

# PROBLEM SOLVING AND PROGRAMMING IN



**COURSE CODE: B21CA02DC**  
**Bachelor of Computer Applications**  
**Discipline Core Course**

**SELF LEARNING MATERIAL**



SREENARAYANAGURU  
OPEN UNIVERSITY

**SREENARAYANAGURU OPEN UNIVERSITY**

The State University for Education, Training and Research in Blended Format, Kerala

# SREENARAYANAGURU OPEN UNIVERSITY

## Vision

*To increase access of potential learners of all categories to higher education, research and training, and ensure equity through delivery of high quality processes and outcomes fostering inclusive educational empowerment for social advancement.*

## Mission

To be benchmarked as a model for conservation and dissemination of knowledge and skill on blended and virtual mode in education, training and research for normal, continuing, and adult learners.

## Pathway

Access and Quality define Equity.

# **Problem Solving and Programming in C**

**Course Code: B21CA02DC**  
**Semester - I**

**Discipline Core Course**  
**Undergraduate Programme**  
**Bachelor of Computer Applications**  
**Self Learning Material**



**SREENARAYANAGURU**  
**OPEN UNIVERSITY**

**SREENARAYANAGURU OPEN UNIVERSITY**

The State University for Education, Training and Research in Blended Format, Kerala

# Problem Solving and Programming in C

## Semester - I



SREENARAYANAGURU  
OPEN UNIVERSITY

All rights reserved. No part of this work may be reproduced in any form, by mimeograph or any other means, without permission in writing from Sreenarayanaguru Open University. Printed and published on behalf of Sreenarayanaguru Open University by Registrar, SGOU, Kollam.

[www.sgu.ac.in](http://www.sgu.ac.in)

ISBN 978-81-970547-0-9



## DOCUMENTATION

### Academic Committee

Dr. Aji S.  
P. M. Ameera Mol  
Shamly K.  
Dr. Jeeva Jose  
Dr. Priya R.  
Dr. Anil Kumar

Sreekanth M. S.  
Dr. Vishnukumar S.  
Joseph Deril K. S.  
Dr. Bindu N.  
Dr. Ajitha R. S.  
N. Jayaraj

### Development of the Content

Dr. Jennath H. S., Rekha Raj C. T., Shamin S, Suramya Swamidas  
P.C., Prabha M.R., Divya Das

### Review

Content : Prof. Viji Balakrishnan  
Format : Dr. I. G. Shibi  
Linguistics : Dr. Subhash Chandran

### Edit

Prof. Viji Balakrishnan

### Scrutiny

Dr. Gopakumar C., Dr. Jennath H. S., Shamin S, Suramya  
Swamidas P. C.

### Co-ordination

Dr. I. G. Shibi and Team SLM

### Design Control

Azeem Babu T. A.

### Cover Design

Jobin J.

### Production

April 2024

### Copyright

© Sreenarayanaguru Open University 2024





Dear

With immense joy and excitement, I extend my heartfelt greetings to all of you and warmly welcome you to Sreenarayanaguru Open University.

Established in September 2020 as a state-driven initiative, Sreenarayana-guru Open University is dedicated to advancing higher education through open and distance learning. Our vision is guided by the principle of “access and quality define equity,” laying the foundation for a celebration of excellence in education. I am delighted to share that we are steadfast in our commitment to uphold the highest standards and refrain from compromising on the quality of education we offer. The university draws its inspiration from the legacy of Sreenarayana Guru, a revered figure in the Indian renaissance movement. His name serves as a constant reminder for us to prioritize quality in all our academic endeavors.

Sreenarayanaguru Open University operates within the practical framework of the widely recognized “blended format.” Acknowledging the constraints faced by distance learners in accessing traditional classroom settings, we have curated a pedagogical approach centered on three main components: Self Learning Material, Classroom Counselling, and Virtual Modes. This comprehensive blend is poised to deliver dynamic learning and teaching experiences, maximizing engagement and effectiveness. Our unwavering commitment to quality ensures excellence across all aspects of our educational initiatives.

The university aims to offer you an engaging and stimulating educational environment that fosters active learning. The SLM is designed to offer a comprehensive and cohesive learning experience, fostering a deep interest in the study of technological advancements in IT. Careful consideration has been given to ensure a logical progression of topics, facilitating a clear understanding of the discipline’s evolution. The curriculum is thoughtfully crafted to provide ample opportunities for students to navigate through the current trends in information technology. Furthermore, this course is designed to provide essential insights into computer hardware, software classification, and foundational HTML concepts crucial for web development.

We assure you that the university student support services will closely stay with you for the redressal of your grievances during your student-ship. Feel free to write to us about anything that seems relevant regarding the academic programme.

Wish you the best.



Regards,  
Dr. Jagathy Raj V. P.

24-04-2024

# BLOCK 01

# BLOCK 02

# BLOCK 03

# BLOCK 04

# BLOCK 05

# BLOCK 06

## CONTENTS

<b>BASIC PROGRAMMING CONCEPTS</b>	<b>1</b>
Unit 1 Problem Solving and Algorithms	2
Unit 2 Introduction to C Programming	17
Unit 3 Variables and Data Types	26
Unit 4 Operators and Expressions	38
<b>IO STATEMENTS, CONTROL STRUCTURES, ARRAYS, AND POINTERS</b>	<b>56</b>
Unit 1 Input Output Statements	57
Unit 2 Control Structures and Looping	67
Unit 3 Arrays and Strings	99
Unit 4 Pointers and Dynamic Memory Allocation	117
<b>FUNCTIONS, STRUCTURES AND UNION</b>	<b>137</b>
Unit 1 Functions	138
Unit 2 Recursion	163
Unit 3 Call by Value and Call by Reference	181
Unit 4 Structures and Union	196
<b>STORAGE CLASSES, FILES, AND PREPROCESSORS</b>	<b>234</b>
Unit 1 Storage Classes	235
Unit 2 Managing Files	248
Unit 3 Command-line Arguments	260
Unit 4 Macros and Preprocessor Directives	268
<b>LAB MANUAL PART A</b>	<b>285</b>
Experiment 5.1 Familiarization of Input and Output Statements	286
Experiment 5.2 Variables and Datatypes: Familiarizing basic conversions	290
Experiment 5.3 Operators and Expressions : Age Calculator	297
Experiment 5.4 Control Structures: Familiarization of various Looping Statements	300
Experiment 5.5 Arrays Pointers and Recursion	304
Experiment 5.6 Structures and Unions	307
<b>LAB MANUAL PART B</b>	<b>311</b>
Experiment 6.1 Functions : Recursion, Call by Value and Call by Reference, File Management	312
<b>MODEL QUESTION PAPER SETS</b>	<b>319</b>

```
#include "KMotionDef.h"
```

```
int main()
```

```
{
```

```
    ch0->Amp = 250;
```

```
    ch0->output_mode=MICROSTEP_MODE;
```

```
    ch0->Vel=70.0f;
```

```
    ch0->Accel=500.0f;
```

```
    ch0->Jerk=2000.0f;
```

```
    ch0->Lead=0.0f;
```

```
    EnableAxisDest(0,0);
```

```
    ch1->Amp = 250;
```

```
    ch1->output_mode=MICROSTEP_MODE;
```

```
    ch1->Vel=70.0f;
```

```
    ch1->Accel=500.0f;
```

```
    ch1->Jerk=2000.0f;
```

```
    ch1->Lead=0.0f;
```

```
    EnableAxisDest(1,0);
```

```
    DefineBoardSystem(1,0,0);
```

```
    return 0;
```

```
}
```

# BLOCK 1

# Basic Programming Concepts





# Problem Solving and Algorithms

## Learning Outcomes

After the successful completion of the unit, the learner will be able to:

- ◆ understand the concept of problem-solving with computers
- ◆ familiarize algorithms and flowcharts
- ◆ develop skills in systematic approaches to solve a problem
- ◆ understand different types of computer languages

## Prerequisites

Intelligence is one of the main characteristics that distinguishes humans from all living beings on the planet. Essential intelligence includes problem-solving and developing methods to deal with the many problems that arise in daily life.

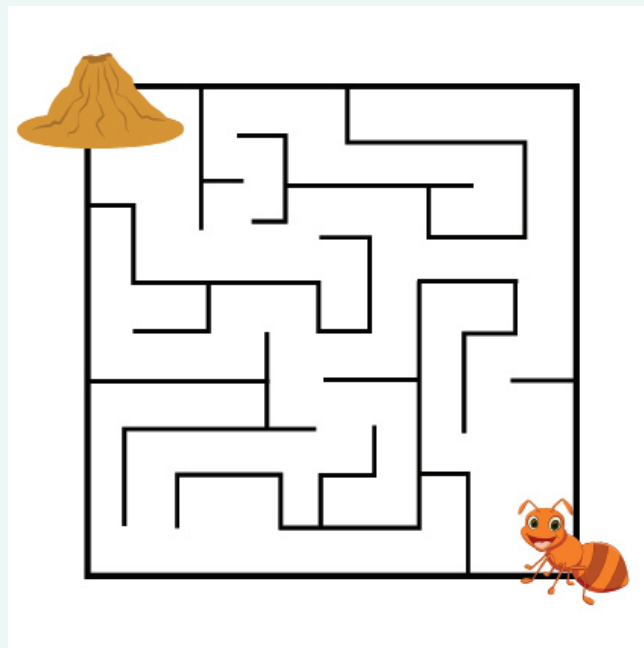


Fig 1.1.1 Finding path problem



Fig 1.1.1 must be familiar to all of us. Perhaps this was one of the first problems we encountered in our childhood. Finding the path for the ant was difficult, but it was fun to do it. In the same way, we come across a lot of situations that require problem-solving skills.

Problem-solving is an essential skill that is necessary for a human being. In our daily life, no day goes without involving problem-solving. Purchasing, solving puzzles, or solving a mathematical problem involves problem-solving skills.

How will you solve a problem? The solution to a problem depends on the data available to you. For example, how will you decide how much money you want to withdraw from an ATM?

The decision is made based mainly on three factors:

1. The amount you want
2. Bank balance and
3. The minimum balance that you want to maintain on your account

Computers are designed with an aim to perform what human beings do. Programs are the instructions given to computers for solving problems. So, what is programming..?

Programming a computer means identifying and enlisting the steps to solve a problem. One who does this process is called a programmer.

In this unit, let us understand what problem-solving is and its importance in computer science.

## Key Concepts

Problem-solving, algorithm, flowchart, programming language

## Discussion

### 1.1.1 Problem-solving

In this section, let us discuss the problem-solving techniques involved in a computer program. As we know, a computer is a machine that blindly follows the instructions given by the user. So it is necessary to give a sequence of instructions to the computer to solve a given problem. This sequence of instructions is known as a computer program.

The most challenging aspect of com-

puter programming is problem-solving or breaking down the problem into solutions involving sequential stages. Let us start with a mathematical problem to understand the problem-solving methodology:

**Example 1.1.1:** *You are asked to hire a Gardner for seven days on daily wages. You have a gold bar in your hand, which is equally marked at seven points. You need to give  $1/7$ th of the golden bar each day as wages. But you are not allowed to cut the golden bar more than two.*



How will you cut the bar to pay the gardener for seven days? Consider the image below as the gold bar (ref. Fig 1.1.1a), which is marked at seven equal points. Solution is explained in Table 1.1.1



Fig 1.1.1a: Gold Bar

Table 1.1.1 Steps involved in problem-solving

Steps involved	Solution
Analysing the problem	<p>Let's understand the question:</p> <ul style="list-style-type: none"> <li>◆ We need to pay the gardener for seven days</li> <li>◆ Each day he is eligible to get <math>1/7^{\text{th}}</math> of the golden bar</li> </ul> <p>The number of times that we are allowed to cut the bar is 2. This means we will have 3 bars of gold at the end</p>
Develop the step by step solution to the problem(Algorithm)	<ul style="list-style-type: none"> <li>◆ First, we will cut <math>1/7^{\text{th}}</math> of the bar (let's say b1) to give it on the first day.</li> <li>◆ Then cut <math>2/7^{\text{th}}</math> (b2) of the bar; the remaining (b3) contains <math>4/7^{\text{th}}</math> of the golden bar.</li> <li>◆ Give b2 on the second day and get back b1</li> <li>◆ Give b1 on day 3</li> <li>◆ On day 4 give b3 and get b2 and b1 back.</li> <li>◆ On day 5 give b1</li> <li>◆ On day 6 give b2 and get b1 back.</li> <li>◆ On day 7 give b1</li> </ul>
Convert the solution to any computer language (coding)	Implementing the solution
Verify the correctness of the code (testing and debugging)	Evaluating the solution

In Table 1.1.1, you have seen how a mathematical problem is solved and how the steps are interrelated with computers.

### Problem-solving using a computer

In the previous example, we showed how to solve a mathematical problem. Now let's understand how to resolve computer problems. A computer needs our instruction to solve any problem.

An **instruction** is a statement telling the computer which action should be performed. There are multiple ways to solve a problem.

### 1.1.2 Approaches in problem-solving

There may be multiple solutions and techniques available to solve a problem using computer. Mainly there are two popular approaches for problem-solving:

1. Top-down approach
2. Bottom-up approach

### 1.1.3 Top-Down Approach

The name itself explains the concept. In the top down approach, we first understand the whole problem, without going for the details. Formulate an overall design for the solution, and then move on to the details as required.

Let us consider example 1 again. What did we do to solve the problem? First, we analyzed the question to understand it properly. Then we broke down the question to arrive at the step by step solution. Once we were clear with the logic, we implemented the solution. It is a perfect example of a top-down approach.

### 1.1.4 Bottom-Up Approach

This approach will find the smallest module, solve it first, and then integrate all such modules to get the whole solution.

Manufacturing a car or vehicle can be considered as a bottom-up approach where the individual parts of the vehicle are manufactured individually and then integrated to make the car. You can take real-life examples like constructing a house.

So far, we have seen a problem, how it is decomposed, and the different approaches for solving the problem. Now let's see the steps involved in problem-solving.

### 1.1.5 Steps involved in problem-solving

What did we do first to solve the problem? We **analyse the problem**. Why do you think it is so important to analyze the problem first?

Let us consider a real-life scenario: suppose you went to a restaurant and ordered *dosa* for your breakfast. The waiter will bring the *dosa* on a plate along with the side dishes, and it is served in a standard servicing environment.

What was your order? Did you ask for the side dishes? Did you specify that *dosa* was to be served on a plate? Did you ask for a table and chair?

Not exactly, You got all these facilities because you were communicating with humans. When you instruct a human, he/she assumes other requirements that are needed to complete the instruction. But when you instruct a computer, it assumes nothing. So it is crucial to understand and define the problem.

The second step of problem-solving is identifying the step by step solutions to solve it. There might be more than one solution available for a problem, but we need to choose the best solution from them.

Suppose we have identified the problem and the solution to it. How will we solve it using a computer? So, the third step of problem-solving involves the precise communication between the human and the computer. Here we have a challenge in that, the computer does not understand human languages. So we need to convert the instructions into the computer's language, this step is known as *coding*.

