```

3. Declare a function that accepts a string as an argument.

The code for the same is,

```
void uppername(char name_arr[]);
```

4. Accept 5 strings into the array.

The code is,

```
for(i=0;i<5;i++)
{
    printf("\nEnter string %d: ", i+1);
    scanf("%s", names[i]);
}
```

5. Pass each string to the function for upper case conversion. After conversion, display the modified string.

The code for the same will be,

```
for(i=0;i<5;i++)
{
    uppername(names[i]);
    printf("\nNew string %d: %s", i+1, names[i]);
}
```

6. Define the function.

The code is,

```
void uppername(char name_arr[])
{
    int x;
    for(x=0;name_arr[x] != '\0'; x++)
    {
        if(name_arr[x]>=97 && name_arr[x]<=122)
            name_arr[x]=name_arr[x]-32;
    }
}
```

The condition checks the ASCII values of each character in the string. If the character is in lower case, it is converted to upper case. Note that the ASCII value of ‘A’ is 65 and that of ‘a’ is 97.

Let us look at the complete program.

1. Create a new file.

2. Type the following code:

```
#include <stdio.h>
#include <string.h>
void main()
{
    int i;
    char names[5][20];
    void uppername(char name_arr[]);
    clrscr();
    for(i=0;i<5;i++)
    {
        printf(”\nEnter string %d: ”, i+1);
        scanf(”%s”, names[i]);
    }
    for(i=0;i<5;i++)
    {
        uppername(names[i]);
        printf(”\nNew string %d: %s”, i+1, names[i]);
    }
    getch();
}

void uppername(char name_arr[])
{
    int x;
    for(x=0;name_arr[x] != ’\0’; x++)
    {
        if(name_arr[x]>=97 && name_arr[x]<=122)
            name_arr[x]=name_arr[x]-32;
    }
}
```

To see the output, follow these steps:

3. **Save the file with the name `stringII.C`.**

4. **Compile the file, `stringII.C`.**

5. **Execute the program, `stringII.C`.**

6. **Return to the editor.**

A sample output of the program is shown below.

```
Enter string 1: Sharon

Enter string 2: Christina

Enter string 3: Joanne

Enter string 4: Joel

Enter string 5: Joshua

New string 1: SHARON
New string 2: CHRISTINA
New string 3: JOANNE
New string 4: JOEL
New string 5: JOSHUA
```

Part II – For the next 30 Minutes:

1. Write a C Program to display the number of times a specified character occurs in a string. Set a loop to perform the operation 5 times.

   To do this,

   a. Declare a character variable and a character array.

   b. Declare a function that accepts a character array and a character variable and returns an integer value.

   c. Set up a loop to accept a string and a character 5 times.

   d. Accept the string and the character.

   e. Pass the string and the character to the function and accept the value returned in an integer variable.

   f. Print the returned value.

   g. Code the function definition. Compare each character of the string with the specified character. Increment an integer variable whenever the specified character occurs in the string. Finally, return the integer variable to the `main()`.

Try It Yourself

1. Write a C Program to accept 5 names and a prefix. Insert the prefix at the beginning of each name in the array. Display the modified names.

2. Write a C Program to accept the average yearly temperature of the past five years, for five cities. Display the maximum and minimum temperature for each city. Use functions to determine the maximum and minimum temperatures.

“Vertical thinking is digging the same hole deeper. Lateral thinking is trying again elsewhere”

Session

19 Advanced Data Types & Sorting

Concepts

Objectives

At the end of this session, you will be able to:

- *Explain structures and their use*
- *Define structures*
- *Declare structure variables*
- *Explain how structure elements are accessed*
- *Explain how structures are initialized*
- *Explain how assignment statements are used with structures*
- *Explain how structures can be passed as arguments to functions*
- *Use arrays of structures*
- *Explain the initialization of structure arrays*
- *Explain pointers to structures*
- *Explain how structure pointers can be passed as arguments to functions*
- *Explain the typedef keyword*
- *Explain array sorting with the Insertion sort and Bubble sort methods*

Introduction

Applications in the real world situations require different types of data to be stored. The predefined data types supported by C may prove inadequate in such situations. So, C allows custom data types to be created. One such data type is **structure**. A structure is a grouping of variables under one name. A data type can also be assigned a new name using the `typedef` keyword.

Applications store enormous amounts of data. In such cases, locating a particular data item can be very time consuming. Arranging the values in some sequence eases the situation. In this session, we will also look at some algorithms for sorting arrays.

Concepts

19.1 Structures

Variables can be used to hold one piece of information at a time and arrays can be used to hold several pieces of information of the same data type. However, a program may require operating upon data items of different types together as a unit. In this case, neither a variable nor an array is adequate.

For example, a program is written to store data on a catalog of books. The program requires that the name of each book (a character array), its author's name (another character array), the edition number (an integer), the price of the book (a float) be entered and stored. A multidimensional array cannot be used to do this, as an array must be of the same data type. This is where a structure makes the things simpler.

A structure consists of a number of data items, which need not be of the same type, grouped together. In the above example, a structure would consist of the book's name, the author name, the edition number, and the price of the book. The structure could hold as many of these items as desired.

19.1.1 Defining a Structure

A **structure definition** forms a template for creating structure variables. The variables in the structure are called **structure elements** or **structure members**.

Generally, the elements in a structure are logically related because they refer to a single entity. The book catalog example can be represented as follows:

```
struct cat
{
    char bk_name [25];
    char author [20];
    int edn;
    float price;
};
```

The above statement defines a new data type called `struct` **`cat`**. Each variable of this type consists of four elements - **`bk_ name`**, **`author`**, **`edn`**, and **`price`**. The statement does not declare any variable, and so it does not set aside any storage in memory. It just defines the structure of cat. The keyword `struct` tells the compiler that a structure is being defined. The tag **`cat`** is not a variable name, since a variable is not being declared. It is a **type name**. The elements of the structure are defined within braces, and a semicolon terminates the entire statement.

19.1.2 Declaring Structure Variables

Once the structure has been defined, one or more variables of that type can be declared. This can be done as follows:

```
struct cat books1;
```

This statement will set aside enough memory to hold all items in the structure. The above declaration performs a function similar to variable declarations like `int` **`xyz`** and `float` **`ans`**. It tells the compiler to set aside storage for a variable of specific type and assigns a name to the variable.

As with `int`, `float` and other data types there can be any number of variables of a given structure type. In a program, two variables **`books1`** and **`books2`** of the structure type **`cat`** can be declared. This can be done in several ways.

```
struct cat
{
    char bk_name[25];
    char author[20];
    int edn;
    float price;
} books1, books2;
```

or

```
struct cat books1, books2;
```

or

```
struct cat books1;
struct cat books2;
```

These declarations set aside memory for both **`books1`** and **`books2`**.

Individual structure elements are referenced through the use of the **dot operator (.)**, which is also known as the **membership operator**. The general syntax for accessing a structure element is:

```
structure_name.element_name
```

For example, the following code refers to the field **`bk_name`** of the structure variable **`books1`** declared earlier.

```
books1.bk_name
```

To read in the name of the book, the code would be:

```
scanf("%s", books1.bk_name);
```

To print the book name, the code would be:

```
printf("The name of the book is %s", books1.bk_name);
```

19.1.3 Initializing Structures

Like variables and arrays, structure variables can be initialized at the point of declaration. The format is similar to the one used to initialize arrays. Consider the following structure that stores the employee number and name:

```
struct employee
{
    int no;
    char name [20];
};
```

Variables **emp1** and **emp2** of the type **employee** can be declared and initialized as:

```
struct employee emp1 = {346, "Abraham"};
struct employee emp2 = {347, "John"};
```

Here, after the usual declaration of the structure type, the two structure variables **emp1** and **emp2** are declared and initialized. Their declaration and initialization happens at the same time through a single line of code. The initialization of a structure is similar to that of an array – the variable type, the variable name, and the assignment operator followed by braces enclosing a list of values, with the values separated by commas.

19.1.4 Assignment Statements used with Structures

It is possible to assign the values of one structure variable to another variable of the same type using a simple assignment statement. That is, if **books1** and **books2** are structure variables of the same type, the following statement is valid.

```
books2 = books1;
```

Also in cases, where direct assignment is not possible, the in-built function `memcpy()` can be used. The prototype for this function is

```
memcpy (char * destn, char &source, int nbytes);
```

This function copies **nbytes** starting from the address source to a block **nbytes** bytes starting from the address **destn**. The function requires the user to specify the size of the structure (**nbytes**), which

can be obtained by the `sizeof()` operator. Using `memcpy()`, the contents of **`books1`** can be copied to **`books2`** as follows:

```
memcpy (&books2, &books1, sizeof(struct cat));
```

19.1.5 Structures within Structures

It is possible to have one structure within another structure. However, a structure cannot be nested within itself. Having one structure within another is many times a practical requirement. Consider an example, where, a record of the person borrowing the book and details of the books borrowed has to be maintained. The following structure can be used for the same.

```
struct issue
{
    char borrower [20];
    char dt_of_issue[8];
    struct cat books;
}issl;
```

This declares **`books`** to be a component of the structure **`issue`**. This component itself is a structure of type `struct` **`cat`**. The structure can be initialized as:

```
struct issue issl = { "Jane", "04/22/03", {"Illusions",
"Richard Bach", 2, 150.00}};
```

Nested braces are used to initialize a structure within a structure.

To access the elements of the structure the format will be similar to the one used with normal structures, that is to access name of borrower, the code will be

```
issl.borrower
```

However to access elements of the structure **`cat`**, which is a part of another structure **`issue`**, the following expression will be used:

```
issl.books.author
```

This refers to the element author in the structure books in the structure **`issl`**.

The level for nesting structures is restricted only by the availability of memory. It is possible to have a structure within a structure within a structure and so on. The variable names used are usually self descriptive of the form.

Concepts

For example,

```
company.division.employee.salary
```

Also remember that if a structure is nested within another, it has to be declared prior to the structure that uses it.

19.1.6 Passing Structures as Arguments

A structure variable can be passed as an argument to a function. This is a useful facility and it is used to pass groups of logically related data items together instead of passing them one by one. However, when a structure is used as an argument, care should be taken that the type of the argument matches the type of the parameter.

For example, a structure is declared for storing the customer name, number and the principal amount deposited by the customer. The data is accepted in the `main()` and the structure is passed to a function **`intcal()`** that calculates the payable interest. The code is as follows:

Example 1:

```
#include <stdio.h>
void main()
{
    struct strucintcal/* Defines the structure */
    {
        char name[20];
        int numb;
        float amt;
    }xyz; /* Declares a variable */
    void intcal(struct strucintcal);
    clrscr();
    /* Accepts data into the structure */
    printf("\nEnter Customer name: ");
    gets(xyz.name);
    printf("\nEnter Customer number: ");
    scanf("%d",&xyz.numb);
    printf("\nEnter Principal amount: ");
    scanf("%f", &xyz.amt);
    intcal(xyz); /* Passes the structure to a function */
    getch();
}
```

```
void intcal(struct { char name[20];
                     int numb;
                     float amt;
                   } abc)
{
    float si, rate = 5.5, yrs = 2.5;
    /* Computes the interest */
    si = (abc.amt * rate * yrs) / 100;
    printf ("\nThe customer name is %s",abc.name);
    printf("\nThe customer number is %d",abc.numb);
    printf("\nThe amount is %f",abc.amt);
    printf("\nThe interest is %f",si);
    return;
}
```

A sample output of the program is given below.