The final step of problem-solving is the *Evaluation of the solution*. What is involved in Evaluation? We might make mistakes while we give instructions to a computer. Evaluation is the process by which such mistakes are weeded out; because we want the results to be correct. So we need to walk through the solution and check if all possibilities of the problem statement are met. This process is known as *testing*. The analysis of the step by step solution to detect flaws, if any, is called *debugging*.

- ◆ Solve the problem effectively and error-free
- ◆ To obtain a sequential step to solve the problem

It is important to note here that all problems of the world cannot be solved. Several problems have no solution, and they are referred to as **Open Problems**. You can write an algorithm for a problem if you can solve it. In the next section, we will learn what an algorithm is.

### 1.1.6 Algorithm

In the last section, we discussed the problem-solving technique and here, let's discuss algorithms. An algorithm can be defined as a set of instructions or a step-by-step process to complete a task or a problem. Let's understand the algorithmic concept through an example.

Let's say you want to divide 16 by 3. What are the steps to do it?

1. Write down 16 by 3
2. Check if 16 comes in the table of 3
3. If not, find the closest number less than 16 that comes in the

Algorithm is a sequence of activities to be processed for getting the desired output from a given input, which comprises of a set of unambiguous rules and have a clear stopping point.

Why is problem-solving critical in computer science?

- ◆ To breakdown and understand the complex logic of the problem

table of 3

4. We will get 15 ( $5 \times 3 = 15$ )
5. So the quotient is five, and the remainder is 1 ( $16-15$ )



The example above is a simple one. There may be multiple ways to solve a problem.

*to solve the problem)*

♦ *and output.*

An algorithm has three parts:

While writing the algorithm, we have to use the following symbols:

- ♦ **the input,**
- ♦ **process**(*step by step approach*

Table 1.1.2 Arithmetic operators and description

Symbols	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
←	Assignment (example: $A \leftarrow B+3$ means $A = B+3$ )

Let us understand it better through examples:

**Example 1.1.2:** Find the perimeter of a square

**Algorithm:**

Input: Let 'a' be the length of a side of the square

Output: *the* calculated perimeter of the square

Process :

**Step 1:** input the side length 'a'

**Step 2:**  $\text{perimeter} \leftarrow 4 * a$

**Step 3:** print perimeter

**Example 1.1.3:** Write an algorithm to find the largest of two numbers

**Algorithm:**

Input: Let 'a' be the first number and 'b' be the second number. C will be the output.

Output: The largest of 'a' & 'b'.

Process :

**Step 1:** input the numbers 'a' and 'b'.

**Step 2:** check whether 'a' is greater than 'b'.

    If yes:  $C \leftarrow a$

    If not:  $C \leftarrow b$

**Step 3:** print C

**Example 1.1.4:** Write an algorithm to print all odd numbers between 1 to 100

**Algorithm:**

Input: Let 'I' be a number and  $I \leftarrow 0$ .

Output: Odd numbers between 1 and 100

Process :

**Step 1:** start

**Step 2:**  $I \leftarrow 1$

**Step 3:** print I

**Step 4:**  $I \leftarrow I + 2$

**Step 5:** if  $I < 100$ , go to step 3

**Step 6:** End

Did you analyse three examples? Are they the same? Analyse and find the different type of algorithms.

### 1.1.7 Flowchart

A flowchart is a pictorial/graphical representation of an algorithm. A flowchart uses different symbols, shapes and arrows to represent the process flow.



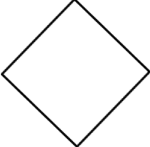
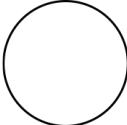

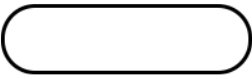

The table shown below depicts the algorithm and flowchart of the process for unlocking a phone. From this illustration, we understand that once the flowchart is made, it is easy to convert it into any programming language.

Table 1.1.3 An illustrative example of the difference between algorithm and flowchart

Algorithm	Flowchart
<p>Step 1: Get the password</p> <p>Step 2: Check the password</p> <ul style="list-style-type: none"><li>a) If matches: unlock the phone</li><li>b) If not: print incorrect password message and repeat step 1</li></ul>	<pre>graph TD; Start([Start]) --&gt; GetPass[/Get the password/]; GetPass --&gt; IsCorrect{Is the password correct?}; IsCorrect -- Yes --&gt; PhoneUnlock[Phone unlock]; PhoneUnlock --&gt; End([End]); IsCorrect -- No --&gt; WrongPassword[Wrong password]; WrongPassword --&gt; GetPass;</pre>

### 1.1.7.1 Symbols used in a flowchart

Table 1.1.4 Symbols used in flowchart

Symbol	Name	Function
	Process	Indicates any type of internal operation inside the processor or memory.
	Input/ Output	Used for any input/output operation. Used to obtain value from the user or to represent the output data.
	Decision	Used to ask a question that can be answered with yes/no or true/false.
	Connector	Allows the flowchart to be drawn without intersecting lines or without a reverse flow
	Predefined process	Used to invoke a subroutine or an interrupt program
	Terminator	Indicates the starting or end of the program, process or interrupt program
	Flow lines	Show direction of flow

### 1.1.7.2 Rules for flowcharting

1. All the boxes in the flowchart should be connected with arrows.
2. All flowchart should start and end with a terminal box.
3. All flowchart symbols have only one entry point on the top. No other entry points are allowed.
4. Exit points of all symbols are at the bottom except for the decision symbol.
5. The decision symbol has two exit points: these can be on the sides or at the bottom.
6. The general flow of a flowchart is from top to bottom. Upward flow can be shown as long as it

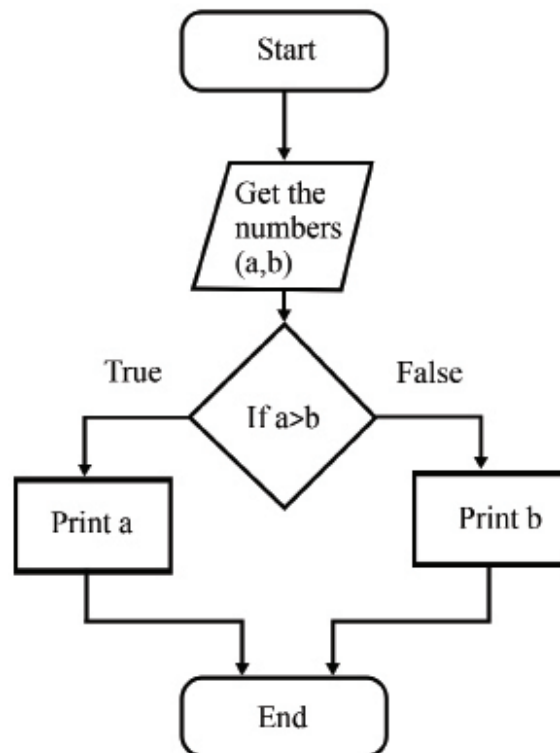


doesn't exceed more than three symbols.

In the previous section, we discussed algorithms and flowchart, which is a way

to express the steps we need to follow. It is the task of a programmer to generate a program from an algorithm or a flowchart. A programmer is a person who is an expert in any programming language.

**Example 1.1.5 :** Draw a flowchart to find the largest of two numbers



### 1.1.8 Evolution of Programming languages

A programming language is just like any other language. It has a character set, vocabulary and grammar. Do you know which languages are understood by the computer?

To understand that, first, we need to understand how a computer works.

A computer is an electronic device that can only understand the presence or absence of an electronic signal. The presence of

the signal is denoted as '1', and lack of signal is indicated as '0'. All information (alphabets, numbers etc.) are represented in a pattern of 1s and 0s inside a computer as shown in Fig 1.1.2.

For example, the smallest unit of representing a signal is referred to as a bit. Anything inside a computer is represented as a sequence of bits. This representation is known as a binary representation.

#### 1.1.8.1 Machine languages

The language that a computer understands can be called a machine language.

Digit	Representation of digits in 1's and 0's
10	1010
14	1110

Fig 1.1.2 Data representation in binary

The computer understands only binary language. Binary has only two characters (0 and 1). The binary code is very easy to understand by the computer and less computation is required. The major disadvantage of a binary language is the complexity in representation, which is challenging to a human. So it is difficult to make, and hard to find out mistakes while writing a program in binary. How can we solve it? We need a system that can convert natural languages into binary language. But it is not an easy job at all. The biggest challenge is the ambiguity in natural languages. The instructions may not be obvious or may have multiple meanings. For example, if someone asks you about the bank, what would you say? Here the word “bank” is a little confusing. It can be a financial institution or a riverside.

So we need an intermediate language that should have the following properties:

- ◆ Close to English
- ◆ The grammar should be stricter
- ◆ Reduce ambiguity

The intermediate languages are of two types

1. Assembly language
2. High-level language

#### 1.1.8.2 Assembly language

Assembly language uses mnemonics, words or symbolic instructions (ex: ADD for addition and MUL for multiplication). It allows complex jobs to run in a straightforward way. Assembly language requires less memory. It works fast. But the language is hardware-oriented and difficult to interpret. The effort of the programmer is high, and they need to code differently for different systems.

#### 1.1.8.3 High-level language

All those we refer to as programming languages in our general literature are high level languages. This category of languages is more like English. It is programmer-friendly. The high-level languages are easy to write, maintain and find the errors(debugging). The high-level language requires a translator (compiler/interpreter) to convert the program into machine level language (Refer fig 1.1.3). The high-level languages are machine-independent.

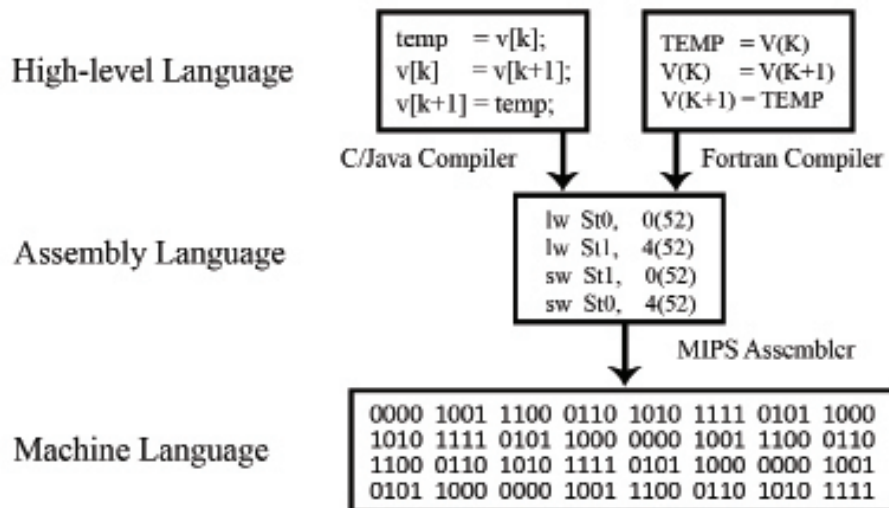


Fig 1.1.3 High-level language, Assembly Language, Machine Language

### 1.1.9 Translators

The high-level language needs to be translated into machine language. There are two approaches in doing this translation – one is a compilation and the other is interpretation.

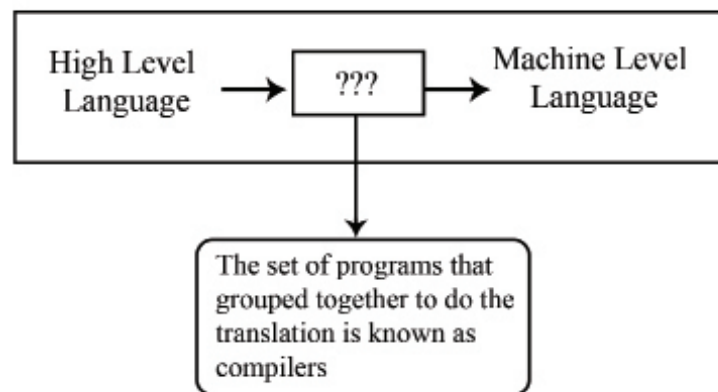


Fig 1.1.4 Language Translator

#### Compilers

A compiler is a software that translates the program you write in high level language to its corresponding machine language. The compiler analyzes the entire program as a unit, and translates it as whole and

shows syntax errors with line numbers. If the program is syntactically correct, it can be translated to a machine language program which is ready for execution.

#### Interpreters

Interpreters also do the same job, translating

high level language programs to machine language programs. Then what makes it different? Interpreters read each line in your program and translate them one by one. If there is any error in one line, it will translate the next line only after correcting that line.

The program given for compilation is termed source program (source code) and the translated program is called object program (object code).

## Recap

- ◆ Problem solving helps to simplify the complex logic of a problem
- ◆ Top-down and bottom-up are two approaches used to solve problems
- ◆ Four steps involved in problem-solving are:
  - a. Analysing the problem
  - b. Identifying the step by step solution
  - c. Coding
  - d. Evaluation of the solution
- ◆ An algorithm is a set of instructions. It can be a procedure or a formula to solve a problem
- ◆ We use some notions to write down the steps in the algorithm.
- ◆ A good algorithm must be readable and finite.
- ◆ A flowchart is a graphical representation to depict the steps to solve a problem
- ◆ A flowchart uses a set of standard symbols to represent various steps
- ◆ A flowchart helps to easily understand the logic of complicated and lengthy problems.
- ◆ Computers understand only binary language, the language of 1s and 0s. We need to express everything in terms of 1s and 0s. This language is termed machine language.
- ◆ Assembly language uses mnemonics for machine operations. The assembler converts the assembly language program to machine language.
- ◆ We write programs in high level language. A translator software converts it to machine language.
- ◆ Compilers read the entire program and make corrections and optimizations before translation.
- ◆ Interpreters perform a line by line translation of the source program.



## Objective Type Questions

1. What are the approaches to get a problem solved?
2. What is the main purpose of program planning (list maximum two)?
3. What is the first step involved in problem-solving?
4. What is the technique to find mistakes in a computer program?
5. How will you communicate with the computer?
6. What is an algorithm?
7. A problem can have multiple algorithms. Write whether true or false.
8. An algorithm is a special method that a computer can use to solve a problem. Write whether true or false.
9. Which language is used to represent algorithms?
10. Flowchart depicts the \_\_\_\_ flow in an algorithm
11. \_\_\_\_ symbol used for the decision statements in a flowchart.
12. Rectangle and parallelogram can be used interchangeably in a flowchart. Is this true?
13. Which symbol is used as a connector in a flowchart?
14. What are the symbols used in binary language?
15. MOV, ADD are examples of \_\_\_\_ in assembly language.
16. Compiler is a software for converting machine programs to high level programs. State whether true or false.
17. We write programs in English-like languages and a translator converts it for execution. State whether true or false.

## Answers to Objective Type Questions

1. Top-down and bottom-up
2. Analysing the problem
3. i) breakdown and understand the complex logic of the problem  
ii) solve the problem effectively
4. Testing and Debugging
5. Through coded program
6. An algorithm is a step by step instruction to solve a problem
7. true



8. true
9. Any language
10. Sequential
11. Diamond symbol
12. No. Rectangle for operational statements and parallelogram for I/O statements only
13. Circle
14. 0 and 1
15. Mnemonics/op-code
16. False
17. True

## Assignments

1. Write an algorithm to print the numbers 1 to 100.
2. Write an algorithm to read two numbers and find their sum.
3. Write an algorithm to find the most significant value of any three numbers.
4. Write an algorithm to find the sum of the first 100 natural numbers.
5. Draw the flowchart of the above problems.
6. What is an algorithm and flowchart ? Explain different tools used in algorithm design.
7. Explain different steps to solve a problem.
8. Explain different types of computer languages.
9. Explain language processors in a computer.

## Reference

1. Balagurusamy, E. Programming in C. Tata McGraw-Hill Education, 2008.
2. Forouzan, Behrouz A. "Computer Science, A Structured Programming Approach Using C, Brooks." Cole. California, 2001.
3. Kernighan, Brian W., and Dennis M. Ritchie. The C programming language. Prentice Hall, Englewood Cliffs, N.J., 1988.
4. Gottfried, Byron S. Schaum's Outline of Programming in Structured Basic. McGraw-Hill Professional, 1992.



# Introduction to C Programming

## Learning Outcomes

After the successful completion of the unit, the learner will be able to:

- ◆ make aware of the structure of C programming
- ◆ know the execution of C programming
- ◆ understand the steps of compiling and running

## Prerequisites

In Unit 1, we learned how to create algorithms for a variety of problems. Let us take a step further and write programs in the C programming language. But before we get into the C programming language, let us first define what a programming language is and how it functions.

A program is an ordered set of instructions to be executed by a computer to carry out a particular task. A programming language is a language used to specify this set of instructions to the computer.

As we all know, computers understand only 0s and 1s, known as machine language or low-level language. But humans find it challenging to write or understand instructions based on 0s and 1s. This resulted in developing high-level programming languages such as Python, C++, Visual Basic, PHP, and Java, which are easier to understand for humans but not for computers.

Source code refers to a program written in a high-level language. As you might recall from Unit 1, language translators such as compilers and interpreters must convert source code into machine language.

## Key Concepts

C programming, Execution of C program



## Discussion

### 1.2.1 What is C programming?

Let us start the discussion on a specific programming language, the C language. Just like any other language C has its vocabulary and strict grammar rules. The compiler will not accept any statement written in C that does not follow the syntax rules. The compiler will reject it, saying that it's a syntax error.

C is a widely-used general-purpose programming language that is easy to learn and use. It is a machine-independent structured programming language widely used to create various applications, operating systems such as Windows, and other applications such as the Oracle database, Git, Python interpreter, etc.

Before going further into a discussion of C, let us look briefly at the history of C. C is a general-purpose, high-level language written by Dennis Ritchie in 1972. Initially, C was developed to document the early versions of the Unix operating system. C is machine-independent, so that

the developer need not worry about the targeted machine.

### 1.2.2 Popularity of C

#### Why has C become so popular?

The C language is robust; it has rich built-in functions, and operators. The programs written in C are efficient and fast. C is highly portable. This means you can run your C program on any machine with little or zero changes. The C is a structured language that helps the user think of a problem in small independent blocks (the top-down approach discussed in unit 1).

Before going into specific C features, let us look at some sample C programs and see how they work.

#### Let us Code

We can write a C program in an editor and save it with an extension of “.c”. Consider the sample program given below.

#### Example 1.2.1 :

```
/*  
program to print a message, and the program is saved with the name "goodday.c"  
You can save the program with any name with an extension ".c".  
*/  
  
#include <stdio.h >  
  
void main()  
{  
    printf("Have a Good Day..!");  
}
```

When “goodday.c” executed, it will produce the following output

*Have a Good Day*

Let us have a close look at the program shown in Example 1.2.1.

1. The first section of the program, that is, line number 1 is "comment" lines. It usually includes information like the name of the program, date, author etc. The comments can appear anywhere in a program. Remember, we can not have comments inside comments. But C supports multi-line comments.
2. Line 2 “#include <stdio.h>” is a preprocessor directive. This statement has to be included in all C programs. It informs the compiler to include the files present in the standard input-output library. “stdio.h” is a header file that consists of the “printf()” library function, which we are going to use in the above program to print “Hello world.”
3. Line 3 is the main function. Every C program should have the main function. It denotes the start of a program.
4. Line 4 - Indicates the beginning of the code block.
5. Line 5 - Print everything is given to the output screen. printf() is a standard printing function that has already been defined. The information contained inside the parentheses is known as arguments.
6. Line 6 - Indicates the end of the code block.

Before we get into the details and examples, there’s one thing to keep in mind. The C language is case sensitive, which means it distinguishes between uppercase and lowercase letters. For example, printf, PRINTF and PrintF are not the same.

From Example 1.2.1, it is evident that the c program follows a structure. Let us discuss the structure of a C program.

### 1.2.3 Basic structure of a C program

A ‘C’ program can be considered as a collection of building blocks. Each C program consists of one or more sections, as shown in fig. 1.2.1.

The **documentation section** is optional in a C program. It is better if you include it while programming. This section contains comments.

// → single-line comment

/\*.....

.....\*/ → multi-line comment

The compiler completely ignores this section. This section increases the program readability and understandability. The documentation section can include

- ◆ Program name
- ◆ Author
- ◆ Date of development

This section is used for future reference.

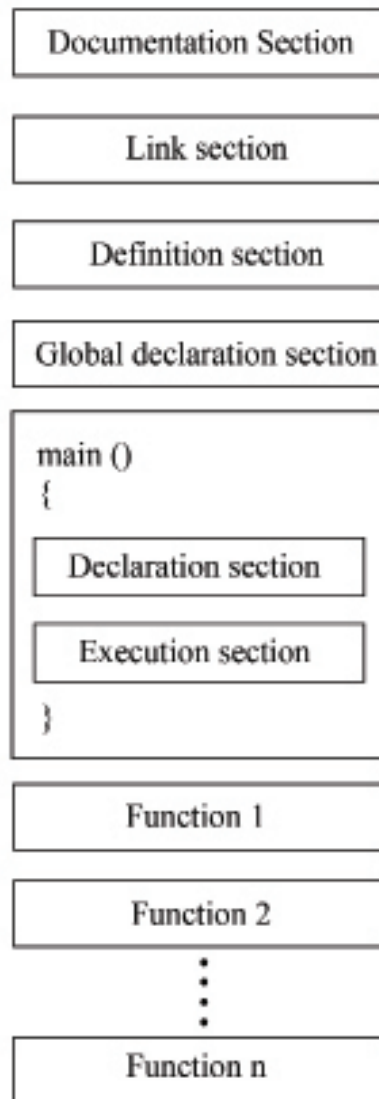


Fig 1.2.1 Structure of a C program

All library files included in the program are mentioned in the **link section**. The compiler is informed about the library files to be linked with the problem. This process is known as linking. All statements belonging to this section will start with “ #include<text belong to the section>”. Where “#” a preprocessor directive.

For example, #include<stdio.h> will link the program with standard input-output function.

All the symbolic constants used in the

program are defined in the **Definition section**.

Example : #define **PI** = 3.14

All the variables used in more than one section are called global variables. All global variables should be declared before the main, and it is declared in the **Global declaration**. It can be used anywhere in the program. Declaration of all user-defined functions is made here.

All C programs must have precisely



one **main** function. All programming statements belonging to the main function must appear between the opening and closing braces of main function. Each statement ends with a ‘;’. The main section has two parts:

1. Declaration part, where the variables are declared here and used later.
2. Execution part implements the main logic that solves the problem

All user-defined functions are defined in the **User-Defined Function Section**. A ‘C’ program can have 0 to any number of user-defined function. It is generally defined after the main function, but it can

text editor or IDE (integrated development environment) to write a c program. C program has to be saved with an extension of “.c” (ex: first.c). The program is known as source program or source code.

There are several phases involved in executing a C program. These are:

1. Creating the program
2. Compiling the program
3. Linking the program with functions that are needed from the c library and
4. Executing the program

Figure 1.2.2 depicts the process of compiling and running a C program. Although

#### Main function

- ◆ The main function is part of every C program
- ◆ In C, the following forms of the main function are allowed
  - a. main()
  - b. int main()
  - c. void main()
  - d. main(void)
  - e. void main(void)
  - f. int main(void)
- ◆ Users allowed to use exactly one main function.

be defined anywhere.

### 1.2.4 Execution of C program

Do you ever wonder how a C program is executed inside a computer? In this section, we can discuss the execution of a c program. First of all, we need to save a c program on the computer. You can use any

these stages remain the same regardless of the operating system, system commands for implementing these procedures and file naming conventions may alter between systems.

Let’s have an overview of the compilation process.



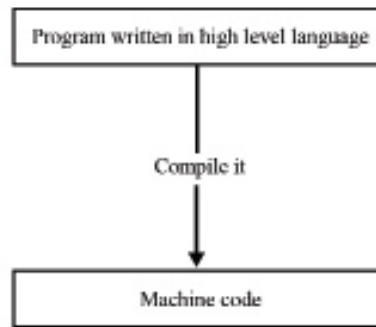


Fig.1.2.2 Compilation overview

Figure 1.2.3 is self-explanatory. The program written in C will be compiled and converted to binary/machine code. Is the binary code sufficient to get the output?

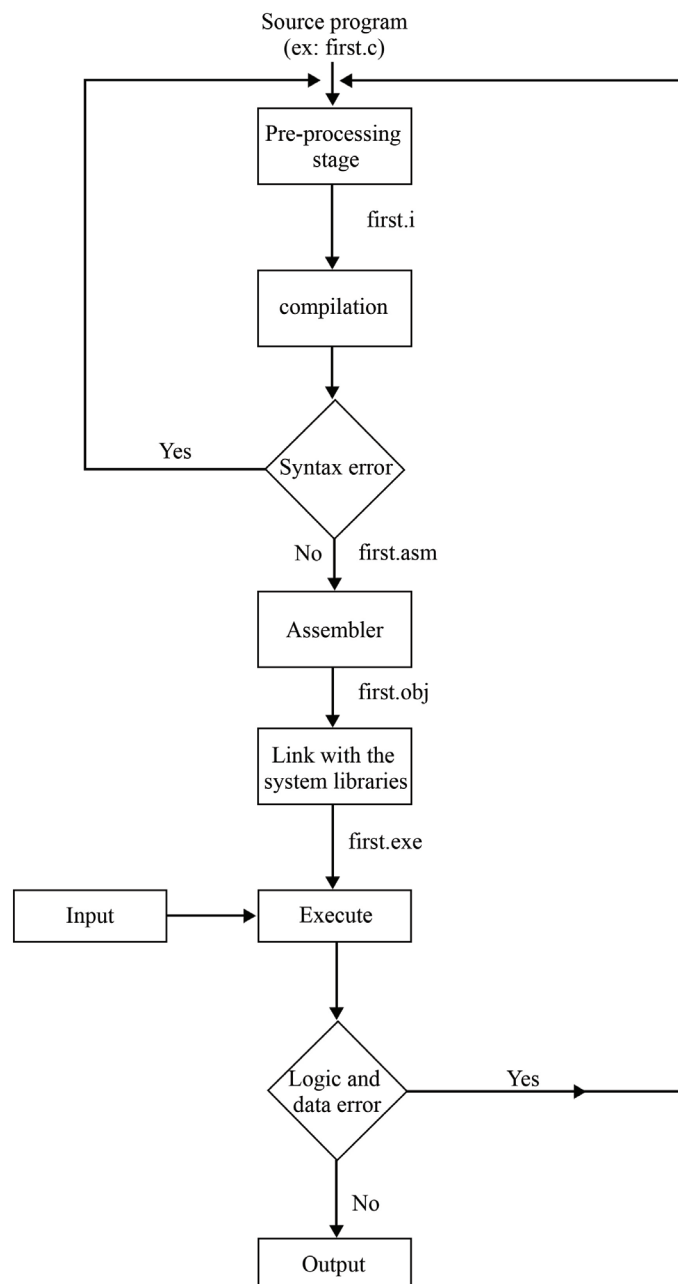


Fig. 1.2.3 Process of compiling and execution of a C program

### 1.2.5 Compiling and linking

Let us assume that the name of the source program is "first.c". The compilation process deals with various files and also produces a number of files. In the compilation process, we mainly deal with four important files.

The preprocessor will replace the lines starting with '#' with the function definition. In this phase, the source code will be expanded and an intermediate code will be generated and the code is saved with an extension '.i'.

The intermediate code will be compiled by a compiler. It will check for syntax

errors. If there is any error it will produce an error message and we need to change the source code to fix the error, and after that the preprocessor will proceed with the compilation process. If there is no syntax error an assembly file will be generated from the intermediate code with the help of the compiler.

The assembler will generate an object code from the assembly file. The linker will link object code into one file called the executable file and will link with system libraries. The loader will load the file into the main memory. The linker will link object code with system library to generate a file called executable file.

## Recap

- ◆ C is a structured programming language.
- ◆ Every C program must have a main function.
- ◆ C follows very strict syntax.
- ◆ Every statement in C language must end with a semicolon.
- ◆ The programs in high level language are written using an editor.
- ◆ All C programs shall be saved with a filename with extension ".c"
- ◆ The compiler scans the program, generates error messages if any, and converts the program to machine code if it is syntactically correct.

## Objective Type Questions

1. Which is the function that we write in all C programs?
2. Where do we declare the variables in C?
3. Who developed the C language?
4. Why do we need to use comments in programs?
5. What will happen when you remove the semicolon from the end of a statement?

6. What is the extension given for C program file names?
7. What is the output of a compiler?
8. Where can you find the output of a C program?
9. What type of error occurs if you type a wrong statement in C?

## Answers to Objective Type Questions

1. Main function
2. Typically declared at the beginning of a block of code, before any executable statements. However, at any points within a block of code.
3. Dennis Ritchie
4. It improves the readability of the program
5. It will lead to compilation error
6. .c
7. Machine level language/ binary code
8. Console
9. Syntax error

## Assignments

1. Write a C program to print address in the following form  
Name : First name of a person  
House Name : name  
Pin code : pin code
2. Describe the basic structure of the C programming language.
3. How to compile a C program?
4. What is main() of a c program? Explain its relevance.
5. How to quote comments in C programming language.
6. Write a C program to print a message.
7. How to read and write data in C programming.

## Reference

1. Balagurusamy, E. Programming in C. Tata McGraw-Hill Education, 2008.
2. Forouzan, Behrouz A. "Computer Science, A Structured Programming Approach Using C, Brooks." Cole. California, 2001.
3. Kernighan, Brian W., and Dennis M. Ritchie. The C programming language. Prentice Hall, Englewood Cliffs, N.J., 1988.
4. Gottfried, Byron S. Schaum's Outline of Programming in Structured Basic. McGraw-Hill Professional, 1992.



## Variables and Data Types

### Learning Outcomes

After the successful completion of the unit, the learner will be able to:

- ◆ understand the Character set in C programming
- ◆ study tokens in C Programming
- ◆ learn the uses of variables to manipulate data to solve a real-world problem

### Prerequisites

Communicating with a computer includes using the language that the computer knows. You know such languages are known as programming languages. We started learning C language in the previous unit.

Learning the English language and learning the C language are, however, somewhat similar. For learning English, we must learn the alphabet of the language, then combine these alphabets to form words, which are then combined to form sentences, and sentences are then combined to form paragraphs.