```
Enter Customer name: Jane
Enter Customer number: 6001
Enter Principal Amount: 30000
The customer name is Jane
The customer number is 6001
The amount is 30000.000000
The interest is 4125.000000
```

It is possible to define a structure without a tag. This is useful when the variable is declared along with the structure definition itself. The tag is not needed in such situations.

19.1.7 Array of Structures

One of the most common uses of structures is in arrays of structures. To declare an array of structures, a structure is first defined, and then an array variable of that type is declared. For example, to declare an array of structures of the type **cat**, the statement would be:

```
struct cat books[50];
```

Like all array variables, arrays of structures begin indexing at 0. The array name followed by its subscript enclosed in square brackets stands for an element of that array. After the above declaration, this element is a structure by definition. So all the rules of referencing fields apply thereafter. After the structure array **books** is declared,

Concepts

```
books[4].author
```

will refer to the variable author of the fourth element of the array **books**.

19.1.8 Initialization of Structure Arrays

An array of any type is initialized by enclosing the list of values of its elements within a pair of braces. This rule applies even when the elements are structures. The effective initialization contains nested braces. Consider the following example,

```
struct unit
{
    char ch;
    int i;
};
struct unit series [3] =
{
    {'a', 100}
    {'b', 200}
    {'c', 300}
};
```

This declares **series** to be an array of structures, each of the type **unit**. While initializing, each element is initialized as a structure. The whole list is then enclosed within braces to indicate that the array is being initialized.

19.1.9 Pointers to Structures

C supports pointers to structures but there are some special aspects to structure pointers. Like other pointers, structure pointers are declared by placing an asterisk(*) in front of the structure variable's name. For example, the following statement declares pointer **ptr_bk** of the structure type **cat**.

```
struct cat *ptr_bk;
```

Now to assign the address of structure variable books of type **cat**, the statement would be:

```
ptr_bk = &books;
```

The -> operator is used to access the elements of a structure using a pointer. This operator is a combination of the minus (-) and the greater than (>) signs and is also known as the **combination operator**. For example, the field **author** can be accessed in any of the following ways:

```
ptr_bk->author
```

or

```
books.author
```

or

```
(*ptr_bk).author
```

In the last expression, the parentheses are required because the period operator (.) has a higher precedence than the indirection (*) operator. Without the parentheses the compiler would generate an error, because **ptr_bk** (a pointer) is not directly compatible with the period operator.

As with all pointer declarations, the declaration of a pointer allocates space for the pointer and not what it points to. So, when a structure pointer is declared, space is allocated for the address of the structure and not for the structure itself.

19.1.10 Structure Pointers as Arguments

Structure pointers are useful as arguments to functions. At the time of calling the function, a pointer to the structure or the explicit address of a structure variable is passed to the function. This enables the function to modify the structure elements directly.

19.2 The typedef keyword

A new data type name can be defined by using the keyword `typedef`. This does not create a new data type, but defines a new name for an existing type. The general syntax of the `typedef` statement is:

```
typedef type name;
```

where **type** is any allowable data type and **name** is the new name for this type.

The new name defined, is in addition to, and not a replacement for, the existing data type. For example, a new name for `float` can be created in the following way:

```
typedef float deci;
```

This statement will tell the compiler to recognize **deci** as another name for `float`. A `float` variable can be declared using **deci** as shown below:

```
deci amt;
```

Here, **amt** is a floating point variable of type **deci**, which is another name for float. After it has been defined, **deci** can be used as a data type in a **typedef** statement to assign another name to float. For example,

```
typedef deci point;
```

The above statement tells the compiler to recognize point as another name for **deci**, which is another name for float. The typedef feature is particularly convenient when defining structures, since it eliminates the need to repeatedly write struct tag whenever a structure is referenced. As a result, the structure can be referenced more concisely. In addition, the name given to a user defined structure type often suggests the purpose of the structure within the program. In general terms, a user-defined structure can be written as:

```
typedef struct new_type
{
    type var1;
    type var2;
}
```

Here, **new_type** is the user-defined structure type and not a structure variable. Structure variables can now be defined in terms of the new data type. An example is:

```
typedef struct
{
    int day;
    int month;
    int year;
} date;
date due_date;
```

Here, **date** is the new data type and **due_date** is a variable of type **date**.

Remember that **typedef** cannot be used with storage classes.

19.3 Sorting Arrays

Sorting means arranging the array data in a specified order such as ascending or descending. Data in an array is easier to search when the array is sorted.

There are several methods for sorting arrays. We will examine the following two methods:

- Bubble Sort
- Insertion Sort

19.3.1 Bubble Sort

The name of this sorting process describes the way it works. Here, the comparisons begin from the bottom-most element and the smaller element bubble up towards the top. The process of sorting a 5-element array in ascending order is given below:

- The value in the 5th element is compared against the value in the 4th element.
- If the value in the 5th element is smaller than the value in the 4th, the values in the two elements are swapped.
- Next, the value in the 4th element is compared against the value in the 3rd, and in a similar way, the values are swapped if the value in the lower element is found to be greater than that in the upper element.
- The value in the 3rd element is compared against the value in the 2nd, and this process of comparing and swapping continues.
- At the end of one pass, the smallest value reaches the first element. In symbolic terms, it can be stated that the smallest value has ‘bubbled’ up.
- In the next pass, the comparing starts off again with the lowest element, and works its way up to the 2nd element. Since the first element already contains the smallest value, it does not need to be compared.

In this way, at the end of the sorting process, the smaller elements bubble up towards the top, while the bigger values sink down. Figure 19.1 illustrates the bubble sort method.

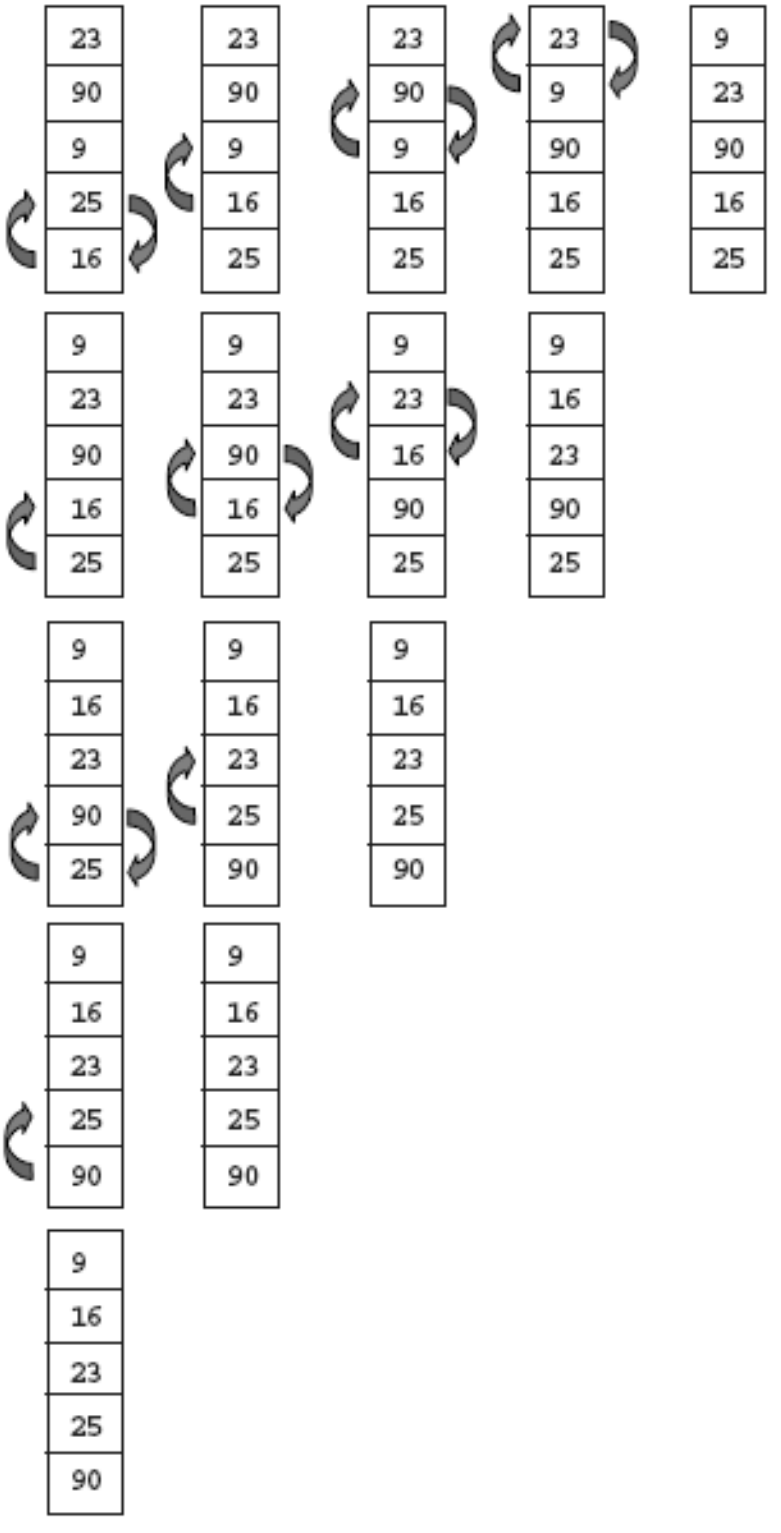


Figure 19.1: Bubble Sort

The program to perform the bubble sort is given here.

Example 2:

```
#include <stdio.h>
void main()
{
    int i, j, temp, arr_num[5] = { 23, 90, 9, 25, 16};
    clrscr();
    for(i=3;i>=0;i—) /* Tracks every pass */
        for(j=4;j>=4-i;j--) /* Compares elements */
        {
            if(arr_num[j]<arr_num[j-1])
            {
                temp=arr_num[j];
                arr_num[j]=arr_num[j-1];
                arr_num[j-1]=temp;
            }
        }
    printf(”\nThe sorted array”);
    for(i=0;i<5;i++)
    printf(”\n%d”, arr_num[i]);
    getch();
}
```

19.3.2 Insertion Sort

In the Insertion sort method, each element in the array is examined, and put into its proper place among the elements that have already been sorted. When the last element is put into its proper position, the array is sorted. For example, considering that an array has 5 elements,

- The value in the 1st element is assumed to be in sorted order.
- The value in the 2nd element is compared with the sorted part of the array that currently contains only the 1st element.
- If the value in the 2nd element is smaller, it is inserted before the 1st element. Now, the first two elements form the sorted list and the remaining form the unsorted list.
- The next element from the unsorted list, element 3, is then compared with the sorted list.
- If the value in the 3rd element is smaller than the 1st element, the value in the 3rd element is inserted before the 1st element.

Concepts

- Else, if the value in the 3rd element is smaller than the 2nd element, the value in the 3rd element is inserted before the 2nd element. Now, the sorted part of the array contains 3 elements while the unsorted part contains 2 elements.

- The process of comparing the elements in the unsorted part with those in the sorted part continues till the last element of the array has been compared.

At the end of the sort process, each element has been inserted in its proper location. Figure 19.2 illustrates the working of insertion sort.

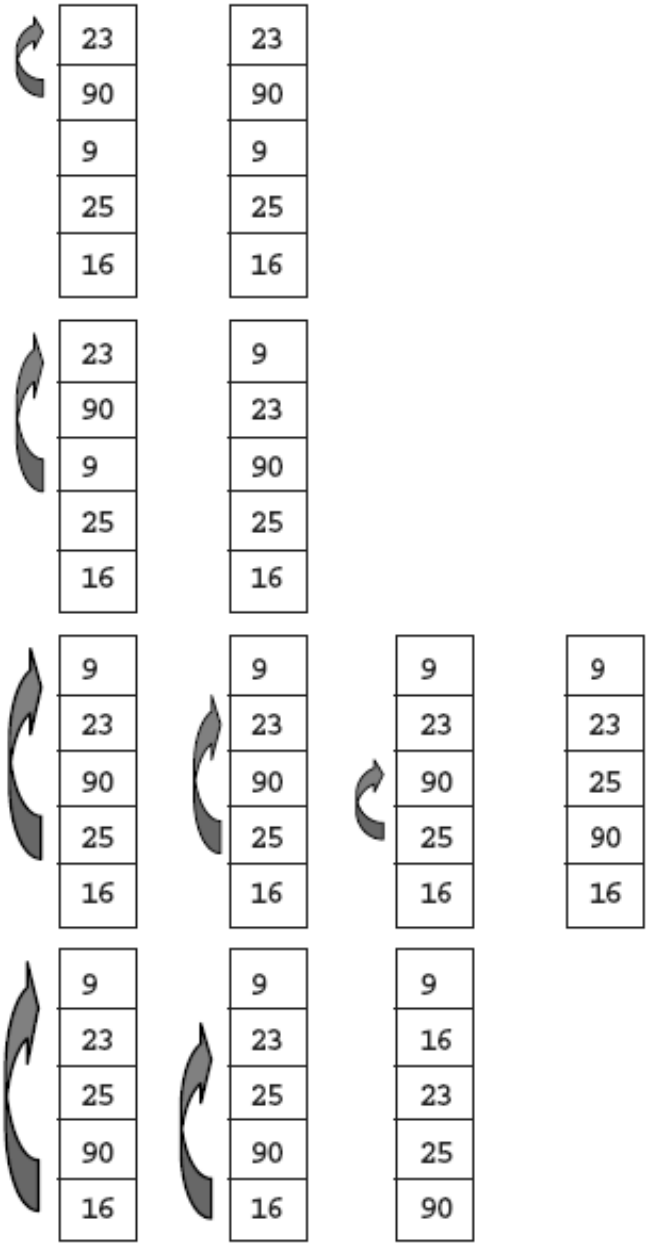


Figure 19.2: Insertion Sort

The program to perform the insertion sort is given below.

Example 3:

```
#include<stdio.h>
void main()
{
    int i, j, arr[5] = { 23, 90, 9, 25, 16 };
    char flag;
    clrscr();
    /*Loop to compare each element of the unsorted part of the array*/
    for(i=1; i<5; i++)
    /*Loop for each element in the sorted part of the array*/
        for(j=0, flag='n'; j<i && flag=='n'; j++)
        {
            if(arr[j]>arr[i])
            {
                /*Invoke the function to insert the number*/
                insertnum(arr, i, j);
                flag='y';
            }
        }
    printf("\n\nThe sorted array\n");
    for(i=0; i<5; i++)
        printf("%d\t", arr[i]);
    getch();
}
insertnum(int arrnum[], int x, int y)
{
    int temp;
    /*Store the number to be inserted*/
    temp=arrnum[x];
    /*Loop to push the sorted part of the array down from the position*/
    /*where the number has to inserted*/
    for(; x>y; x--)
        arrnum[x]=arrnum[x-1];
    /*Insert the number*/
    arrnum[x]=temp;
}
```

Concepts

Summary

- A structure is a grouping of variables of different data types under one name.
- A structure definition forms a template that may be used to create structure variables.
- Individual structure elements are referenced using the dot operator (.), which is also known as the membership operator.
- Values of one structure variable can be assigned to another variable of the same type using a simple assignment statement.
- It is possible to have one structure within another structure. However a structure cannot be nested within itself.
- A structure variable can be passed as an argument to another function.
- The most common implementation of structures is in the form of arrays of structures.
- The -> operator is used to access the elements of a structure using a pointer to that structure.
- A new data type name can be defined by using the keyword typedef.
- Two techniques for sorting an array are bubble sort and insertion sort.
- In bubble sort, the values of the elements are compared with the value in the adjacent element. In this method, the smaller elements bubble up, and at the end the array is sorted.
- In insertion sort, each element in the array is examined, and inserted into its proper place among the elements that have already been sorted.

Check Your Progress

1. A _____ groups together a number of data items, which need not be of the same data type.

2. Individual structure elements are referenced through the use of the _____.

3. Values of one structure variable can be assigned to another variable of the same type using a simple assignment statement. **(True / False)**

4. It is impossible to have one structure within another structure. **(True / False)**

5. A new data type name can be defined by using the _____keyword.

6. In bubble sort, the _____elements are compared.

7. In insertion sort, if an unsorted element has to be put in a particular sorted location, values are swapped. **(True / False)**

Concepts

Try It Yourself

1. Write a C program to implement an inventory system. Store the item number, name, rate and quantity on hand in a structure. Accept the details for five items into a structure array and display the item name and its total price. At the end, display the grand total value of the inventory.

2. Write a C program to store the names and scores of 5 students in a structure array. Sort the structure array in descending order of scores. Display the top 3 scores.

Session

20 Advanced Data Types & Sorting (Lab)

Objectives

At the end of this session, you will be able to:

- *Use structures and structure arrays*
- *Pass structures to functions*
- *Sort arrays*

The steps given in this session are detailed comprehensive and carefully thought through. This has been done so that the learning objectives are met and the understanding of the tool is complete. Please follow the steps carefully.

Part I – For the first 1 Hour and 30 Minutes:

20.1 Structures

A structure is a grouping of a number of data items, which may be different data types. Every structure has to be defined before it is declared. A structure definition can contain another structure. Initialization of structures is similar to that of arrays.

20.1.1 Structure arrays and Sorting

In C, it is possible to create a structure array. As with arrays, data in structure arrays can be sorted using Selection sort or Bubble Sort. Let us write a C program to implement a basic library management system. The system maintains a book catalog and a register for books issue and receipt transactions. Using the system, it is possible to add book details, record issue/receipt transactions and sort the issue/receipt register. The steps for creating the system are listed below:

1. Define a structure to store the book details.

The code will be,

```
struct book_st {
  int book_cd;
  char book_nm[30];
  char author[30];
  int copies;
};
```

2. Define a structure to store the issue/receipt register. Note that the issue/receipt date will also be a structure, and will have to be defined too.

The code will be,

```
struct date_st {
  int month;
  int day;
  int year;
};
struct tran_st {
  int book_code;
  char tran_type;
  struct date_st tran_dt;
};
```

3. Declare variables of the two structure types. For practical purposes, let us assume that details of 5 books and 10 transactions will be recorded.

The code will be,

```
struct book_st books[5];
struct tran_st trans[10];
```

4. Set up a loop to display a menu for the possible operations.

The code for the same will be,

```
while(choice!=4)
{
  clrscr();
  printf("\nSelect from Menu\n1. Add book names\n2. Record Issue/Return\
n3. Sort Transactions\n4. Exit\n\nEnter choice: ");
  scanf("%d", &choice);
  :
  :
}
```

5. If the selected operation is adding book details, accept the details within a loop.

The code will be,

```
for(i=0; i<5 && addflag=='y'; i++)
{
    books[i].book_cd = i+1;
    printf("\n\nBook code: %d\n\nBook name: ", i+1);
    scanf("%s",books[i].book_nm);
    printf("\nAuthor: ");
    scanf("%s",books[i].author);
    printf("\nNumber of copies: ");
    scanf("%d", &books[i].copies);
    printf("\n\nContinue? (y/n): ");
    scanf("%c", &addflag);
}
```

6. If the selected operation is adding transactions, set up a loop to accept the details.

The code will be,

```
for(i=0; i<10 && addflag=='y'; i++)
{
   printf("\n\nBook code:");
   scanf("%d", &trans[i].book_code);
   printf("\nIssue or Return?(I/R):");
   scanf("%c", &trans[i].tran_type);
   printf("\nDate:");
   scanf("%d %d %d", &trans[i].tran_dt.month, &trans[i].tran_dt.day,
&trans[i].tran_dt.year);
   printf("\n\nContinue? (y/n):");
   scanf("%c", &addflag);
}
```

7. If the selected operation is sorting transactions, pass the structure array to a function. The function should sort the array on book codes using selection sort method.

The code will be,