Learning C is a lot like learning English, except that it is a lot more easy! Rather than learning how to write programmes right away, we must first understand what alphabet, numbers, and special symbols are used in C, how to use them, how constants, variables, and keywords are created, and how all of these are combined to form an instruction. A group of instructions in proper sequence, will form a program.

So, let us start learning the basic constructs of C language. You need to learn a bunch of programming tools in the coming semesters to become a successful programmer. Many languages, like C++ and Java use almost similar character sets, variable types and naming rules as that of C. Hence, if you could understand the concepts well here, your basics will be sound enough to make you an IT professional.



# Key Concepts

Character set, Variables, Constants, Data types

## Discussion

### 1.3.1 C character set

As each language has vocabulary and grammar, and so does C, the character set can form words, numbers and expressions depending on the machine on which the program runs. The C consist of a broad set of characters called the “ C character set”, which includes

- ◆ Letters: it can be uppercase or lowercase
- ◆ Digits: ranging from 0 to 9

- ◆ White space
  - a. Horizontal tab
  - b. Vertical tab
  - c. Blank space
  - d. Newline

The compiler ignores white space unless it is part of a string constant.

- ◆ Special characters:

Special characters used in c is shown in table 1.3.1

Table 1.3.1 Special characters used in C

,(comma)	{ (opening curly bracket)
. (period)	} (closing curly bracket)
;(semi-colon)	[ (left bracket)
:(colon)	] (right bracket)
((opening left parenthesis)	? (question mark)
)(closing right parenthesis)	' (apostrophe)
“(double quotation mark)	& (ampersand)
!(exclamation mark)	^ (caret)
(vertical bar)	+ (addition)
/(forward slash)	- (subtraction)
\ (backward slash)	* (multiplication)
~ (tilde)	/ (division)
_ (underscore)	> (greater than or closing angle bracket)
\$ (dollar sign)	< (less than or opening angle bracket)
% (percentage/modulus sign)	# (hash sign)

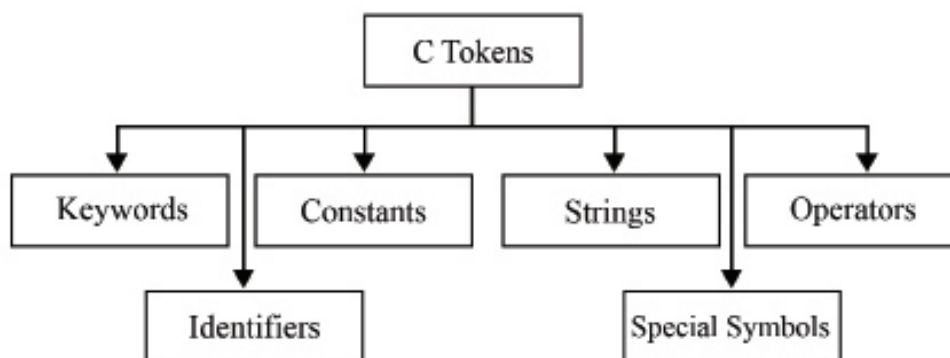


Fig. 1.3.1 C tokens and examples

### 1.3.2 C tokens

When you read a paragraph, you may notice punctuation marks and words. What do you call them? They are called tokens. In the C program, the smallest individual units are called tokens. The C tokens are classified into the following categories.

### 1.3.3 Keyword and Identifiers

All words in C are classified into keywords and identifiers. Each keyword has predefined meaning and use. The

set of keywords are defined when the compiler is developed. Keywords can not be used for any other purpose. There are 32 keywords in C, and they are always written in lowercase. (Refer to table 1.3.2)

Identifiers are user-defined names given to functions, pointers, variables, arrays etc. An identifier consists of a sequence of letters, digits or underscore ( \_ ).

**Example:** number1, num\_1, a, b1, Add, SUB etc.

Table 1.3.2 Keywords in C language

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

### Rules for naming Identifiers

1. The first character should be an alphabet or an underscore
2. It must have only alphabets, digits and underscore
3. Keywords cannot be used as an identifier
4. No whitespaces are allowed

### 1.3.4 Constants

Constants in C are fixed values that are used in a program and whose value remains constant during the program's execution.

Constants in C can be divided into two groups:

- ◆ Primary Constants
- ◆ Secondary Constants

At this stage, we'll just talk about Primary Constants, such as Integer, Real, and Character Constants. Let's take a closer look at each of these constants. Specific rules have been developed for the construction of these various types of constants. The following are the rules:

#### Integer constants

Integers are counting numbers, either

positive or negative. Integer constants can be expressed in three forms- decimal Integer, Octal Integer, Hexadecimal Integer

#### Decimal Integer

- ◆ Consist of digits 0 to 9 with an option of + or -
- ◆ Example: 466, 677, -90, +80
- ◆ White spaces, comma, characters are not allowed in decimal integers

**Hexadecimal integer: Hexadecimal integers are base-16 numbers.**

- ◆ Hexadecimal Integers are preceded with 0x or 0X
- ◆ It includes numbers as well as alphabets from A to F
- ◆ A to F represent digits from 10 to 15
- ◆ Example: 0x12, 0XA, 0X45

**Octal integer: Octal integers are numbers in base-8 format.**

- ◆ Consist of any combination of digits 0 to 7
- ◆ Example: 24, 23.

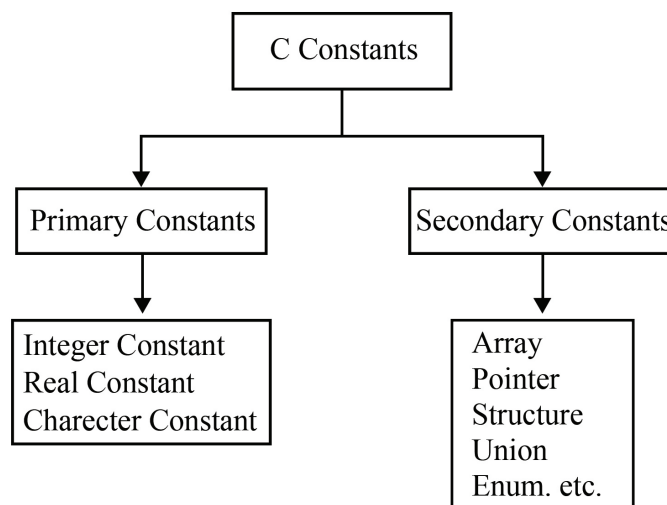


Fig. 1.3.2 Classification of constants

## Real constants

Integer constants are insufficient to represent continuous quantities, like temperature, price etc. To represent such quantities having fractional parts Floating-point numbers or real numbers are used.

- ◆ Example: 12.34, 3.14

## Character constants

The character constants are further divided into:

- ◆ Single character constant
- ◆ String constant
- ◆ Special character constants

### Single character constant

It consists of a single character enclosed inside a pair of single quotes. It is worth

noting that the character '8' isn't the same as the number 8.

Example: 'x', '8'.

## String constant

Letters, numbers, special characters, and blank spaces can all be found in this series of characters enclosed in double-quotes. It is worth noting that "G" and 'G' are not the same things; "G" represents a string because it's contained in a pair of double quotes, while 'G' represents a single character. Another example for string constant is "Hello World".

## Special character constants

C supports special character constants which use '\ ' and are used in the output functions.

Table 1.3.3 Special character constants

Constants	Meaning
\a	beep sound
\b	backspace
\f	form feed
\n	new line
\r	carriage return
\t	horizontal tab
\v	vertical tab
\'	single-quote
\"	double quote
\\	Backslash
\0	Null

### 1.3.5 Variables

We usually do a lot of calculations in any C program. The results of these calculations are saved in the computer's memory. The computer's memory, like human memory, is made up of millions of cells. These memory cells store the measured values. These memory cells (also known as memory locations) are assigned names to make retrieving and using these values easier. The names given to these locations

types? We classify the data based on their characteristics.

Similarly, 'C' offers a variety of data types to make it simple for a programmer to choose the right data type for an application's needs. The three data forms are as follows:

- a. Primitive data types
- b. Derived data types
- c. User-defined data types

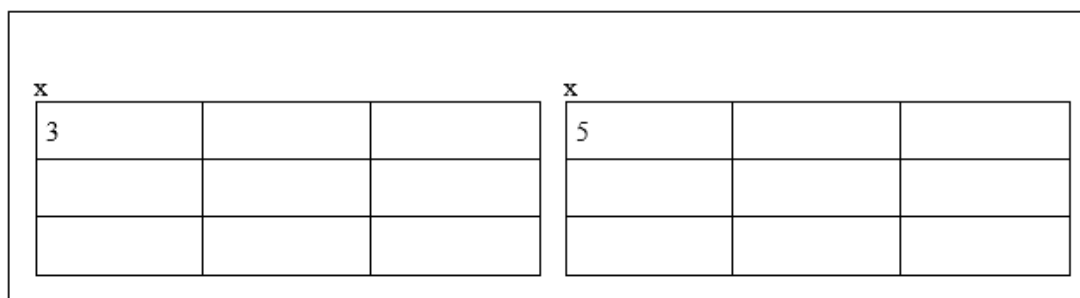


Fig. 1.3.3 Storing values in memory (Variables)

are called variable names because the value stored in each location may change.

Let us look at an example to help us understand.

Take a look at the memory positions in Figure 1.3.3. Here, 3 is saved in a memory location and assigned the name x. Then, we gave the same memory location x a new value of 5. Since a memory location can only hold one value at a time, this will overwrite the previous value 3.

x is defined as a variable since the position with the name x will hold various values at different times.

### 1.3.6 Data types

In our day to day life, we deal with different types of numbers like positive numbers, whole numbers, floating-point numbers etc. What is the meaning of these

The primitive data type is further divided into five:

- ◆ int for integer data
- ◆ char for character data
- ◆ float for floating-point numbers
- ◆ double for double-precision floating-point numbers
- ◆ void

#### Integer (int) type

A whole number is called an integer. Integer data types have a range of -32768 to 32767 as their normal range. However, the range for an integer data type varies based on machines. It can store whole numbers without decimals.

An integer is usually 2 or 4 bytes long, and requires 16 or 32 bits of memory respectively.

The short, long, signed and unsigned are data type modifiers that can be used with some primitive data types (eg: int). The data type modifiers are used to change the size or length of the data type.

Example: short int, long int

### **Floating-point (float) type**

We can use floating-point data type in C programmes, just like integers. The keyword 'float' represents the floating-point data form. It can store a floating-point value, which is a number that includes both a fraction and a decimal component. A real number with a decimal point is called a floating-point value. The size of single precision float data type is 4 bytes, i.e 32 bits.

Since the integer data form does not store

the decimal part of a value, we may use float datatype to store it.

### **Double type**

Double data type in C occupies 8 bytes (64 bits) of memory. It can store double size than float.

### **Void type**

A void data form has no value and does not return any value. In 'C,' it is often used to define functions.

### **Character type**

A single character value enclosed in single quotes is stored in a character data type. A single byte of memory is needed for a character data type. The size of both unsigned and signed char is 1 byte, i.e 8 bits.

## **Recap**

- ◆ A token is the smallest unit in a program.
- ◆ A keyword is reserved word by language.
- ◆ There are a total of 32 keywords.
- ◆ Do not use underscore as the starting symbol of a variable.
- ◆ No limit on the number of characters in an identifier, but advised to use a maximum of 31 characters.
- ◆ Keywords are not allowed to be used as identifiers.
- ◆ Meaningful identifier's name will improve the readability of the program, and it will avoid confusions.
- ◆ The data storage format in which a variable can store data to perform a given action is described as a data type in C.
- ◆ C supports 5 basic data types.
- ◆ Do not combine declaration and execution statements.



## Objective Type Questions

1. What is the size of an int data type?
2. How many keywords are there in C?
3. What are the different types of qualifiers that an int can have at the same time?
4. What is the difference between variables and symbolic names?
5. A memory location can hold how many values at a time.
6. Double can hold how many bits of decimal numbers.

## Answers to Objective Type Questions

1. 2 to 4 bytes depends on the system/compiler
2. 32
3. signed, unsigned, short, unsigned short
4. When a variable is declared, an object instance is produced. A symbolic name is simply defined as a name that may be utilized in a program.
5. One
6. 64

## Assignments

1. Explain character set in C programming language.
2. What is a variable? How to declare and initialize a variable?
3. Explain different types of constants in the C programming language.
4. Explain C tokens.
5. Write a C program to find the area of a circle using constant  $\pi=3.14$ .
6. What are the different types of qualifiers that an int can have at the same time?

## Reference

1. Balagurusamy, E. Programming in C. Tata McGraw-Hill Education, 2008.
2. Forouzan, Behrouz A. "Computer Science, A Structured Programming Approach Using C, Brooks." Cole. California, 2001.
3. Kernighan, Brian W., and Dennis M. Ritchie. The C programming language. Prentice Hall, Englewood Cliffs, N.J., 1988.
4. Gottfried, Byron S. Schaum's Outline of Programming in Structured Basic. McGraw-Hill Professional, 1992.



# Operators and Expressions

## Learning Outcomes

After the successful completion of the unit, the learner will be able to:

- ◆ learn to manipulate data to get the desired output
- ◆ understand different Operators in C Programming
- ◆ use operators in different programming cases

## Prerequisites

What is the meaning of the following mathematical statement?

$$12 + 10 = 22$$

Here two numbers (10 and 12) are added to get a result (22). Here the '+' symbol indicates the addition operation, and these types of symbols are known as mathematical operators. In the same way, we need operators to perform operations on the data stored in a computer. In this section, we are going to discuss the operators and expressions used in C programming.

You might be familiar with arithmetic operations like addition, subtraction, multiplication and division. While writing mathematical expressions, we use symbols like +, -, × and ÷ to represent these operations. Similarly, we need to use operator symbols to represent various operations, when we write programs.

We have seen that computations on numbers are essential to computers. Let us drive into learning about various types of operators and how to construct mathematical expressions in C.

## Key Concepts

Operators, Operator precedence, Expressions



## Discussion

### 1.4.1 Operators in C

An operator is used to perform specific mathematical or logical operations on values. The values that the operators work on are called operands. Let us consider an example:

$x * 230 \rightarrow \text{expression}$

The  $x * 230$  is an expression in C language, and the variable  $x$  and value 230 are the operands. The '\*' (star/asterisk) sign is an operator. The C programming language has a large number of built-in operators, and they are classified into the following types:

- ◆ Arithmetic operators
- ◆ Relational operators

- ◆ Logical operators
- ◆ Bitwise operators
- ◆ Assignment operators
- ◆ Conditional operators
- ◆ Special operators

### 1.4.2 Arithmetic operator

A mathematical function that accepts two operands and performs a computation on them is known as an arithmetic operator. They are frequent in everyday math, and most computer languages provide a collection of them that can be employed in equations to do a variety of sequential calculations. The C language provides the basic arithmetic operators, and they are listed in table 1.4.1.

Table 1.4.1. Arithmetic operators

Operator	Operation	Description	Example
+	Addition	Adds the two numeric values on either side of the operator	<pre>#include&lt;stdio.h&gt; #include&lt;conio.h&gt; void main() {     int a=5, b=7;     printf("the sum = %d," a+b);     getch(); }</pre>
-	Subtraction	Subtracts the operand on the right from the operand on the left	<pre>#include&lt;stdio.h&gt; #include&lt;conio.h&gt; void main() {     int a=5, b=7;     printf("the difference between %d and %d = %d",b,a b-a);     getch(); }</pre>

*	Multiplication	Multiplies the two values on both sides of the operator	<pre>#include&lt;stdio.h&gt; #include&lt;conio.h&gt; void main() { int a=5, b=7; printf("the product = %d," a*b); getch(); }</pre>
/	Division	Divides the operand on the left by the operand on the right and returns the quotient	<pre>#include&lt;stdio.h&gt; #include&lt;conio.h&gt; void main() { int a=9, b=2; printf("the quotient = %d," a/b); getch(); }</pre>
%	Modulus	Divides the operand on the left by the operand on the right and returns the remainder	<pre>#include&lt;stdio.h&gt; #include&lt;conio.h&gt; void main() { int a=9, b=2; printf("the remainder = %d," a%b); getch(); }</pre>

### 1.4.3 Relational Operator

We frequently compare two quantities and make decisions based on their relationship. For example, we may compare the prices of two items or the time taken to finish certain work and so on. In C, rela-

tional operators are used for comparison. The values of the operands on either side of the relational operator are compared to determine the relationship between them. The relational operators supported by C are listed in Table 1.4.2

Table 1.4.2 Relational operators

Operator	Operation	Example
==	Equal to	If the values of two operands are equal, then the condition is True, otherwise, it is False
!=	Not equal to	If the values of two operands are not equal, then the condition is True. Otherwise, it is False
>	Greater than	If the value of the left-side operand is greater than the value of the right side operand, then the condition is True, otherwise, it is False
<	Less than	If the value of the left-side operand is less than the value of the right side operand, then the condition is True. Otherwise, it is False
>=	Greater than or equal to	If the value of the left-side operand is greater than or equal to the value of the right-side operand, then the condition is True. Otherwise, it is False
<=	Less than or equal to	If the value of the left operand is less than or equal to the value of the right operand, then it is True. Otherwise it is False

#### 1.4.4 Assignment operator

The assignment operator changes or assigns the value of the variable to the left of it.

Table 1.4.3 Assignment operator

Operator	Description
=	Assigns value from right-side operand to left side variable
+=	It adds the value of the right-side operand to the left-side operand and assigns the result to the left-side operand Note: $x += y$ is the same as $x = x + y$



<code>-=</code>	It subtracts the value of the right-side operand from the left-side operand and assigns the result to the left-side operand Note: <code>x -= y</code> is the same as <code>x = x - y</code>
<code>*=</code>	It multiplies the value of the right-side operand with the value of the left-side operand and assigns the result to the left-side operand Note: <code>x *= y</code> is the same as <code>x = x * y</code>
<code>/=</code>	It divides the value of the left-side operand by the value of the right-side operand and assigns the result to the left-side operand Note: <code>x /= y</code> is the same as <code>x = x / y</code>
<code>%=</code>	It performs modulus operation using two operands and assigns the result to left-side operand Note: <code>x %= y</code> is the same as <code>x = x % y</code>

Let us consider a simple C program snippet to understand the assignment operators

```
int a = 21;

int c ;

c = a;

printf("Line 1 - = Operator Example, Value of c = %d\n", c );

c += a;

printf("Line 2 - += Operator Example, Value of c = %d\n", c );

c -= a;

printf("Line 3 - -= Operator Example, Value of c = %d\n", c );

c *= a;

printf("Line 4 - *= Operator Example, Value of c = %d\n", c );

c /= a;

printf("Line 5 - /= Operator Example, Value of c = %d\n", c );

c = 200;

c %= a;

printf("Line 6 - %= Operator Example, Value of c = %d\n", c );
```

### Output:

Line 1 - = Operator Example, Value of c = 21

Line 2 - += Operator Example, Value of c = 42

Line 3 - -= Operator Example, Value of c = 21

Line 4 - \*= Operator Example, Value of c = 441

Line 5 - /= Operator Example, Value of c = 21

Line 6 - %= Operator Example, Value of c = 11

Example 1 program (source: internet)

Table 1.4.4 Logical operators

Operator	Operation	Description
&&	Logical AND	If both the operands are true, then the condition becomes True
	Logical OR	If any of the two operands are True, then the condition becomes True
!	Logical NOT	Used to reverse the logical state of its operand

### 1.4.5 Logical operator

Logical operators are used when we want to check multiple conditions to make a decision. The C language supports 3 logical operators. Based on the operands on either side, the logical operator evaluates to True or False. Every value is either True or False logically. Except for 0 (zero), all values are True by default. The logical operators used in C are listed in Table 1.4.4.

### 1.4.6 Increment and decrement operator

To change the value of an operand by one, C programming offers two operators: increment “++” and decrement “--” The increment operator increases the value by one, while the decrement operator reduces it by one. These two operators are unary, which means they only work with one operand.

Table 1.4.5 Increment or decrement operators

Operator	Operation	Description	Example
++	Increment	Operates on a single value. Increases the value of the integer operand by 1	<pre>#include&lt;stdio.h&gt; #include&lt;conio.h&gt; void main() { int a=9, b; b = ++a; printf("the output of increment operator = %d", b); getch(); }</pre>
--	Decrement	Operates on a single value. Decreases of the value of the integer operand by 1	<pre>#include&lt;stdio.h&gt; #include&lt;conio.h&gt; void main() { int a=9, b; b = --a; printf("the output of increment operator = %d", b); getch(); }</pre>

### Rules for increment and decrement operator

1. Increment and decrement operators are unary operators.
2. They need variables as their operand.
3. When a variable is used in an expression with a postfix increment or decrement (eg: a++/a--), the expression is evaluated first using the initial value of the variable, then update the value of the variable by one.
4. When the prefix increment or decrement (eg: ++a/--a) is used with a variable in an expression, the value of the variable is updated first, and then the expression is evaluated.

Are you confused with the 3rd and 4th rules? Well, let us understand it through an example code.

## Example 1

### Program

```
int main()
{
    int x = 10, a, y = 10, b;

    a = ++x;
    b = y++;
    printf("Pre Increment Operation");
    // Value of a will change
    printf( "\na =%d", a);
    // Value of x changes before execution of  a=++x;
    printf ( "\nx = %d" , x);
    printf("Post Increment Operation");
    // Value of b will be assigned first, then the value of y will get incremented.
    printf( "\nb =%d", b);
    // Value of y changes after assigning value to b, b=y++;
    printf ( "\ny = %d" , y);
    return 0;
}
```

### Output

Pre Increment Operation

a = 11

x = 11

Post Increment Operation

b =10

y = 11

### 1.4.7 Conditional operator

The conditional operator is otherwise known as the ternary operator. It needs three operands to work on. The syntax of the ternary operator is:

**expression 1 ? expression 2: expression 3;**

- ◆ The question mark “?” in the syntax to check the condition.
- ◆ The first expression (expression 1) usually returns true or false, which determines whether (expression 2) will be executed or otherwise (expression 3)
- ◆ If (expression 1) is valid, the expression on the left side of “:”, i.e. (expression 2), is executed.
- ◆ If expression 1 returns false then expression 3 will be executed

For example, consider the following statements

v1 = 5;

v2 = 20;

Result = v1 > v2 ? v1 : v2;

The result will be assigned the value of v2 ie, 20

### 1.4.8 Bitwise operator

Data is manipulated at the bit level with bitwise operators. Shifting bits from right to left is often performed by these operators. Float and double data types do not support bitwise operators.

#### 1.4.8.1 The & (bitwise AND)

The & (bitwise AND) in C takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.

The output of bitwise AND is 1 if the corresponding bits of two operands are 1. If either bit of an operand is 0, the result of the corresponding bit is evaluated to 0.

Let us suppose the bitwise AND operation of two integers 12 and 25.

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bit Operation of 12 and 25



```

00001100
& 00011001
-----
00001000 = 8 (In decimal)

```

### Example Program: Bitwise AND

```

#include <stdio.h>

int main() {
    int a = 12, b = 25;

    printf("Result = %d", a&b);

    return 0;
}

```

### Output

Result = 8

### 1.4.8.2 The | (bitwise OR)

The | (bitwise OR) in C takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1.

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bitwise OR Operation of 12 and 25

```

00001100
| 00011001
-----
00011101 = 29 (In decimal)

```

### Example Program: Bitwise OR

```

#include <stdio.h>

int main() {

```

```

int a = 12, b = 25;

printf("Result = %d", a|b);

return 0;

}

```

### Output

Result = 29

### 1.4.8.3 The ^ (bitwise XOR)

The ^ (bitwise XOR) in C takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.

12 = 00001100 (In Binary)

25 = 00011001 (In Binary)

Bitwise XOR Operation of 12 and 25

```

00001100
^ 00011001

```

---

00010101 = 21 (In decimal)

### Example Program: Bitwise XOR

```

#include <stdio.h>

int main() {

    int a = 12, b = 25;

    printf("Result = %d", a^b);

    return 0;

}

```

### Output

Result = 21

### 1.4.8.4 The << (left shift)

The << (left shift) in C takes two numbers, left shifts the bits of the first operand, the second operand decides the number of places to shift.





212 = 11010100 (In binary)

212<<1 = 110101000 (In binary) [Left shift by one bit]

212<<0 = 11010100 (Shift by 0)

212<<4 = 110101000000 (In binary) =3392(In decimal)

#### 1.4.8.5 The >> (right shift)

The >> (right shift) in C takes two numbers, right shifts the bits of the first operand, the second operand decides the number of places to shift.

212 = 11010100 (In binary)

212>>2 = 00110101 (In binary) [Right shift by two bits]

212>>7 = 00000001 (In binary)

212>>8 = 00000000

212>>0 = 11010100 (No Shift)

#### Example Program for Bitwise Shift Operations

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

```
int main ()
```

```
{
```

```
    int num = 212, i;
```

```
    for (i = 0; i <= 2; ++i)
```

```
    {
```

```
        printf ("Right shift by %d: %d\n", i, num >> i);
```

```
    }
```

```
    printf ("\n");
```

```
    for (i = 0; i <= 2; ++i)
```

```
    {
```

```
        printf ("Left shift by %d: %d\n", i, num << i);
```

```
    }
```

```
    return 0;
```

```
}
```

## Output

Right shift by 0: 212

Right shift by 1: 106

Right shift by 2: 53

Left shift by 0: 212

Left shift by 1: 424

Left shift by 2: 848

### 1.4.8.6 The ~ (bitwise NOT)

The ~ (bitwise NOT) in C takes one number and inverts all bits of it.

35 = 00100011 (In Binary)

Bitwise complement Operation of 35

~ 00100011

---

11011100 = 220 (In decimal)

### Example Program for Bitwise NOT

```
#include <stdio.h>

int main ()
{
    printf ("Output = %d\n", ~35);
    printf ("Output = %d\n", ~~12);
    return 0;
}
```

## Output

Output = -36

Output = 11

### Points to remember while using Bitwise operators

- ◆ The left shift and right shift operators should not be used for negative numbers.
- ◆ The bitwise OR of two numbers is just the sum of those two numbers, if there is no carry involved, otherwise you just add their bitwise AND.
- ◆ The bitwise XOR operator is the most useful operator. It is used in many



problems. A simple example could be “Given a set of numbers where all elements occur an even number of times except one number, find the odd occurring number” This problem can be efficiently solved by just doing XOR to all numbers.

- ◆ The bitwise operators should not be used in place of logical operators.
- ◆ The & operator can be used to quickly check if a number is odd or even. The value of the expression (x & 1) would be non-zero only if x is odd, otherwise the value would be zero.
- ◆ The ~ operator should be used carefully. The result of the ~ operator on a small number can be a big number if the result is stored in an unsigned variable. And the result may be a negative number if the result is stored in a signed variable (assuming that the negative numbers are stored in 2’s complement form where the leftmost bit is the sign bit)
- ◆ The left-shift and right-shift operators are equivalent to multiplication and division by 2 respectively. As mentioned in point 1, it works only if numbers are positive.

The bitwise shift operator changes the value of a bit. The left operand specifies the value to be moved, while the right operand specifies the number of places the value’s bits must be shifted. The precedence of both operands is the same.

Example:

a = 0001000

b = 2

a << b = 0100000

a >> b = 0000010

Table 1.4.6 Bitwise operators

Operator	Operation	Truth Table				
&   ^ << >>	Bitwise AND	a	b	a&b	a b	a^b
	Bitwise OR	0	0	0	0	0
	Bitwise exclusive OR	0	1	0	1	1
	Left shift	1	0	0	1	1
	Right shift	1	1	1	1	0

Table 1.4.7 Special operators

Operator	Description	Example
sizeof	Returns the size of a variable	sizeof(x) return size of the variable x
&	Returns the address of a variable	&x; return address of the variable x
*	Pointer to a variable	*x; will be a pointer to a variable x

### 1.4.9 Special operator

The C language supports some special operators who are listed in table 1.4.7

### 1.4.10 Expression

A combination of constants, variables, and operators is known as an expression. An expression always yields a result. A value or a standalone variable may also be called expressions, but a standalone operator is not. Below are some valid expression examples:

- Num
- $x = a + b$
- $R = 2 * \text{num} - 3.14$
- $++x$
- $x--$
- $h = \text{"good morning"}$

#### Evaluation of Expression

Let us consider a valid C expression.

$x = a + b;$

Here x, a, b are valid C variables. When the statement is encountered, the expression is evaluated first, and the result will be stored in x. The blank space around the variables and operator is optional, which improves the readability of the program.

### 1.4.11 Precedence and Associativity

The precedence of operators is used to evaluate an expression. When there are several types of operators in an expression, the precedence decides which one should be used first. The higher-priority operator is evaluated first, followed by the lower-priority operator. The majority of the operators we've looked at so far are binary operators. Operators of two operands are known as binary operators. The unary operators have higher precedence than the binary operators since they only require one operand. The operators of the same precedence will be evaluated either from left to right or right to left. depending on the level. This property is known as associativity. The operator precedence and associativity are listed in table 1.4.8

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	()	Function call	
	[]	Array subscript	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(type){list}	Compound literal(C99)	
2	++ --	Prefix increment and decrement	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(type)	Cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of	
	_Alignof	Alignment requirement(C11)	
3	* / %	Multiplication, division, and the remainder	Left-to-right
4	+ -	Addition and subtraction	

5	<< >>	Bitwise left shift and right shift	
6	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	
10		Bitwise OR (inclusive or)	
11	&&	Logical AND	
12		Logical OR	
13	?:	Ternary conditional	Right-to-left
14	=  +=, -=  *=, /=, %=  <<=, >>=  &=, ^=,  =	Simple assignment  Assignment by sum and difference  Assignment by product, quotient, and the remainder  Assignment by bitwise left shift and right shift  Assignment by bitwise AND, XOR, and OR	
15	,	Comma	Left-to-right

Note:

1. Parentheses can be used to override the precedence of operators. The expression within () is evaluated first.
2. For operators with equal precedence, the expression is evaluated from left to right.