```
for(i=0; i<10; i++)
   for(j=i+1;j<10;j++)
   {
      if(tran[i].book_code > tran[j].book_code)
      {
         temptran=tran[i];
         tran[i]=tran[j];
         tran[j]=temptran;
      }
   }
```

8. Display the number of transactions for every book in the sort function.

The code will be,

```
for(i=0, j=0; i<10; j=0)
{
    tempcode=tran[i].book_code;
    while(tran[i].book_code==tempcode && i<10)
    {
       j++;
       i++;
    }
    printf(”\nBook code %d had %d transactions”, tempcode, j);
}
```

Let us look at the complete program.

1. Invoke the editor in which you can type the C program.

2. Create a new file.

3. Type the following code:

```
#include<stdio.h>
struct book_st
{
    int book_cd;
    char book_nm[30];
    char author[30];
    int copies;
};
struct date_st
{
    int month;
    int day;
    int year;
};
struct tran_st
{
    int book_code;
    char tran_type;
    struct date_st tran_dt;
};
void main()
{
    int choice=1, i;
    char addflag;
    struct book_st books[5];
    struct tran_st trans[10];
    while(choice!=4)
    {
        clrscr();
        printf("\nSelect from Menu\n 1. Add book names\n 2. Record
Issue/Return\n 3. Sort Transactions\n 4. Exit\n\n Enter choice: ");
        scanf("%d", &choice);
        if(choice==1)
        {
            addflag='y';
            clrscr();
            for(i=0; i<5 && addflag=='y'; i++)
            {
                books[i].book_cd=i+1;
```

```
                printf("\n\nBook code: %d\n\nBook name: ", i+1);
                scanf("%s",books[i].book_nm);
                printf("\nAuthor: ");
                scanf("%s",books[i].author);
                printf("\nNumber of copies: ");
                scanf("%d", &books[i].copies);
                printf("\n\nContinue? (y/n): ");
                scanf("%c", &addflag);
        }
    }
    else if(choice==2)
    {
        addflag='y';
        clrscr();
        for(i = 0; i<10 && addflag == 'y'; i++)
        {
            printf("\n\nBook code:");
            scanf("%d", &trans[i].book_code);
            printf("\nIssue or Return?(I/R):");
            scanf("%c", &trans[i].tran_type);
            printf("\nDate:");
            scanf("%d %d %d", &trans[i].tran_dt.month,&trans[i].tran_
dt.day, &trans[i].tran_dt.year);
            printf("\n\nContinue? (y/n):");
            scanf("%c", &addflag);
        }
    }
    else if(choice==3)
    {
        sorttran(trans);}
    }
}
sorttran(struct tran_st tran[10])
{
    int i, j, tempcode;
    struct tran_st temptran;
    clrscr();
    for(i=0;i<10;i++)
        for(j = i+1; j < 10; j++)
```

```
    {
        if(tran[i].book_code>tran[j].book_code)
        {
            temptran=tran[i];
            tran[i]=tran[j];
            tran[j]=temptran;
        }
    }
    for(i=0, j=0;i<10;j=0)
    {
        tempcode=tran[i].book_code;
        while(tran[i].book_code==tempcode && i<10)
        {
            j++;
            i++;
        }
        printf("\nBook code %d had %d transactions", tempcode, j);
    }
    getch();
}
```

To see the output, follow these steps:

4. **Save the file with the name `structI.C`.**

5. **Compile the file, `structI.C`.**

6. **Execute the program, `structI.C`.**

7. **Return to the editor.**

The sample output of the program is given below.

```
Select from Menu
1. Add book names
2. Record Issue/Return
3. Sort Transactions
4. Exit
Enter choice:
```

If 1 is entered, the sample output of the program will be as shown below.

```
Book code: 1
Book name: Detective
Author: Hailey
Number of copies: 3
Continue? (y/n): y
```

If 2 is entered, the sample output of the program will be as shown below.

```
Book code: 1
Issue or Return? (I/R): I
Date: 2 22 03
Continue? (y/n): y
```

If 3 is entered, the sample output of the program will be as shown below.

```
Book code 1 had 3 transactions
Book code 2 had 1 transactions
Book code 3 had 2 transactions
Book code 4 had 0 transactions
Book code 5 had 4 transactions
```

Part II – For the next 30 Minutes:

1. Write a C Program to store student data in a structure. The data should include student ID, name, course registered for, and year of joining. Write a function to display the details of students enrolled in a specified academic year. Write another function to locate and display the details of a student based on a specified student ID.

   To do this,

   a. Define the structure to store the student details.

   b. Declare and initialize the structure with details of 10 students.

   c. Set a loop to display a menu for the operations to be performed.

   d. Accept the menu choice and invoke appropriate functions with the structure array as parameter.

   e. In the function to display students for a year, accept the year. Set a loop to check each student's enrollment year, and display if it matches. At the end, allow the user to specify another year.

   f. In the function to locate student details, accept the student ID. Set a loop to check each student's ID, and display if it matches. At the end, allow the user to specify another student ID.

Try It Yourself

1. Write a C program to store 5 lengths in a structure array. The lengths should be in the form of yards, feet and inches. Sort and display the lengths.

2. Write a C program to store employee details in a structure array. The data should include employee ID, name, salary, and date of joining. The date of joining should be stored in a structure. The program should perform the following operations based on a menu selection:

   a. Increase the salaries according to the following rules:

Salary Range	Percentage increase
<= 2000	15 %
> 2000 and <= 5000	10 %
>5000	No increase

   b. Display the details of employees who complete 10 years with the company.

Session

21 File Handling

Concepts

Objectives

At the end of this session, you will be able to:

- *Explain streams and files*
- *Discuss text streams and binary streams*
- *Explain the various file functions*
- *Explain file pointer*
- *Discuss current active pointer*
- *Explain command-line arguments*

Introduction

Most programs need to read and write data to disk-based storage systems. Word processors need to store text files, spreadsheets need to store the contents of cells, and databases need to store records. This session explores the facilities in C for input and output (I/O) to a disk system.

The C language does not contain any explicit I/O statements. All I/O operations are carried out using functions from the standard C library. This approach makes the C file system very powerful and flexible. I/O in C is unique because data may be transferred in its internal binary representation or in a human-readable text format. This makes it easy to create files to fit any need.

It is important to understand the difference between files and streams. The C I/O system provides an interface to the user, which is independent of the actual device being accessed. This interface is not actually a file but an abstract representation of the device. This abstract interface is known as a stream and the actual device is called the file.

21.1 File Streams

The C file system works with a wide variety of devices including printers, disk drives, tape drives and terminals. Even though all these devices are very different from each other, the buffered file system transforms each device into a logical device called a stream. Since all streams act similarly, it is easy to handle the different devices. There are two types of streams-the text and binary streams.

Concepts

21.1.1 Text Streams

A text stream is a sequence of characters. The text streams can be organized into lines terminated by a new line character. However, the new line character is optional on the last line and is determined by the implementation. Most C compilers do not terminate text streams with new line characters. In a text stream, certain character translations may occur as required by the environment. For example, a new line may be converted to a carriage return/linefeed pair. Therefore, there may not be a one-to-one relationship between the characters that are written (or read) and those in the external device. Also, because of possible translations, the number of characters written (or read) may not be the same as those in the external device.

21.1.2 Binary Streams

A binary stream is a sequence of bytes with a one-to-one correspondence to those in the external device, that is, there are no character translations. Also, the number of bytes written (or read) is the same as the number on the external device. Binary streams are a flat sequence of bytes, which do not have any flags to indicate the end of file or end of record. The end of file is determined by the size of the file.

21.2 File functions and FILE Structure

A file can refer to anything from a disk file to a terminal or a printer. However, all files do not have the same capabilities. For example, a disk file can support random access while a keyboard cannot. A file is associated with a stream by performing an open operation. Similarly, it is disassociated from a stream by a close operation. When a program terminates normally, all files are automatically closed. However, when a program crashes, the files remain open.

21.2.1 Basic File Functions

The ANSI file system is composed of several interrelated functions. The most common ones are listed in Table 21.1.

Name	Function
`fopen()`	Opens a file
`fclose()`	Closes a file
`fputc()`	Writes a character to a file
`fgetc()`	Reads a character from a file
`fread()`	Reads from a file to a buffer
`fwrite()`	Writes from a buffer to a file
`fseek()`	Seeks a specific location in the file
`fprintf()`	Operates like printf(), but on a file
`fscanf()`	Operates like scanf(), but on a file
`feof()`	Returns true if end-of-file is reached

Name	Function
`ferror()`	Returns true if an error has occurred
`rewind()`	Resets the file position locator to the beginning of the file
`remove()`	Erases a file
`fflush()`	Writes data from internal buffers to a specified file

Table 21.1: Basic File Functions

The above functions are contained in the header file stdio.h. This header file must be included in a program that makes use of the functions. Most of the functions are similar to the console I/O functions. The stdio.h header file also defines several macros useful for file processing. For example, the EOF macro defined as -1, contains the value returned when a function tries to read past the end of the file.

21.2.2 File Pointer

A file pointer is essential for reading or writing files. It is a pointer to a structure that contains information about the file. The information includes the file name, current position of the file, whether the file is being read or written, and whether any errors or the end of the file have occurred. The user does not need to know the details, because the definitions obtained from stdio.h include a structure declaration called `FILE`. The only declaration needed for a file pointer is symbolized by:

```
FILE *fp;
```

This denotes that fp is a pointer to a `FILE`.

21.3 Text Files

There are various functions to handle text files. They are discussed below:

21.3.1 Opening a Text File

The `fopen()` function opens a stream for use and links a file with that stream. The file pointer associated with the file is returned by the `fopen()` function. In most cases, the file being opened is a disk file. The prototype for the `fopen()` function is:

```
FILE *fopen(const char *filename, const char *mode);
```

where `filename` is a pointer to a string of characters that make up a valid file name and can also include a path specification.

The string pointed to by mode determines how the file is to be opened. Table 21.2 shows the valid modes in which a file may be opened.

Mode	Meaning
r	Open a text file for reading
w	Create a text file for writing
a	Append to a text file
r+	Open a text file for read/write
w +	Create a text file for read/write
a+f	Append or create a text file for read/write

Table 21.2: File Opening Modes for Text Files

As can be seen from Table 21.2, the files can be opened in the text or the binary mode. A null pointer is returned if an error occurs when the `fopen()` function is opening a file. Note that strings like "`a+f`" can also be represented as "`af+`".

If a file xyz were to be opened for writing, the code for it would be:

```
FILE *fp;
fp = fopen ("xyz", "w");
```

However, a file is generally opened using the set of statements similar to the following:

```
FILE *fp;
if ((fp = fopen ("xyz", "w")) == NULL)
{
    printf("Cannot open file");
    exit (1);
}
```

The macro NULL is defined in stdio.h as '`\0`'. If a file is opened using the above method, `fopen()` detects any error in opening a file, such as a write-protected or a full disk, before attempting to write to it. A null is used to indicate failure because no file pointer will ever have that value.

If a file is opened for writing, any file with the same name and is already open will be overwritten. This is because when a file is opened in a write mode, a new file is created. If records have to be added to an existing file, it should be opened with the mode "`a`". If a file is opened in the read mode and it does not exist, an error is returned. If a file is opened for read/ write operations, it will not be erased if it exists. However, if it does not exist, it will be created.

According to the ANSI standard, eight files can be opened at any one time. However, most C compilers and environments allow more than eight files to be opened.

21.3.2 Closing a Text File

Since there is a limit on the number of files that can be opened at one time, it is important to close a file once it has been used. This frees system resources and reduces the risk of overshooting the set limit. Closing a stream also flushes out any associated buffer (an important operation that prevents loss of data) when writing to a disk. The `fclose()` function closes a stream that was opened by a call to `fopen()`. It writes any data still remaining in the disk buffer to the file. The prototype for `fclose()` is:

```
int fclose(FILE *fp);
```

where `fp` is the file pointer.

The function `fclose()` returns an integer value 0 for successful closure. Any other return value would indicate an error. The `fclose()` function will fail if a disk has been prematurely removed from the drive or there is no more space on the disk.

The other function used for closing streams is the `fcloseall()` function. This function is useful when many open streams have to be closed at the same time. It closes all open streams and returns the number of streams closed or EOF if any error is detected. It can be used in the following manner:

```
fcl = fcloseall();
if (fcl == EOF)
    printf("Error closing files");
else
    printf("%d file(s) closed ", fcl);
```

21.3.3 Writing a Character

Streams can be written to a file either character by character or as strings. Let us first discuss writing characters to a file. The `fputc()` function is used for writing characters to a file previously opened by `fopen()`. The prototype for this is:

```
int fputc(int ch, FILE *fp);
```

where `fp` is the file pointer returned by `fopen()` and `ch` is the character to be written.

Even though `ch` is declared as an `int`, it is converted by `fputc()` into an unsigned char. The `fputc()` function writes a character to a specified stream at the current file position and then advances the file position indicator. If `fputc()` is successful it returns the character written, otherwise it returns EOF.

Concepts

21.3.4 Reading a Character

The `fgetc()` function is used for reading characters from a file opened in read mode, using `fopen()`. The prototype for `fgetc()` is:

```
int fgetc (FILE *fp);
```

where `fp` is a file pointer of type `FILE`, returned by `fopen()`.

The `fgetc()` function returns the next character from the current position in the input stream, and increments the file position indicator. The character read is an unsigned char and is converted to an integer. If the end of file is reached, `fgetc()` returns `EOF`.

To read a text file until the end of file is encountered, the code will be:

```
do
{
    ch = fgetc(fp);
} while (ch != EOF);
```

The following program uses the functions discussed till now. It takes in characters from the keyboard and writes them to a file till the character '@' is entered by the user. After the user has entered the information, the program displays the file contents on the screen.

Example 1:

```
#include <stdio.h>
main()
{
    FILE *fp;
    char ch= ' ';
    /* Writing to file JAK */
    if ((fp=fopen("jak", "w"))==NULL)
    {
        printf("Cannot open file \n\n");
        exit(1);
    }
    clrscr();
    printf("Enter characters (type @ to terminate): \n");
    ch = getche();
```

```
    while (ch !='@')
    {
        fputc(ch, fp) ;
        ch = getche();
    }
    fclose(fp);
    /* Reading from file JAK */
    printf("\n\nDisplaying contents of file JAK\n\n");
    if((fp=fopen("jak", "r"))==NULL)
    {
        printf("Cannot open file\n\n");
        exit(1);
    }
    do
    {
        ch = fgetc (fp);
        putchar(ch) ;
    } while (ch!=EOF);
    getch();
    fclose(fp);
}
```

A sample run for the above will be:

```
Enter Characters (type @ to terminate):
This is the first input to the File JAK@
Displaying Contents of File JAK
This is the first input to the File JAK
```

21.3.5 String I/O

In addition to `fgetc()` and `fputc()`, C supports the related functions `fputs()` and `fgets()`. These functions write and read character strings to and from a disk file.

The prototypes for these functions are as follows:

```
int fputs(const char *str, FILE *fp);
char *fgets( char *str, int length, FILE *fp);
```

The `fputs()` function works like `fputc()`, except that it writes the entire string to the specified stream.

It returns EOF if an error occurs.

The `fgets()` function reads a string from the specified stream until either a new line character is read or length-1 characters have been read. If a new line is read, it is considered as a part of the string (unlike `gets()`). The resultant string will be terminated by the null character. The function returns a pointer to the string if successful and a null pointer if an error occurs.

21.4 Binary Files

The functions used to handle binary files are same as the ones used to handle text files. However the opening modes of the `fopen()` functions are different in the case of binary files.

21.4.1 Opening a binary file

The following table lists the various modes for the `fopen()` function in case of binary files.

Mode	Meaning
`rb`	Open a binary file for reading
`w b`	Create a binary file for writing
`ab`	Append to a binary file
`r+b`	Open a binary file for read/write
`w + b`	Create a binary file for read/write
`a+b`	Append a binary file for read/write

Table 21.3: File Opening Modes for Binary Files

If a file **`xyz`** were to be opened for writing, the code for it would be:

```
FILE *fp;
fp = fopen ("xyz", "wb");
```

21.4.2 Closing a binary file

The `fclose()` function can also be used to close a binary files in addition to text files. An example of `fclose()` is:

```
int fclose(FILE *fp);
```

where `fp` is a file pointer that points to an open file.

21.4.3 Writing a binary file

Some applications involve the use of data files to store blocks of data, where each block consists of contiguous bytes. Each block will generally represent a complex data structure or an array.

For example, a data file may consist of multiple structures having the same composition, or it may contain multiple arrays of the same type and size. For such applications it may be desirable to read the entire block from the data file or write the entire block to the data file rather than reading or writing the individual components (i.e. structure members or array elements) within each block separately.

The `fwrite()` function is used to write data to data under such circumstances. This function can be used to write any type of data. The prototype for the `fwrite()` function is:

```
size_t fwrite(const void *buffer, size_t num_bytes, size_t count, FILE *fp);
```

The data type `size_t` is an ANSI C addition to improve portability. It is predefined as an integer type large enough to hold `sizeof()` results. For most systems it can be taken as an unsigned integer.

The buffer is a pointer to the information that will be written to the file. The number of bytes to be read or written is specified by **`num_bytes`**. The argument count determines how many items (each **`num_bytes`** in length) are read or written. Finally, `fp` is a file pointer to a previously opened stream. Files opened for these operations should be in binary mode.

This function returns the number of objects written to the file if the write operation was successful. If this value is less than **`num`** then an error has occurred. The `ferror()` function (that will be discussed shortly) can be used to determine the problem.

21.4.4 Reading a binary file

The `fread()` function can be used to read any type of data. The prototype for the same is:

```
size_t fread(void *buffer, size_t num_bytes, size_t count FILE *fp);
```

The **`buffer`** is a pointer to the region of memory that will receive the data from the file. The number of bytes to be read or written is specified by **`num_bytes`**. The argument count determines how many items (each `num_bytes` in length) are read or written. Finally, `fp` is a file pointer to a previously opened stream. Files opened for these operations should be in binary mode.

This function will return the number of objects read if the read operation is successful. It returns 0 if either end of file is reached or an error has occurred. The `feof()` and the `ferror()` functions (that will be discussed shortly) can be used respectively to determine the problem.

The `fread()` and `fwrite()` functions are often referred to as unformatted read or write functions.

As long as the file has been opened for binary operations, `fread()` and `fwrite()` can read and write any type of information. For example, the following program writes and then reads back a double, an `int` and a long value to and from a disk file. Notice how it uses the `sizeof()` function to determine the length of each data type.

Example 2:

```
#include <stdio.h>
main ()
{
    FILE *fp;
    double d = 23.31 ;
    int i = 13;
    long li = 1234567L;
    clrscr();
    if ( ( fp= fopen (“jak”, “wb+”)) == NULL )
    {
        printf(“cannot open file “);
        exit(1);
    }
    fwrite (&d, sizeof(double), 12, fp);
    fwrite (&i, sizeof(int), 1, fp);
    fwrite (&li, sizeof(long), 1,fp);
    fclose (fp);
    if ((fp= fopen (“jak”, “rb+”)) == NULL )
    {
        printf(“cannot open file “);
        exit(1);
    }
    fread (&d, sizeof(double), 1, fp);
    fread(&i, sizeof(int), 1, fp);
    fread (&li, sizeof(long), 1, fp);
    printf (“%f %d %ld”, d, i, li);
    fclose (fp);
}
```

As this program illustrates, the buffer can be read and often is simply the memory used to hold a variable. In this simple program, the return value of `fread()` and `fwrite()` are ignored. These values should however, be checked for errors for efficient programming.

One of the most useful applications of `fread()` and `fwrite()` involves reading and writing user-defined

data types, especially structures. For example given the structure:

```
struct struct_type
{
    float balance;
    char name[80];
} cust;
```

The following statement writes the contents of **cust** to the file pointed to by **fp**.

```
fwrite(&cust, sizeof(struct struct_type), 1, fp);
```

21.5 File Handling Functions

The other file handling functions are discussed in this section.

21.5.1 Using 'feof()'

When a file is opened for binary input, an integer value equal to the EOF may be read. The input routine will indicate an end of file in such a case, even though the physical end of the file has not reached. A function `feof()` can be used in such cases. The prototype of this function is:

```
int feof(FILE *fp );
```

It returns true if the end of the file has been reached, otherwise it returns false (`0`). This function is used while reading binary data.

The following code segment reads a binary file till the end of file is encountered.