### 1.4.12 Type conversion

When we perform operations on two different data types in C, the resulting type is automatically converted to the available higher data type.

Let us understand it through a sample program.

```
#include<stdio.h>

int main(int argc, char const *argv[])
{
    int num_1 = 67;
    int num_2 = 5;
    float num_3 = 5;
    /*Division of int with int the result will be int*/
    printf("Int with Int division = %d \n", (num_1 / num_2) );
    /*Division of int with float the result will be float*/
    printf("Int with Float division = %f \n", (num_1 / num_3) );
    /*Division of int with int type casted with float the result will be float*/
    printf("Int with Int type casted with float division = %f \n", ((float)num_1 / num_2) );
    return 0;
}
```

#### Output:

Int with Int division = 13

Int with float division = 13.400000

Int with Int type casted with float division = 13.400000

num\_1 and num\_2 are integers initialized with values 67 and 5, respectively and num\_3 is a floating point variable initialised with 5.

The first printf statement prints the result of integer division num\_1/num\_2, which is 67 divided by 5 resulting in an integer value 13.



The second printf statement attempts to print the result of an integer value divided by a float value i.e num\_1/num\_3 and the resultant data type will be float.

The third printf statement correctly demonstrates type casting by casting num\_1 to a float before division, ensuring the correct float division result. It divides num\_1 (67) by num\_2 (5) as floats resulting in the floating point value.

## Recap

- ◆ There are different types of operations in programming, like arithmetic operations, logical operations, assignment operations etc.
- ◆ In arithmetic operation, divide by zero is undefined, so avoid chances to make this operation while coding.
- ◆ Usage of increment/decrement operator is tricky.
- ◆ C allows the use of a few special operators other than the common operators in practice.
- ◆ The expression must end with a semicolon.
- ◆ Precedence of operators is used to evaluate an expression
- ◆ Use parentheses to get higher precedence to an operator/operation

## Objective Type Questions

1. Which symbol represents a modulus operator?
2. What will be the value of the expression “int a = 10 + 4.867;” ?
3. What will be the value stored in variable ‘a’, “int a = 3.5 + 4.5;” ?
4. Which is the value stored in ‘var’, “float var = 3.5 + 4.5;” ?
5. What is the output of the following code snippet ?

```
int main()
{
    float c = 3.5 + 4.5;
    printf(“%f”, c);
    return 0;
}
```

6. What is the priority of operators \*, / and % in C language?
7. Associativity of C Operators \*, /, %, +, - and = is.?
8. What is the output of the below program?

```
int main()
{
    int a=0;

    a = 5>2 ? printf("4"): 3;

    printf("%d",a);

    return 0;
}
```

## Answers to Objective Type Questions

1. %
2. a = 14
3. a = 8
4. var = 8.0
5. 8.000000
6. All three operators \*, / and % are the same. But if all three came in the same expression without parentheses it will get evaluated from left to right
7. Operators \*, / and % have Left to Right Associativity. Operators + and - have Left to Right Associativity. Operator = has Right to Left Associativity
8. 41. **Explanation:** 5>2 is true. So expression1 i.e printf("4") is executed printing 4. Function printf() returns 1. So the value is 1.

## Assignments

1. Explain basic arithmetic operations .
2. Explain Logical and bitwise operations
3. Write a program to solve a quadratic equation.
4. What is an operator and an operand . Explain different types of operators in C programming.
5. Write a program to show working of a simple calculator.
6. Explain different relational operators in C programming.
7. What is a conditional operator in C programming? Explain with an example.
8. Write a C program to find the largest among three numbers using conditional operator.

## Reference

1. Balagurusamy, E. Programming in C. Tata McGraw-Hill Education, 2008.
2. Forouzan, Behrouz A. "Computer Science, A Structured Programming Approach Using C, Brooks." Cole. California, 2001.
3. Kernighan, Brian W., and Dennis M. Ritchie. The C programming language. Prentice Hall, Englewood Cliffs, N.J., 1988.
4. Gottfried, Byron S. Schaum's Outline of Programming in Structured Basic. McGraw-Hill Professional, 1992.

```
#include "KMotionDef.h"
```

```
int main()
```

```
{
```

```
    ch0->Jerk = 250f;
```

```
    ch0->output_mode=MICROSTEP_MODE;
```

```
    ch0->Vel=700f;
```

```
    ch0->Accel=500.0f;
```

```
    ch0->Jerk =2000f;
```

```
    ch0->Lead=0.0f;
```

```
    EnableAxisDest(0,0);
```

```
    ch1->Amp = 250;
```

```
    ch1->output_mode=MICROSTEP_MODE;
```

```
    ch1->Vel=700f;
```

```
    ch1->Accel=500.0f;
```

```
    ch1->Jerk =2000f;
```

```
    ch1->Lead=0.0f;
```

```
    EnableAxisDest(1,0);
```

```
    DefineCoordSystem(0,1,-1,-1);
```

```
    return 0;
```

```
}
```

## BLOCK 2

# IO Statements, Control Structures, Arrays, and Pointers



# Input Output Statements

## Learning Outcomes

After the successful completion of the unit, the learner will be able to:

- ◆ get familiarized with input-output functions of C
- ◆ make aware of formatted input/output
- ◆ apply the concepts to build C programs.

## Prerequisites

We all are familiar with calculators. In a calculator, we have an option to give numbers and operations (+, -, /...) and it will display the output on the screen. Can you imagine a calculator without the keys in it? Well, the calculator won't serve the purpose and the desired result will not be available to the user. In this section, we are going to discuss the function that helps us to give input to a C program and get output from it.

Why do we write programs?

We need to read, write and process data to solve problems. So a computer program takes data as input and after processing the data it will display the processed data as output. How does a program get data? In unit 3 we saw the initialization of variables such as `a=3, b=0`; and another technique is to get data from the keyboard. Also we need the output displayed on the screen.

The C programming language includes standard libraries that allow program input and output. Let us see how input/output operations are performed in C language.

## Key Concepts

Formatted and Unformatted Input-Output functions



## Discussion

### 2.1.1 Reading a character

Reading a character from the input unit and writing it to the output unit are the most basic of all input/output processes. The function **getchar()** can be used to read a single character. Only a single character is read at a time by this feature. If you want to read more than one character, you can use this method in a loop. The **getchar** takes the following form:

**Variable\_name = getchar()**

For storing the input from **getchar()** function, the **Variable\_name** must be a **char** type variable. When **getchar()** is encountered, the screen will wait until a key is pressed. Let us consider the code snippet to understand the concept properly.

```
char c;
printf( "Do you want to know my name? Type 'y' for yes 'n' for no");
c = getchar( );
if( c == 'y' || c == 'Y')
printf("I'm a Code Bee");
else
printf("It's okay. Your loss");
```

Program 2.1.1

In program 2.1.1 if the user input is **y/Y** “ I’m a Code Bee” will be displayed in the terminal. If the user inputs **n/N** then “ It’s okay. You loss” will be displayed on the screen.

Before using the **getchar()** function you should be aware of the fact, **getchar()** accepts any character through the keyboard.

The **getchar()** function reads only one character from the terminal. What if you want to read a sentence? Well, C has another input function **gets()**. The **gets()** function reads a line from **stdin** into the buffer pointed to by a string. Let us consider a code snippet to understand the **gets()** function as given in Program 2.1.2 :

```
char str[100];
printf( "Enter any value :");
gets( str );
printf( "\nYou entered: %s", str);
```

Program 2.1.2

### Output

Enter any value :Good Morning

You entered: Good Morning

## 2.1.2 Writing a character

**putchar()** is a function similar to **getchar()** for writing characters one by one to the terminal. It has the following syntax:

**putchar( variable\_name )**

This statement displays the character contained in the variable\_name at the terminal. For example, see the following program 2.1.3.

```
char a;  
  
a = 'y';  
  
putchar(a);
```

program 2.1.3

### Output

y

Output screen will display 'y'

Instead of **putchar()**, C supports another function called **puts()** function. Let us understand the function through a sample program 2.1.4:

```
char str[100];  
  
printf( "Enter a value :");  
  
gets( str );  
  
printf( "\nYou entered: ");  
  
puts( str );
```

Program 2.1.4

### Output

Enter a value :Alan Turing

You entered: Alan Turing





### 2.1.3 Formatted Input

Formatted input means the input data has a particular format. **scanf()** function is used to get the formatted input. **scanf()** is specified in the standard input-output header file **stdio.h**, and is used to display output on the screen and take user input, respectively.

The general syntax of the **scanf** function is

```
scanf(const char *format, ...)
```

The output of this function is written to the regular output stream **stdout**, and it is formatted as specified.

The format can be specified using format specifiers “%s, %d, %c, %f,” for printing or reading data that are string, integer, character or float respectively. There are a variety of other formatting methods that can be used depending on the situation will be listed at the last part of the unit. Let us start with a quick example to help you understand the concepts (refer Program 2.1.5).

```
#include<stdio.h>

void main()
{
    int a, b, c;

    printf("Please enter any two integer numbers: \n");

    scanf("%d %d", &a, &b);

    c = a + b;

    printf("The sum of two numbers is: %d", c);

}
```

Program 2.1.5

#### Output

Please enter any two integer numbers:

2

3

The sum of two numbers is: 5

When you compile the above code, you will be asked to enter a value. When you enter a value, the entered value will be displayed on the screen. You may be wondering why %d is used in the scanf() and printf() functions. It's called the format string, and it tells the scanf() function what kind of input to expect, and it's used in printf() to tell the compiler what kind of output to expect. Details are tabulated in Table 2.1.1.

Table 2.1.1 Format Strings

Format string	Meaning
%d	Scan or print an integer assigned decimal number
%f	Scan or print a floating-point number
%c	To scan or print a character
%s	To scan or print a character string. The scanning ends at whitespace.

### Rules to use scanf

- ◆ A type specification is required for each variable to be read.
- ◆ The scanf reads characters from the terminal until:
  - a. It encounters a whitespace
  - b. It has read the maximum number of characters
  - c. It reaches the end of the document

If the return value is not 1, an error message is printed to the console and the program exits with an error code.

### 2.1.4 Formatted output

So far, we have seen that the printf function is used to print the processed data to the screen. The printf is a necessary and easy function for the programmer to print data in a formatted way. How can we change the appearance of output data? Let us consider an example C program to understand the formatted output (refer Program 2.1.6).

```
printf("The color: %s\n", "blue");

printf("First number: %d\n", 12345);

printf("Second number: %04d\n", 25);
```

```
printf("Third number: %i\n", 1234);
printf("Float number: %3.2f\n", 3.14159);
printf("Hexadecimal: %x\n", 255);
printf("Octal: %o\n", 255);
printf("Unsigned value: %u\n", 150);
printf("Just print the percentage sign %%\n", 10);
```

Program 2.1.6

#### Output

The color: blue  
 First number: 12345  
 Second number: 0025  
 Third number: 1234  
 Float number: 3.14  
 Hexadecimal: ff  
 Octal: 377  
 Unsigned value: 150  
 Just print the percentage sign %

Let us have a cheat sheet on printf formatting as shown in Table 2.1.2.

Table 2.1.2 Formatted Integer

Code	Output
printf("%3d", 0);	0
printf("%3d", 123456789);	123456789
printf("%3d", -10);	-10
printf("%3d", -123456789);	-123456789

When used for integers, the %3d specifier indicates a minimum width of three spaces, which is right-justified by default. Simply add a minus sign (-) after the percent symbol in printf to left-justify integer performance, as seen in Table 2.1.3.

Table 2.1.3 Left-justify Integer

Code	Output
printf(“%-3d”, 0);	0
printf(“%-3d”, 123456789);	123456789

Formatted floating point numbers with printf are shown in Table 2.1.4.

Table 2.1.4 Numeric Justification

Description	Code	Result
Print one position after the decimal	printf(“”%.1f””, 10.3456);	‘10.3’
Two positions after the decimal	printf(“”%.2f””, 10.3456);	‘10.35’
Eight-wide, two positions after the decimal	printf(“”%8.2f””, 10.3456);	‘ 10.35’
Eight-wide, four positions after the decimal	printf(“”%8.4f””, 10.3456);	‘ 10.3456’
Eight-wide, two positions after the decimal, zero-filled	printf(“”%08.2f””, 10.3456);	‘00010.35’
Eight-wide, two positions after the decimal, left-justified	printf(“”%-8.2f””, 10.3456);	‘10.35 ’
Printing a much larger number with that same format	printf(“”%-8.2f””, 101234567.3456);	‘101234567.35’

Formatted string using printf is given in Table 2.1.5

Table 2.1.4 Numeric Justification

Description	Code	Result
A simple string	<code>printf(“”%s”, “Hello”);</code>	‘Hello’
A string with a minimum length	<code>printf(“”%10s”, “Hello”);</code>	‘   Hello’
Minimum length, left-justified	<code>printf(“”%-10s”, “Hello”);</code>	‘Hello   ’

## Recap

- ◆ `getchar()` function reads any character from the input stream
- ◆ Standard input-output (`stdio. h`) header file has `printf()` and `scanf()` function
- ◆ The format specifier is important for any variable to read or print.

## Objective Type Questions

1. What is the purpose of `scanf()`?
2. What is the output of the C program?

```
#include <stdio.h>

int main()
{
    printf(“variable! %d”, x);
    return 0;
}
```

3. What is the output of the program given below?

```
#include <stdio.h>

int main()
```

```

{
    int main = 3;
    printf("%d", main);
    return 0;
}

```

4. Suppose we input a=1 and b=2 then what is the output of the following code?

```

#include <stdio.h>

int main()
{
    int a, b;
    printf("%d", scanf("%d %d",&a,&b));
    return 0;
}

```

5. What is the output of the program ?

```

#include <stdio.h>

void main()
{
    printf("hello\rworld\n");
    printf("hello\b\b\bworld\n");
}

```

6. what is the output of the statement “printf ( “%d” , printf ( “hello” ) );” ?  
 7. Which formatted string is used to scan or print a character string?

## Answers to Objective Type Questions

1. It reads data from the terminal.
2. It gives a **compilation error** since the x is not declared
3. 3
4. 2

5. world  
    heworld
6. Hello5
7. %s

## Assignments

1. How to read a character from a console in C programming language.
2. Explain putchar() and puts() function in C programming language.
3. Write a program in C, read your name, designation, id card number and write these details.
4. Explain formatted input and output in C programming language.
5. Write a C program to read a number and print a multiplication table of that number.
6. What are the different header files used for read and write in C programming.

## Reference

1. Balagurusamy, E. Programming in C. Tata McGraw-Hill Education, 2008.
2. Forouzan, Behrouz A. "Computer Science, A Structured Programming Approach Using C, Brooks." Cole. California, 2001.
3. Kernighan, Brian W., and Dennis M. Ritchie. The C programming language. Prentice Hall, Englewood Cliffs, N.J., 1988.
4. Gottfried, Byron S. Schaum's Outline of Programming in Structured Basic. McGraw-Hill Professional, 1992.



# Control Structures and Looping

## Learning Outcomes

After the successful completion of the course, the learner will be able to:

- ◆ study the aspects of condition checking in C Programming.
- ◆ make aware of the working of nesting of if Statements.
- ◆ know about switch case statements.
- ◆ get familiarized with different loop concepts and apply it to programming.
- ◆ make aware of break and continue Statements.
- ◆ have knowledge in the area of nested loops.

## Prerequisites

In Figure 2.2.1, shown below, we all are familiar with the maze game. There is only one entry point and one way to reach the goal, and we may need to find the path towards the goal. The player has no choice but to follow the path in the maze to reach the goal.

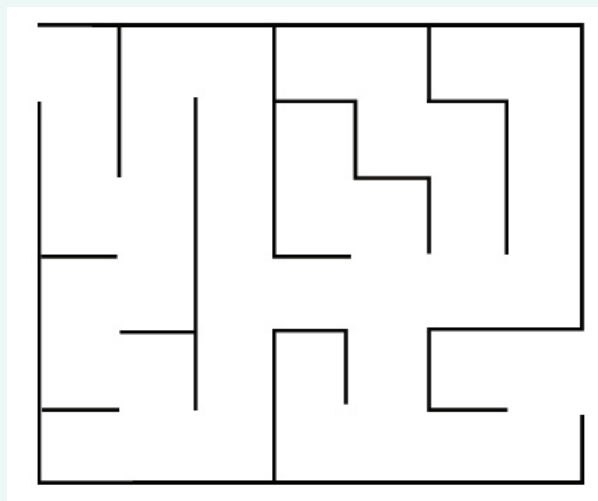


Figure 2.2.1 Maze Game





Is this similar to the concept of sequencing in C? In the unit dealing with the execution of C programming, we learned the concept of sequence. Sequence means C executes one statement after another from beginning to the end of the program.

Let us consider the below C program.

<pre>/* program to print the difference between two numbers*/  #include &lt;stdio.h&gt;  #include &lt;conio.h&gt;  void main() { int num1, num2, diff;  printf("Enter the first number\t:\t"); scanf("%d",&amp;num1);  printf("Enter the second number\t:\t"); scanf("%d", &amp;num2);  diff = num1 - num2  printf("The difference between %d and %d is %d", num1, num2, diff);  getch(); }</pre>	<p><b>Output:</b></p> <p>Enter the first number : 8</p> <p>Enter the second number : 5</p> <p>The difference between 8 and 5 is 3</p>
---	---

Program 2.2.1

This program will be executed in sequence, which means the commands will be executed one by one in the order specified in the program. The order in which statements in a program are executed is referred to as control flow or flow of control. Control systems can be used to implement control flow. C supports three types of control structures - sequential, selection and loop or repetition. The sequential control flow is the default one, just as shown in the program 2.2.1. Let us discuss the selection and loop control structures.

## Key Concepts

if, if-else, switch, for, while, do-while, break, continue

## Discussion

### 2.2.1 Decision making

In our life, there are times when we are under different available choices. Let us say, you go to brush your teeth and you find that you are out of toothpaste.

*You then ask, “do I have any toothpaste?”.*

*If the answer is **no**, then you need to **add it to your shopping list**.*

*However, if the answer is **yes**, you would just **use the toothpaste**.*

This is what happens at a point of selection, answering a question based on what it finds. A selection entails choosing between two or more alternatives. This section discusses the selection control structures supported by C.

### 2.2.2 The simple if statement

It is one of the most strong conditional statements in C. The 'if' statement is in control of changing a program's flow of execution. A condition is often used in the if statement. Before any statement inside the body of 'if' is executed, the condition is evaluated. The following is the syntax for the if statement:

<p>The syntax of the if statement is :</p> <pre>if(condition)     Statement ;</pre> <p>The syntax of if statement is :</p> <pre>if(condition) {     Statement1 ;     Statement2 ;     .     .     .     StatementN ; }</pre>	<p>For single-line statement</p>         <p>For multi-line statements</p>
--	--

Let us say, if we want to show the positive difference between the two numbers 'num1' and 'num2' in the program 2.2.1. To do so, we will have to change our approach.

Examine program 2.2.2 written below, which subtracts the smaller number from the more significant number to ensure that the difference is always positive. The values for the two numbers 'num1' and 'num2', are used to make this decision.

```
/* program to print the positive difference between two numbers*/
#include <stdio.h>
#include <conio.h>
void main()
{
    int num1, num2, diff;
    printf("Enter the first number\t:\t");
    scanf("%d",&num1);
    printf("Enter the second number\t:\t");
    scanf("%d", & num2);
    if(num1 > num2)
        diff = num1 - num2
    if(num2 > num1)
        diff = num2 - num1
    printf("The positive difference between %d and %d is %d",num1, num2,
    diff);
    getch();
}
```

Program 2.2.2

### Output:

```
Enter the first number : 5
Enter the second number : 9
The positive difference between 5 and 9 is 4
```

In the program 2.2.2:

1. We declare two integer variables num1 and num2 and we accept its value from the user.
2. We found the largest number among them and calculated the difference.

Let's consider another program:

```

/* program to check the answer entered by the user is right or not */
#include <stdio.h>
#include <conio.h>
void main()
{
    int key, ans;
    key = 12+24;
    printf("what is the sum of 12 and 24\t:\t");
    scanf("%d", &ans);
    if(ans==key)
        printf("Right answer");
    if (ans!=key)
        printf("wrong answer");
    getch();
}

```

Program 2.2.3

*Could you predict the output of program 2.2.3?*

### 2.2.3 The if-else statement

The if-else statement is a variation of the if statement which allows one to write two alternate paths. The control condition will determine which path should be taken. The if-else sentence has the following syntax.

<p>The syntax of if statement is :</p> <pre> if(condition)     Statement ; else     Statement ; </pre>	<p>For single-line statement</p>
--	----------------------------------

<p>The syntax of if statement is :</p> <pre> if(condition) {     Statement1 ;     Statement2 ;     .     .     .     StatementN ; } else {     Statement1 ;     Statement2 ;     .     .     .     StatementN ; } </pre>	For multi-line statements
--	---------------------------

Let us consider the flowchart to understand program flow in if-else statement

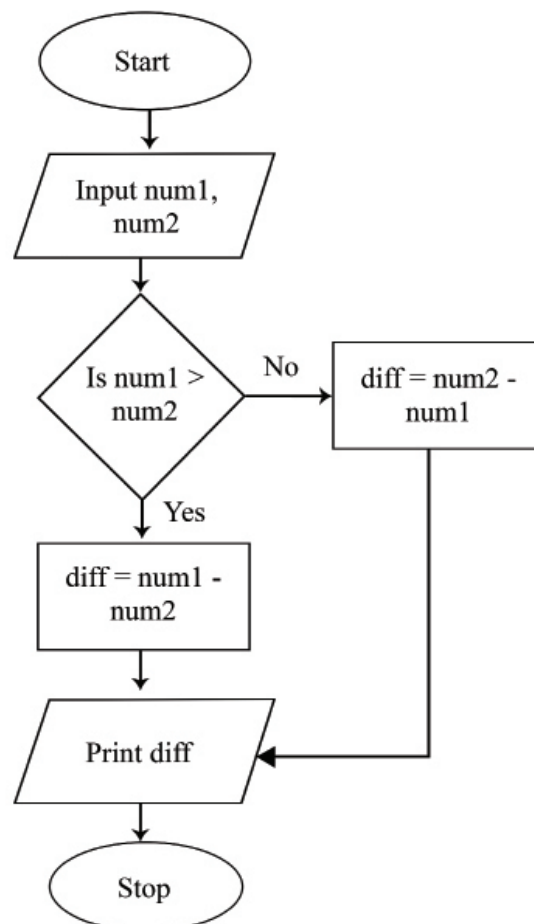


Fig 2.2.1 if-else flowchart

Let us modify program 2.2.3 with an if-else statement in 2.2.4

```
/* program to check the answer entered by the user is right or not */  
#include <stdio.h>  
#include <conio.h>  
void main()  
{  
    int key, ans;  
    key = 12+24;  
    printf("what is the sum of 12 and 24\t:\t");  
    scanf("%d",& ans);  
    if(ans==key)  
        printf("Right answer");  
    else  
        printf("Wrong answer");  
    getch();  
}
```

Program 2.2.4

1. Could you predict the output?
2. How do program 2.2.3 and program 2.2.4 differs

### 2.2.4 Nesting of if-else statement

Nested if-else is used when a series of decisions are required. Using one if-else construct within another is known as nesting. Let us write a program to demonstrate how to use nested if-else statements.

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

```

{
    int num=1;
    if(num<10)
    {
        if(num==1)
        {
            printf("The value is:%d\n",num);
        }
        else
        {
            printf("The value is greater than 1");
        }
    }
    else
    {
        printf("The value is greater than 10");
    }
    getch();
}

```

Program 2.2.5

Output

The value is: 1

The above code uses nested if-else constructs to check if a number is less or greater than 10 and prints the answer:

1. To begin, we have declared a variable num with the value 1. Then we used the if-else statement.
2. The condition given in the outer if-else checks if a number is less than 10. If

the condition is valid, then the inner loop will be executed. The condition is valid in this case, so the inner block is processed.

3. We have another condition in the inner block that checks whether our variable has the value 1 or not. If a condition is valid, the **if** block is executed; otherwise, the **else** block is executed. The condition is valid in this case because the if block is executed, and the value is shown on the output screen.

### 2.2.5 The if-else if ladder

There are many incidences that call for several conditions to be reviewed, leading to a plenty of options. In such instances, we can use **if-else if** to chain the conditions. The syntax of “if-else if” is shown below:

```
if( condition 1)
{
    statements;
}
else if( condition 2)
{
    statements;
}
else if(condition N)
{
    statements;
}
else
{
    statements;
}
```

The number of “else if”, refers to the total number of conditions tested determines conditions. If the first condition is incorrect, the following condition, and so on, are verified. If one of the conditions is valid, the indented block that corresponds to it executes, and the “if statement” ends. If none of the condition holds good, then the last **else** block gets executed.

Let us consider a sample c program (refer Program 2.2.6) to understand the if- else if concept



```

#include<stdio.h>

int main()
{
    int marks=83;
    if(marks>75){
        printf("First class");
    }
    else if(marks>65){
        printf("Second class");
    }
    else if(marks>55){
        printf("Third class");
    }
    else{
        printf("Fourth class");
    }
    return 0;
}

```

Program 2.2.6

Here in this program:

1. We initialize a variable called marks. We've included a variety of criteria in the 'else if' ladder structure.
2. The variable marks' value will be compared to the first condition, and if it matches, the sentence associated with it will be written on the output screen.
3. If the first condition is found to be incorrect, it is compared to the second condition.
4. This process will continue until the entire expression is evaluated, at which control will exit the 'else if' ladder and the default statement will be printed.

## 2.2.6 The switch-case statement

In real life, we are often faced with circumstances in which we must choose from a

variety of options rather than from one or two. For example, deciding which school to attend or which hotel to visit, or even more difficult, deciding which profession to pursue. C programming is the same; the decision we must make is more complex than simply choosing between two options. Rather than using a sequence of if statements, C provides a special control statement that helps us to manage certain cases effectively.

A switch, or more accurately a **switch-case**, is a control statement that helps one to make a decision from a variety of options. The syntax of the switch-case is shown below:

```
switch( expression )
{
    case value-1:
        statement(s);
        break;
    case value-2:
        statement(s);
        break;
    case value-n:
        statement(s);
        break;
    default:
        statement(s);
}
```

#### The rules of switch casing

- ◆ The expression may be either an integer or a character expression.
- ◆ Value-1, 2, and n are case labels that are used to mark each case separately. Remember that case labels should never be the same, as this can cause a problem when running a program.
- ◆ Case labels must end with a colon (:). Each case will associate with a code block

- ◆ In each case, the break keyword indicates the end of a specific case. If we don't put a break in each case, the switch in C will continue to execute all the cases until the end is reached, even though the particular case is executed.
- ◆ The default case is optional. When the value of test-expression does not match any of the cases in the switch, the default is executed.
- ◆ Otherwise, there is no need to use default in the switch.

The control flow that happens in the switch case is shown below

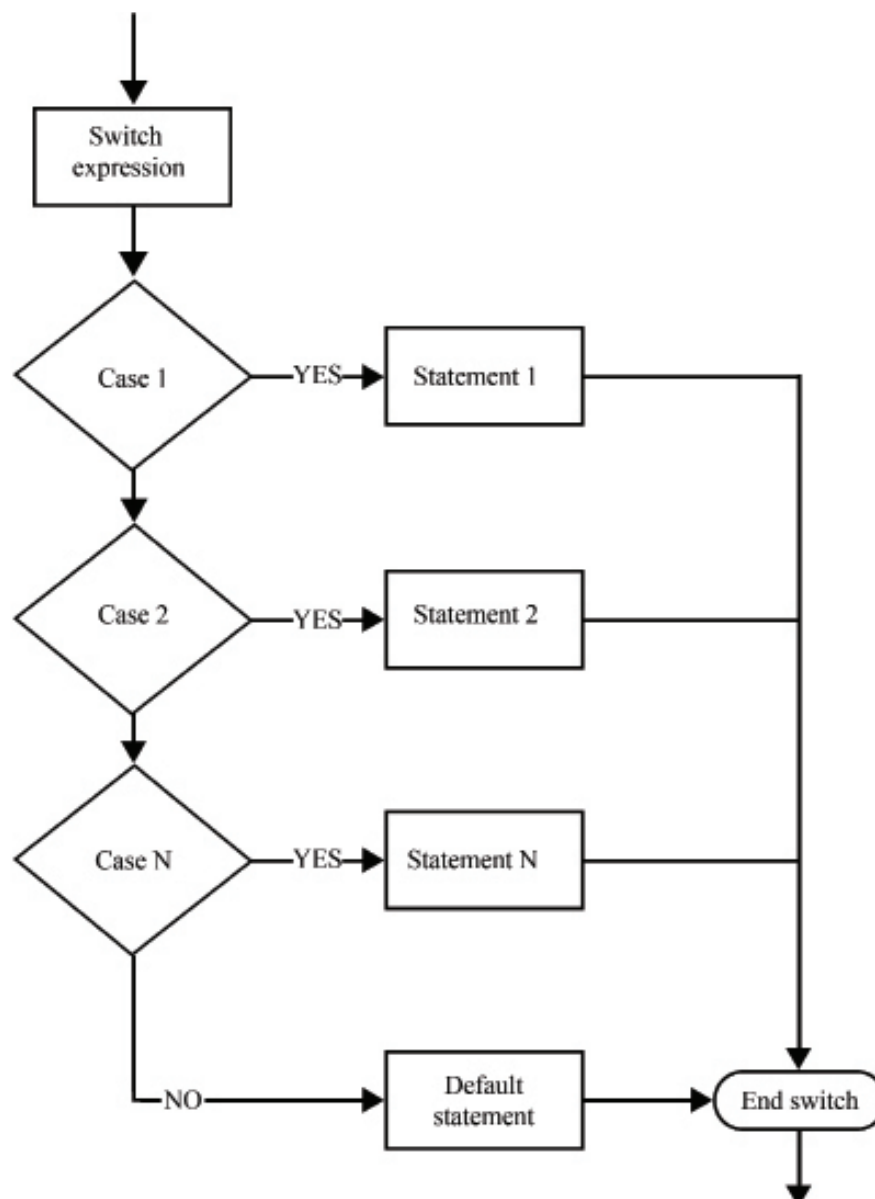


Fig. 2.2.2 Flowchart representation of Switch Statement

Let us understand the switch-case using an example (refer Program 2.2.7)

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

void main() {
    int num = 8;
    switch (num) {
        case 7:
            printf("Value is 7");
            break;
        case 8:
            printf("Value is 8");
            break;
        case 9:
            printf("Value is 9");
            break;
        default:
            printf("Out of range");
    }
    getch();
}
```

Program 2.2.7

### Output

Value is 8

Let us understand the program:

1. In the previous programme, we explained how to initialize a variable num with the value 8.

2. The value stored in the variable **num** is compared using a switch construct, and the block of statements associated with the matched case is executed.
3. Since the value stored in variable num is eight in this code, a switch will run the case with the case label of eight. The control will fall out of the switch after the case is completed, and the program will be terminated with a satisfactory result by printing the value on the output screen.

### 2.2.7 Looping

Sometimes, we may repeat tasks; for example, take the act of hammering a nail. Even though you may not realize it, you are constantly asking yourself, “is the nail in?”. When the answer is “no”, you hammer the nail again. You continue to repeat this question until the answer is “yes”, and then you stop. This kind of repetition is called a loop or iteration. Loops allow a programmer to code efficiently, code repetitive tasks instead of having to write the same actions over and over again.

Let us consider an example program 2.2.8 to understand the loop concept.