```
.
.
while (!feof(fp))
    ch = fgetc(fp);
.
.
```

21.5.2 The 'rewind()' function

The `rewind()` function resets the file position indicator to the beginning of the file. It takes the file pointer as its argument.

The syntax for `rewind()` is:

```
rewind(fp);
```

The following program opens a file in write/read mode, takes strings as input using `fgets()`, rewinds the file and then displays the same strings using `fputs()`.

Example 3:

```
#include <stdio.h>
main()
{
    FILE *fp;
    char str [80];
    /* Writing to File JAK */
    if ( ( fp=fopen (”jak”, ”w+”) ) == NULL)
    {
        printf ( ”Cannot open file \n\n”);
        exit(1);
    }
    clrscr ();
    do
    {
        printf (”Enter a string (CR to quit): \n”);
        gets (str);
        if(*str != ’\n’)
        {
            strcat (str, ”\n”); /* add a new line */
            fputs (str, fp);
        }
    } while (*str != ’\n’);
    /*Reading from File JAK */
    printf (”\n\n Displaying Contents of File JAK\n\n”);
    rewind (fp);
    while (!feof(fp))
    {
        fgets (str, 81, fp);
        printf (”\n%s”, str);
    }
    fclose(fp);
}
```

A sample run for the above program is:

```
Enter a string (CR to quit):
This is input line 1
Enter a string (CR to quit) :
This is input line 2
Enter a string (CR to quit):
This is input line 3
Enter a string (CR to quit):
Displaying Contents of File JAK
This is input line 1
This is input line 2
This is input line 3
```

21.5.3 The 'ferror()' function

The `ferror()` function determines whether a file operation has produced an error. Its prototype is:

```
int ferror(FILE * fp) ;
```

where `fp` is a valid file pointer. It returns true if an error has occurred during the last file operation; otherwise, it returns `false`.

As each operation sets the error condition, `ferror()` should be called immediately after each operation; otherwise, an error may be lost. The earlier program can be modified to check and warn about any errors in writing as given below.

```
...
do
{
    printf(" Enter a string (CR to quit): \n");
    gets(str);
    if(*str != '\n')
    {
        strcat (str, "\n"); /* add a new line */
        fputs (str, fp);
    }
    if(ferror(fp))
        printf("\nERROR in writing\n");
    } while(*str!='\n');
...
```

Concepts

21.5.4 Erasing Files

The `remove()` function erases a specified file. Its prototype is:

```
int remove (char *filename);
```

It returns `0` if successful else it returns a nonzero value.

As an example, consider the following code segment:

```
.
.
printf ("\nErase file %s (Y/N) ? ", file1);
ans = getchar ();
.
.
if(remove(file1))
{
    printf ("\nFile cannot be erased");
    exit(1);
}
```

21.5.5 Flushing streams

Often, the standard output file is buffered. This means that the output to the file is collected in memory but not actually displayed until the buffer is full. If the program crashes, some characters may still be in the buffer. The result is that the program appears to have terminated earlier than it actually did. The `fflush()` function resolves this problem. As the name suggests, it flushes out the buffer. The action of flushing depends upon the file type. A file opened for read will have its input buffer cleared, while a file opened for write will have its output buffer written to the files.

The prototype for this function is:

```
int fflush(FILE * fp);
```

The `fflush()` function will write the contents of any buffered data to the file associated with `fp`. The `fflush()` function, with a null, flushes all files opened for output. It returns `0` if successful, otherwise, it returns EOF.

Concepts

21.5.6 The Standard Streams

Whenever a C program starts execution under DOS, five special streams are opened automatically by the operating system. These five streams are:

- The standard input (`stdin`)
- The standard output (`stdout`)
- The standard error (`stderr`)
- The standard printer (`stdprn`)
- The standard auxiliary (`stdaux`)

The `stdin`, `stdout` and `stderr` are assigned by default to the system's console where as the `stdprn` is assigned to the first parallel printer port and `stdaux` is assigned to the first serial port. They are defined as fixed pointers of type `FILE`, so they can be used wherever the use of `FILE` pointer is legal. They can also effectively be transferred to other streams or device files whenever redirection is involved.

The following program prints the contents of the file onto the printer.

Example 4:

```
#include <stdio.h>
main()
{
    FILE *in;
    char buff[81], fname[13];
    clrscr();
    printf("Enter the Source File Name:");
    gets(fname);
    if((in=fopen(fname, "r"))==NULL)
    {
        fputs("\nFile not found", stderr);
        /* display error message on standard error rather than standard output */
        exit(1);
    }
    while(!feof(in))
    {
```

```
        if(fgets(buff, 81, in))
        {
            fputs(buff, stdprn);
            /* Send line to printer */
        }
    }
    fclose(in);
}
```

Note the use of the `stderr` stream with the `fputs()` function in the above program. It is used instead of the `printf()` function because the destination of the `printf()` function is the `stdout`, which can be redirected. If the output of a program was redirected and an error occurred during execution, then any error message given to the `stdout` stream would also be redirected. To avoid this, the `stderr` stream is used to display the error message on the screen because the destination of the `stderr` is also the console, but the `stderr` stream cannot be redirected. It always displays the message on the screen.

21.5.7 Current Active Pointer

In order to keep track of the position where I/O operations take place, a pointer is maintained in the `FILE` structure. Whenever a character is read from or written to the stream, the current active pointer (known as `curp`) is advanced. Most of the I/O functions refer to `curp`, and update it after the input or output procedures on the stream. The current location of the current active pointer can be found with the help of the `ftell()` function. The `ftell()` function returns a `long int` value that gives the position of `curp` from the beginning of the file in the specified stream. The prototype of the `ftell()` function is:

```
long int ftell(FILE *fp);
```

The following extract of a program displays the location of the current pointer on the stream `fp`.

```
printf("The current location of the file pointer is : %1d ", ftell(fp));
```

Setting Current Position

Immediately after opening the stream, the current active pointer position is set to zero and points to the first byte of the stream. As seen earlier, whenever a character is read from or written to the stream, the current active pointer is advanced. The pointer may be set to any position other than the current pointer at any point of time in a program. The `rewind()` function, sets the pointer position to the start of the program. Another function, which can be used to set the pointer position is `fseek()`.

The `fseek()` function repositions the `curp` by the specified number bytes from the start, the current position or the end of the stream depending upon the position specified in the `fseek()` function. The prototype of the `fseek()` function is

```
int fseek(FILE *fp, long int offset, int origin);
```

where `offset` is the number of bytes beyond the file location given by origin.

The origin indicates the starting position of the search and must have the value of either 0, 1 or 2, which represent three symbolic constants (defined in stdio.h) as shown in Table 21.4:

Origin	File location
SEEK_SET or 0	Beginning of file
SEEK_CUR or 1	Current file pointer position
SEEK_END or 2	End of file

Table 21.4: Symbolic Constants

A return value of zero means `fseek()` has been successful and a non-zero value means `fseek()` has failed.

The following code segment seeks 6th record in the file:

```
struct addr
{
    char name[40];
    char street[40];
    char city[40];
    char state[3];
    char pin[7];
}
FILE *fp;
.
.
.
fseek(fp,5L*sizeof(struct addr),SEEK_SET);
```

The `sizeof()` function is used to find the length of each record in terms of bytes. The return value is used to determine the number of bytes, to skip the first 5 records.

21.5.8 'fprintf()' and 'fscanf()'

In addition to the discussed I/O functions, the buffered I/O system also includes `fprintf()` and `fscanf()`. These functions are similar to `printf()` and `scanf()` except that they operate with files.

Concepts

The prototypes of `fprintf()` and `fscanf()` are:

```
int fprintf(FILE * fp, const char *control_string,..);
int fscanf(FILE *fp, const char *control_string,...);
```

where, `fp` is the file pointer returned by a call to `fopen()`.

The `fprintf()` and `fscanf()` functions direct their I/O operations to the file pointed to by `fp`. The following program code reads a string and an integer from the keyboard, writes them to a disk file, and then reads the information and displays it on the screen.

```
.
.
printf("Enter a string and a number: ");
fscanf(stdin, "%s %d", str, &no);
/* read from the keyboard */
fprintf(fp, "%s %d", str, no);
/* write to the file */
fclose (fp);
.
.
fscanf(fp, "%s %d", str, &no)
/* read from file */
fprintf(stdout, "%s %d", str, no)
/* print on screen */
.
.
```

Remember that, although `fprintf()` and `fscanf()` often are the easiest way to write and read assorted data to and from disk files, they are not always the most efficient. The reason being that extra overhead is incurred with each call, since the data is written in formatted ASCII data (as it would appear on the screen) instead of being written in binary format. So, if speed or file size is a concern, `fread()` and `fwrite()` are a better choice.

Summary

- The C language does not contain any explicit I/O statements. All I/O operations are carried out using functions from the standard C library.
- There are two types of streams - the text and binary streams.
- A text stream is a sequence of characters.
- A binary stream is a sequence of bytes.
- A file may be anything from a disk file to a terminal or a printer.
- A file pointer is a pointer to a structure, which contains information about the file, including its name, current position of the file, whether the file is being read or written, and whether errors or end of the file have occurred.
- The fopen() function opens a stream for use and links a file with that stream.
- The fclose() function closes a stream that was opened by a call to fopen().
- The function fcloseall() can be used when many open streams have to be closed at the same time.
- The function fputc() is used to write characters, and the function fgetc() is used to read characters from an open file.
- The functions fgets() and fputs() do the same operations as that of fputc() and fgetc(), except that they work with strings.
- The feof() function is used to indicate end-of-file when the file is opened for binary operations.
- The rewind() function resets the file position indicator to the beginning of the file.
- The function ferror() determines whether a file operation has produced an error.
- The remove() function erases the specified file.

Concepts

Summary

- The fflush() function flushes out the buffer. If a file is opened for read the input buffer will be cleared, while a file opened for write will have its output buffer written to the files.
- The fseek() function can be used to set the file pointer position.
- The library functions, fread() and fwrite() are used to read and write entire blocks of data onto the file.
- The buffered I/O system also includes two functions fprintf() and fscanf(), which are similar to the functions printf() and scanf(), except that they operate on files.

Check Your Progress

1. The two types of streams are the _____ and _____ streams.

2. Open files are closed when a program crashes. **(True/False)**

3. The _____ function opens a stream for use and links a file with that stream.

4. The function used for writing characters to a file is _____.

5. The `fgets()` function considers a new line character as a part of the string. **(True/False)**

6. The _____ function resets the file position indicator to the beginning of the file.

7. Whenever a character is read from or written to the stream, the _____ is incremented.

8. Files on which `fread()` and `fwrite()` operate must be opened in _____ mode.

9. The current location of the current active pointer can be found with the help of the _____ function.

Try It Yourself

1. Write a program that accepts data into a file and prints it in reverse order.

2. Write a program that transfers data from one file to another, excluding all the vowels (a, e, i, o, u). Exclude vowels in both upper and lower case. Display the contents of the new file.

Session

22 File Handling (Lab)

Objectives

At the end of this session, you will be able to:

- *Perform operations on text and binary files*
- *Open and close files*
- *Read from and write to files*
- *Use the file pointer*

The steps given in this session are detailed, comprehensive and carefully thought through. This has been done so that the learning objectives are met and the understanding of the tool is complete. Follow the steps carefully.

Part I – For the first 1 Hour and 30 Minutes:

22.1 File Handling in C

C provides a uniform interface for handling input and output (I/O). The access methods for files are same as those for handling devices. The key to this uniformity is that there are no file types in C. C treats all the files as streams.

22.1.1 Reading, Writing and Accessing data in files

There are several file handling functions in the stdio.h header file. Let us write a C program that makes use of these functions. The program creates a simple banking system. Customer details are accepted and stored in a file called customer. Details of transactions such as deposit and withdrawal are validated against the customer file. Valid transactions are recorded in the trans file. A report on low customer balances is also printed. The steps are listed below:

1. Define structures for customer and transaction data.

The code will be,

```
struct cust_st
{
    int acc_no;
    char cust_nm[30];
```

```
    float bal;
};
struct tran_st
{
    int acc_no;
    char trantype;
    float amt;
};
```

2. Display a menu to perform the various operations based on user input.

The code will be,

```
while(choice!=4)
{
    clrscr();
    printf("\nSelect choice from menu\n\n1. Accept customer details\
n 2. Record Withdrawal/Deposit transaction\n 3. Print Low Balance
Report\n 4. Exit\n\n Enter choice: ");
    scanf(" %d", &choice);
    .
    .
}
```

3. Invoke the appropriate function based on user choice.

The code will be,

```
if(choice==1)
    addcust();
else if(choice==2)
    rectran();
else if(choice==3)
    prnlowbal();
```

4. In the function to add customer details, define the file pointer to be associated with the customer file. Declare a structure variable for accepting the customer data.

The code for the same will be,

```
FILE *fp;
struct cust_st custdata;
```

5. Open the customer file in append mode so as to be able to add customer records. Confirm that the file open operation takes place.

The code for the same will be,

```
if((fp=fopen("customer", "a+"))==NULL)
{
    printf("\nERROR opening customer file");
    getch();
    return;
}
```

6. Accept the customer data into the structure variable and write the data to the customer file.

The code for the same will be,

```
fwrite(&custdata, sizeof(struct cust_st), 1, fp);
```

7. Close the customer file at the end of data entry.

The code for the same will be,

```
fclose(fp);
```

8. In the function to record transactions, define variables for the file pointers to the customer and trans files. Also, define structure variables to accept transaction data and read customer data.

The code for the same will be,

```
FILE *fp1, *fp2;
struct cust_st custdata;
struct tran_st trandata;
```

9. Open the two files in appropriate modes. The customer file should be opened for reading and updation, whereas the trans file should allow adding of new records.

The code for the same will be,

```
if((fp1=fopen("customer", "r+w"))==NULL)
{
    printf("\nERROR opening customer file");
    getch();
    return;
}
if((fp2=fopen("trans", "a+"))==NULL)
{
    printf("\nERROR opening transaction file");
    getch();
    return;
}
```

10. Accept the account number for the transaction and ensure that it exists in the customer file.

The code for the same will be,

```
while((fread(&custdata, size, 1, fp1))==1 && found=='n')
{
    if(custdata.acc_no==trandata.acc_no)
    {
        found='y';
        break;
    }
}
```

11. Ensure that a valid transaction type is entered.

The code for the same will be,

```
if(trandata.trantype!='D' && trandata.trantype!='d' && trandata.
trantype!='W' && trandata.trantype!='w')
    printf("\t\tInvalid transaction type, please reenter");
```

12. For withdrawal transactions, ensure that the withdrawal amount is available in the customer account. If available, update the account balance. Update the account balance for deposit transactions too.

The code for the same will be,

```
if(trandata.trantype=='W' || trandata.trantype=='w')
{
    if(trandata.amt>custdata.bal)
        printf("\nAccount balance is %.2f. Please reenter withdrawal
amount.", custdata.bal);
    else
    {
        custdata.bal-=trandata.amt;
        .
        .
    }
    else
    {
        custdata.bal+=trandata.amt;
        .
        .
    }
}
```

13. Write the new transaction record to the trans file and the updated customer record to the customer file.

The code for the same will be,

```
fwrite(&trandata, sizeof(struct tran_st), 1, fp2);
fseek(fp1, (long)(-size), 1);
fwrite(&custdata, size, 1, fp1);
```

Note that during the check for the customer account number, the last record read was for the customer whose transaction is being processed. So, the file pointer to the **`customer`** file would be at the end of the record that needs to be updated. The file pointer is repositioned to the beginning of the record using the `fseek()` function. Here **`size`** is the integer variable that stores the size of the structure for customer data.

14. Close the two files after the transactions data entry.

The code for the same will be,

```
fclose(fp1);
fclose(fp2);
```

15. In the function to display low balances, define the file pointer to be associated with the customer file. Declare a structure variable for reading the customer data.

The code for the same will be,

```
FILE *fp;
struct cust_st custdata;
```

16. After opening the file in the read mode, read each customer record and check the balance. If it is less than 250, print the record.

The code for the same will be,

```
while((fread(&custdata, sizeof(struct cust_st), 1, fp))==1)
{
    if(custdata.bal<250)
    {
        .
        .
        printf("\n%d\t%s\t%.2f", custdata.acc_no, custdata.cust_nm,
custdata.bal);
    }
}
```

17. Close the customer file.

The code for the same will be,

```
fclose(fp);
```

Let us look at the complete program.

1. Invoke the editor in which you can type the C program.

2. Create a new file.

3. Type the following code:

```
#include<stdio.h>
struct cust_st
{
    int acc_no;
    char cust_nm[30];
    float bal;
};
struct tran_st
{
    int acc_no;
    char trantype;
    float amt;
};

void main()
{
    int choice=1;
    while(choice!=4)
    {
        clrscr();
        printf("\nSelect choice from menu\n\n 1. Accept customer
details\n 2.Record Withdrawal/Deposit transaction\n 3. Print Low
Balance Report\n 4. Exit\n\n Enter choice: ");
        scanf(" %d", &choice);
        if(choice==1)
            addcust();
        else if(choice==2)
            rectran();
        else if(choice==3)
            prnlowbal();
    }
}
```

```
addcust()
{
    FILE *fp;
    char flag='y';
    struct cust_st custdata;
    clrscr();
    if((fp=fopen("customer", "a+"))==NULL)
    {
        printf("\nERROR opening customer file");
        getch();
        return;
    }
    while(flag=='y')
    {
        printf("\n\nEnter Account number: ");
        scanf(" %d", &custdata.acc_no);
        printf("\nEnter Customer Name: ");
        scanf("%s", custdata.cust_nm);
        printf("\nEnter Account Balance: ");
        scanf(" %f", &custdata.bal);
        fwrite(&custdata, sizeof(struct cust_st), 1, fp);
        printf("\n\nAdd another? (y/n): ");
        scanf(" %c", &flag);
    }
    fclose(fp);
}

rectran()
{
    FILE *fp1, *fp2;
    char flag='y', found, val_flag;
    struct cust_st custdata;
    struct tran_st trandata;
    int size=sizeof(struct cust_st);
    clrscr();
    if((fp1=fopen("customer", "r+w"))==NULL)
    {
        printf("\nERROR opening customer file");
```

```
        getch();
        return;
    }
    if((fp2=fopen("trans", "a+"))==NULL)
    {
        printf("\nERROR opening transaction file");
        getch();
        return;
    }
    while(flag=='y')
    {
        printf("\n\nEnter Account number: ");
        scanf(" %d", &trandata.acc_no);
        found='n';
        val_flag='n';
        rewind(fp1);
        if(found=='y')
        {
            while(val_flag=='n')
            {
                printf("\nEnter Transaction type (D/W): ");
                scanf(" %c", &trandata.trantype);
                if(trandata.trantype!='D' && trandata.trantype!='d' &&
trandata.trantype!='W' && trandata.trantype!='w')
                    printf("\t\tInvalid transaction type, please
reenter");
                else
                    val_flag='y';
                }
            val_flag='n';
            while(val_flag=='n')
            {
               printf("\nEnter amount: ");
               scanf(" %f", &trandata.amt);
               if(trandata.trantype=='W' ||trandata.trantype=='w')
                {
                    if(trandata.amt>custdata.bal)
                        printf("\nAccount balance is %.2f. Please
reenter withdrawal amount.",custdata.bal);
```

```
                    else
                    {
                         custdata.bal-=trandata.amt;
                         val_flag='y';
                    }
               }
               else
               {
                  custdata.bal+=trandata.amt;
                  val_flag='y';
               }
            }
            fwrite(&trandata, sizeof(struct tran_st), 1, fp2);
            fseek(fp1, (long)(-size), 1);
            fwrite(&custdata, size, 1, fp1);
        }
        else
            printf("\nThis account number does not exist");
        printf("\nRecord another transaction? (y/n): ");
        scanf(" %c", &flag);
    }
    fclose(fp1);
    fclose(fp2);
}

prnlowbal()
{
    FILE *fp;
    struct cust_st custdata;
    char flag='n';
    clrscr();
    if((fp=fopen("customer", "r"))==NULL)
    {
        printf("\nERROR opening customer file");
        getch();
        return;
    }
    printf("\nReport on account balances below 250\n\n");
```

```
    while((fread(&custdata, sizeof(struct cust_st), 1, fp))==1)
    {
        if(custdata.bal<250)
        {
            flag='y';
            printf("\n%d\t%s\t%.2f", custdata.acc_no, custdata.cust_nm,
custdata.bal);
        }
    }
    if(flag=='n')
        printf("\nNo account balances found below 250");
    getch();
    fclose(fp);
}
```

To see the output, follow these steps:

4. **Save the file with the name `filesI.C`.**

5. **Compile the file, `filesI.C`.**

6. **Execute the program, `filesI.C`.**

7. **Return to the editor.**

The output of the program is shown below:

```
Select choice from menu

1. Accept customer details

2. Record Withdrawal/Deposit transaction

3. Print Low Balance Report

4. Exit

Enter choice:
```

A sample output of the function to add customer details is shown below:

```
Enter Account number: 123

Enter Customer Name: E.Wilson

Enter Account Balance: 2000

Add another? (y/n):
```

A sample output of the function to add transaction details is shown below:

```
Enter Account number: 123

Enter Transaction type (D/W): W

Enter amount: 1000

Record another transaction? (y/n):
```

A sample output of the function to display the low balance report is shown below:

```
Report on account balances below 250

104    Jones    200
113    Sharon   150
120    Paula    200
```

Part II – For the next 30 Minutes:

1. Write a C Program to display the differences between two files accepted as command line arguments. For each difference, display the position at which the difference is located and the characters in the two files at that position. Also, ensure that the user enters a valid number of command-line arguments. Lastly, display the total number of differences found.

   To do this,

   a. Declare the variables `argv` and `argc` to receive the command line arguments.

   b. Declare file pointers for the two files.

   c. Validate `argc` to ensure the correct number of command-line arguments is entered.

   d. Open the two files in the read mode.

   e. Set a loop to read a character from both the files till the end of file is encountered for either.

   f. If the characters are different, display them along with their position. Increment the counter for differences.

   g. If the end of file is encountered for a file, print the remaining characters of the other file as differences.

   h. Check the differences counter to display appropriate messages.

   i. Close the two files.

Try It Yourself

1. Write a C program to copy the contents of one file onto another excluding the words **`a`**, **`an`** and **`the`**.

2. Write a C program to accept two series of numbers. Store each series in a separate file. Sort the series in each file. Merge the two series into one, sort, and store the resultant series into a new file. Display the contents of the new file.

APPENDIX

“The real voyage of discovery consists not in seeking new lands, but in seeing with new eyes”

Appendix

A Standard Library Functions

Input and Output: <stdio.h>

FILE *fopen(const char *filename, const char *mode)

FILE *freopen(const char *filename, const char *mode, FILE *stream)

int fflush(FILE *stream)

int fclose(FILE *stream)

int remove(const char *filename)

int rename(const char *oldname, const char *newname)

FILE *tmpfile(void)

char *tmpnam(char s[L_tmpnam])

int setvbuf(FILE *stream, char *buf, int mode, size_t size)

void setbuf(FILE *stream, char *buf)

int fprint(FILE *stream, const char *format, ...)

int sprintf(char *s, const char *format, ...)

vprintf(const char *format, va_list arg)

vfprintf(FILE *stream, const char *format, va_list arg)

vsprintf(char *s, const char *format, va_list arg)

int fscanf(FILE *stream, const char *format, ...)

int scanf(const char *format, ...)

int sscanf(char *s, const char *format, ...)

int fgetc(FILE *stream)

char *fgets(char *s, int n, FILE *stream)

int fputc(int c, FILE *stream)

int fputs(const char *s, FILE *stream)

int getc(FILE *stream)

int getchar(void)

char *gets(char *s)

int putc(int c, FILE *stream)

int putchar(int c)

int ungetc(int c, FILE *stream)

size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream)

size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *stream)

int fseek(FILE *stream, long offset, int orogin)

long ftell(FILE *stream)

void rewind(FILE *stream)

int fgetpos(FILE *stream, fpos_t *ptr)

int fsetpos(FILE *stream, const fpos_t *ptr)

void clearerr(FILE *stream)

int feof(FILE *stream)

int ferror(FILE *stream)

void perror(const char *s)

Character Class Tests: <ctype.h>

isalnum(c)

isalpha(c)

iscntrl(c)

isdigit(c)

isgraph(c)

islower(c)

isprint(c)

ispunct(c)

isspace(c)

isupper(c)

isxdigit(c)

String Functions: <string.h>

char *strcpy(s , ct)

char *strncpy(s , ct , n)

char *strcat(s , ct)

char *strncat(s , ct , n)

int strcmp(cs , ct)

int strncmp(cs , ct ,n)

char *strchr(cs , c)

char *strrchr(cs , c)

size_t strspn(cs , ct)

size_t strcspn(cs , ct)

char *strstr(cs , ct)

size_t strlen(cs)

char *strerror(n)

char *strtok(s , ct)

Mathematical Functions: <math.h>

sin(x)

cos(x)

tan(x)

asin(x)

acos(x)

atan(x)

atan2(x)

sinh(x)

cosh(x)

tanh(x)

exp(x)

log(x)

log10(x)

pow(x,y)

sqrt(x)

ceil(x)

floor(x)

fabs(x)

ldexp(x)

frexp(x,double *ip)

modf(x,double *ip)

fmod(x,y)

Utility Functions: <stdlib.h>

double atof(const char *s)

int atoi(const char *s

long atol(const char *s)

double strrod(const char *s, char **endp)

long strtol(const char *s, char **endp, int base)

unsigned long strtoul(const char *s, char **endp, int base)

int rand(void)

void srand(unsigned int seed)

void *calloc(size_t nobj, size_t size)

void *malloc(size_t size)

void *realloc(void *p, size_t size)

void free(void *p)

void abort(void)

void exit(int status)

int atexit(void (*fcn)(void))

int system(const char *s)

char *getenv(const char *name)

void *bsearch(const void *key, const void *base, size_t n, size_t size, int (*cmp)(const void

*keyval, const void *datum))

void qsort(void *base, size_t n, size_t size, int (*cmp)(const void *, const void *))

int abs(int n)

long labs(long n)

div_t div(int num, int denom)

ldiv_t ldiv(long num , long denom)

GLOSSARY

“It doesn’t matter how much you want. **What really matters is how much you want it.**”

Glossary

A

Algorithm

Logical and concise list of steps required in solving a problem.

Array

A group of variables, which are of the same data type, and can be accessed using a common name.

B

Binary operator

An operator which requires two operands.

Binary Stream

A sequence of bytes with a one-to-one correspondence to those in the external device, that is, there are no character translations.

Boolean data type

It consists of either of the two values, True or False. Some languages use 0 for False and some non-zero value for True.

Bubble sort

A type of sort algorithm. In this the values of the elements are compared with the value in the adjacent element. If it is smaller, swapping takes place. In this manner, the smaller elements bubble up, and at the end, the array is sorted.

Buffer

A buffer is a temporary storage area, either in the memory or on the controller card for the device.

C

Char

Char type occupies one byte long and is capable of holding one character.

Glossary

Code

Collection of program statements; A Program.

Code snippet

Few lines from a program. A part of a program, performing an individual function.

Constant

A constant is a value whose worth never changes.

Construct

A set of instructions or steps in a program/pseudocode.

Counter variable

A type of variable used to keep track of the number of times a particular operation has been performed in a loop.

D

Data type

It is used to specify the type of data to be stored in a variable. Indirectly, therefore, it decides the amount of memory to be allocated to a variable to store that particular type of data.

E

Expression

Combination of an operator and its operand.

F

Flowchart

A graphical representation of an algorithm. It charts the flow of instructions or activities in a process. Each such activity is depicted using symbols.

Function

A set of statements, which perform a specific task. Functions may or may not return a value.

Function library

A collection of functions. Generally, a function library contains functions that deal with a specific task.

Glossary

I

Index

It indicates the array element which is to be accessed. The array index generally starts at 0.

Initialization

It is the process of assigning some initial value to a variable.

Insertion Sort

In insertion sort, each element in the array is examined, and inserted into its proper place among the elements that have already been sorted.

Integer

A number without any decimal portion. Integers can be positive or negative.

Iteration

See looping construct

K

Keyword

All languages reserve certain words for their internal use. These words hold a special meaning within the context of the particular language, and are referred to as ‘keywords’. While naming variables we need to ensure that we do not use one of these keywords as a variable name. Refer to Appendix B for the list of keywords in C.

L

Looping Construct

Often it is necessary to repeat certain steps a specific number of times or till some specified condition is met. The constructs which achieve these are known as iterative or looping constructs.

O

Operand

The value upon which the operator acts.

Operator

Symbols that perform some sort of operation upon data.

P

Parameter

A value that is passed to a function.

Pass by value

A way of passing values to a function. The address of the original variables, and not the values, are passed in the pass-by-reference method to the called function. Therefore, modifications made to the contents of this variable affects the values of the original variables.

Pass by reference

A way of passing values to a function. When variables are passed by value, the values of the variables within the functions are not reflected back in the main program, since a copy of the original variables is made.

Pointer

A Pointer is a variable, which contains the address of a memory location of another variable.

Procedure

These are subprograms that essentially break up a program into modules. Procedures cannot return a value.

Program

We need to provide the computer with a set of instructions to solve any problem at hand. This set of instructions is called a program.

Pseudocode

Pseudocode is not actual code (pseudo=false), but a method of algorithm-writing which uses a certain standard set of words which makes it resemble code. However, unlike code, pseudocode cannot be compiled or run.

S

Standard function

These functions are generally built into a programming language, and are used for performing often-required tasks.

Statement

A single line of a pseudocode or a program.

Glossary

Structure

A collection of variables of different data types that can be accessed as one unit using a common name.

Sub-program

Most programming languages provide us a way of breaking a long, continuous program into a series of small-programs, each of which perform a specific task. These small-programs are known as Sub-programs.

Syntax

Refers to the grammar of a programming language.

U

Unary operator

An operator requiring a single operand.

User defined function

These are functions written by the programmers.

V

Variables

Named locations in memory. Programmers use variables to refer to the memory location where a particular value is to be stored.

“A teacher is one who makes himself progressively unnecessary”

Reader's Response

Name Of Book : _____

Batch : _____ **Date :** _____

The members of the design team at Aptech Worldwide are always striving to enhance the quality of the books produced by them. As a reader, your suggestions and feedback are very important to us. They are of tremendous help to us in continually improving the quality of this book. Please rate this book in terms of the following aspects.

Aspects	**Rating**			
	Excellent	**Very Good**	**Good**	**Poor**
Presentation style	☐	☐	☐	☐
Suggestion :				
Simplicity of language	☐	☐	☐	☐
Suggestion :				
Topics chosen	☐	☐	☐	☐
Suggestion :				
Topic coverage	☐	☐	☐	☐
Suggestion :				

Aspects	Rating			
	Excellent	Very Good	Good	Poor
Explanation provided	☐	☐	☐	☐

Suggestion :

_____

_____

Quality of pictures / diagrams	☐	☐	☐	☐

Suggestion :

_____

_____

Overall suggestions :

_____

_____

_____

_____

Please fill up this response card and send it to :

The Design Centre,
Aptech Limited.
Aptech House,
A-65, MIDC, Marol,
Andheri (East),
Mumbai - 400 093.
INDIA

Your efforts in this direction will be most appreciated