```
/* program to print the first five even numbers */  
#include<stdio.h>  
#include<conio.h>  
void main()  
{  
    printf("0\n");  
    printf("2\n");  
    printf("4\n");  
    printf("6\n");  
    printf("8\n");  
}
```

Program 2.2.8

What do we do if we are given 1000 or 100,000 even numbers to print? It would be inefficient to write 100,000 “printf” statements. It would be inconvenient and not the best method for completing the mission.

A better approach is to write a program with a loop or iteration. The algorithm is given below:

1. Take a variable, say, count
2. Initialise the value of count to 0

3. Divide the count by two and Check whether the remainder is 0
4. If yes, print count
5. Increment the value of count (count = count +1)
6. Repeat the step 3 to 5 until the count is less than or equal to 100,000.

Looping control allows you to repeatedly run a series of statements in a program, depending on a condition. A loop's statements are repeated as long as a logical condition remains valid. The value of a variable called the loop's control variable is used to verify this condition. The loop ends when the condition becomes incorrect. The programmer must ensure that this condition becomes incorrect at some stage so that there is an escape condition and the loop does not become infinite. For example, if the condition  $\text{count} \leq 100000$  were not set, the program would crash.

The C program supports three looping controls: *for*, *while* and *do-while*.

### 2.2.8 The “for” loop

The for statement iterates over a set of values or a sequence of values. Each object in the range is passed through the for loop. These values can be numeric, or they can be elements of a data type like a string or an array, as we will see in later chapters.

The control variable tests whether each of the range's values have been traversed or not with each iteration of the loop. The statements inside the loop are not executed until all of the items in the range have been exhausted; the control is then passed to the command immediately following the for loop. When using a for loop, the number of times the loop will run is known ahead of time. Below is a diagram (refer Fig. 2.2.3) of flowchart representing the execution of a for loop.

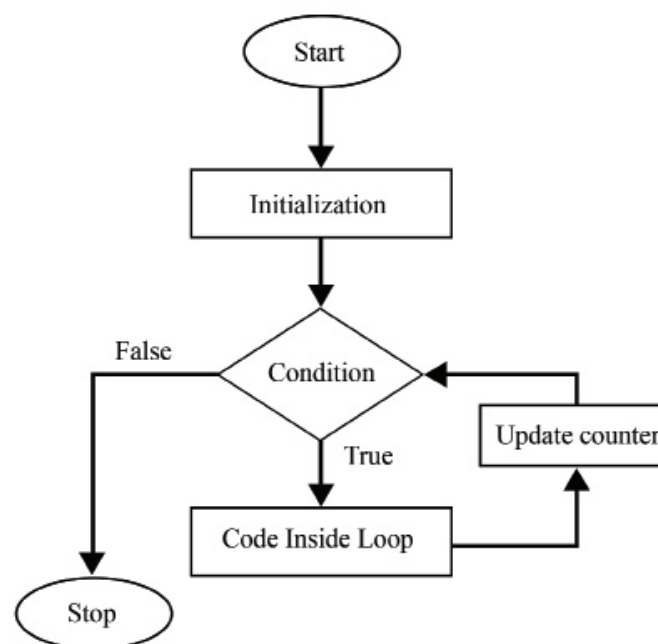


Fig. 2.2.3 Flowchart of a Loop Execution

The syntax of for loop is

```
for (initializationStatement; testExpression; updateStatement)
{
    // statements inside the body of the loop
}
```

- ◆ The initialStatement of the for loop is evaluated only once.
- ◆ The testExpression is a Boolean expression that tests and compares the counter to a fixed value after each iteration, stopping the for loop when false is returned.
- ◆ The updateStatement increases (or decreases) the counter by a set value.

Let us consider a C program (refer 2.2.9) to understand the **for** loop better

	Output
<pre>#include&lt;stdio.h&gt; int main() {     int number;     for(number=1;number&lt;=10;number++) //for loop to print 1-10 nos     {         printf("%d\n",number);        //to print the number     }     return 0; }</pre>	1
	2
	3
	4
	5
	6
	7
	8
	9
	10

Program 2.2.9

The above program prints the numbers from 1 to 10 using a for loop. Let us understand how it works?

To store values, we've declared an int data type variable.

The variable number was given the value 1 in the for loop's initialization section. We stated our condition in the condition section, followed by the increment section.

In for loop initialisation and updation statements are optional. We can initialise before the loop and also can update value inside the loop.

The numbers are printed on a new line by using '\n' in the console using the print feature in the body of the loop. After the first iteration, the value will be incremented, and it will become 2 and the process goes on till the number reaches 10.

### 2.2.9 The “while” loop

In the previous discussion about “for loop”, we saw that the loop executed a fixed number of times. What if we do not know how many times we need to repeat the statements?

For example, suppose you like to listen to music. In that case, every music app has a loop feature that plays your favorite song over and over until you disable the loop function by moving on to the next song.

Similarly, the while loop will execute the code block until the while condition is true. The while loop’s control condition is executed before any of the loop’s statements are executed. The control condition is evaluated for each iteration, and the loop continues as long as the condition is valid. When this condition is incorrect, the statements in the body of the loop are skipped, and control is passed to the statement immediately after the loop’s body. The body is not executed even once if the while loop’s state is initially incorrect.

The while loop’s statements must ensure that the condition becomes false at some point; otherwise, the loop would become infinite, resulting in a logical error in the program (refer Program 2.2.10).

The syntax of while loop is:

```
while(condition)
{
    statement(s);
}
```

Let us consider a C program

<pre>#include&lt;stdio.h&gt;  #include&lt;conio.h&gt;  int main()</pre>	Output
	1
	2



{	3
int num=1; // initializing the variable	4
while(num<=10) //while loop with	5
condition	6
{	7
printf("%d\n",num);	8
num++; //incrementing operation	9
}	10
return 0;	
}	

Program 2.2.10

### 2.2.10 The “do-while” loop

Unlike “for” and “while” loops, which evaluate the loop condition at the top, the do-while loop in C programming tests the loop condition at the bottom.

For example, if a new restaurant opens in your area, you may want to find out whether or not the food is to your liking before you decide to become a regular customer. You will need to visit the restaurant atleast once to try the food, and if it meets your expectations, you can visit frequently.

The do-while loop is identical to the while loop, but it guarantees that the commands will be executed at least once.

The syntax of the do-while loop is:

do
{
statement(s);
} while( condition );

Let us consider a C program

<pre> /* program to print the first n even numbers*/ #include&lt;stdio.h&gt; int main() {     int num=1, limit; //initializing the variable     printf("Enter the limit: ");     scanf("%d",&amp;limit);     do //do-while loop     {         if(num%2==0)         {             printf("%d\n",num);         }         num++; //incrementing operation     }while(num&lt;=limit);     return 0; } </pre>	<p>Enter the limit: 10</p> <p>2</p> <p>4</p> <p>6</p> <p>8</p> <p>10</p>
--	--

Program 2.2.11

In program number 4 we print the first n odd numbers up to the limit entered by the user.

How is the program able to print the series?

1. First we initialize a variable num to 1 and declare a variable limit to get the user input.
2. The scanf() function helps us to get the value from the user and will store it to variable limit. Then we write a do-while loop.
3. In that loop we check whether the remainder is zero when "num" is divided by 2.
4. If so it will print the value of num
5. After each increment, the value of num will increase by 1
6. This will go on until the value of num becomes equal to the value of limit. After that loop will be terminated and a statement which is immediately after the loop will be executed. In this case return 0.

## 2.2.11 Break continue and goto

### Break statement

To get out of a loop quickly, use the **break**. When a break statement is found within a loop or code block, control is immediately removed from the block. The block will terminate. This can also be used in a control structure for switch cases.

Syntax: **break;**

Let us understand the control flow of break command using a flow chart (refer Fig 2.2.4)

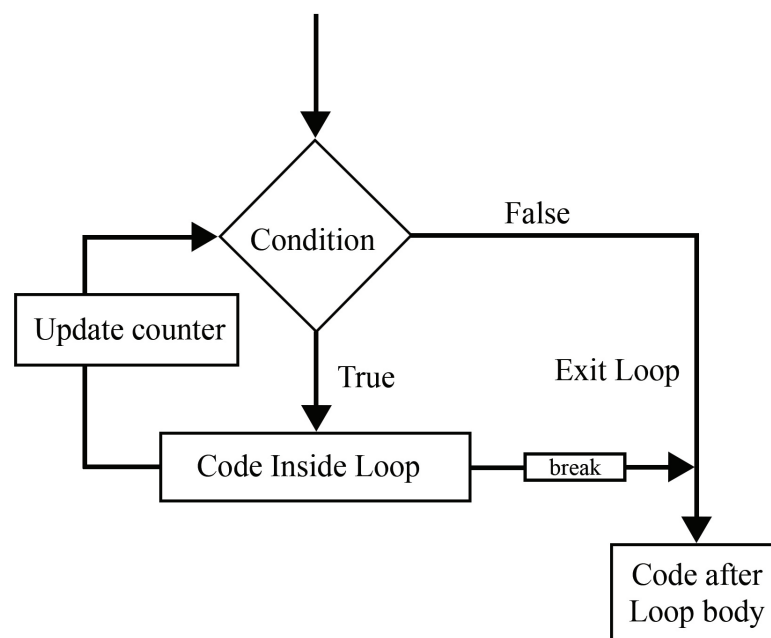


Fig 2.2.4 Flowchart for Break in Loop

Let us consider a sample program to implement the break in while loop (refer Program 2.2.12)

<pre>#include &lt;stdio.h&gt;  int main() {     int num =0</pre>	Output  1  2
--	--------------------------

<pre> while(num&lt;=100) {     printf("value of variable num is: %d\n", num);     if (num &gt;= 3)     {         break;     }     num++; } if(num &lt;=3 )     printf("break command executed"); return 0; } </pre>	<p><b>Output:</b></p> <p>value of variable num is: 0</p> <p>value of variable num is: 1</p> <p>value of variable num is: 2</p> <p>break command executed</p>
---	--

Program 2.2.12

### *Continue statement*

Within loops, the continue expression is used. When a “**continue**” statement appears within a loop, control moves to the beginning of the loop for the next iteration, ignoring the execution of statements inside the loop’s body for the current iteration.

Syntax: <b>continue</b> ;
---------------------------

Let us consider a program (refer 2.2.13) for better clarification

<pre> #include&lt;stdio.h&gt; int main() {     int nb = 7;     while (nb &gt; 0)     {         nb--;         if (nb == 5) </pre>	<p><b>Output</b></p> <p>6</p> <p>4</p> <p>3</p> <p>2</p> <p>1</p> <p>0</p>
--	--

<pre>         continue;         printf("%d\n", nb);     } } </pre>	
--	--

Program 2.2.13

In program 2.2.13, you could notice that the value 5 is skipped because continue is used.

### C - goto command

When a goto statement is encountered in a C program, the control jumps to the label specified in the goto statement.

Syntax of the goto statement is.

<pre> goto label_name;  .. statement(s)  ..  label_name: C-statement(s); </pre>
---

The flowchart of the goto statement is shown below

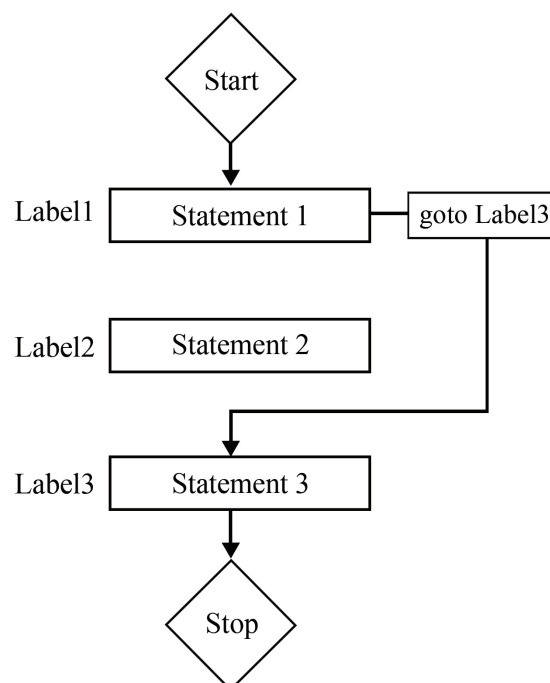


Fig. 2.2.5 Flowchart for goto Statement

Let us understand the goto statement through a c program (refer Program 2.2.14).

<pre>#include &lt;stdio.h&gt; #include &lt;conio.h&gt; void main() {     int sum=0;     for(int i = 0; i&lt;=10; i++){         sum = sum+i;         if(i==5){             goto addition;         }     }     addition:     printf("%d", sum);     getch(); }</pre>	<b>Output:</b>  15
--	--------------------------

Program 2.2.14

Goto is an unconditional jump command that alters the flow of control. It can be used at any point in the program to jump from where it is currently running to another line of code. It will continue to execute the codes sequentially until it has moved to another side. It will not be able to return to previous execution lines. However, it is not advised to use goto statements in a program. The main reasons to avoid the goto command are:

- ◆ It makes the program buggy.
- ◆ Less readable
- ◆ It makes the program logic more complex
- ◆ Debugging will be difficult

The use of goto statements can easily be replaced using break and continue statements.

### 2.2.12 Nested loops

A loop inside a loop is called a nested loop. Let's consider a clock which is a perfect example of a nested loop.

### Include an image of a clock

Inside a clock, once the second's needle completes 60 iterations, only the minute handle move to next. Once the minute handle completes 60 iterations, the hour handle move to next. Here the second's loop will be referred to as the inner loop and the hours loop considered as outer loop.

Let us consider an example (Problem 2.2.15) to understand the concept

<pre>#include &lt;stdio.h&gt;  int main() {     int i, j;     int table = 2;     int max = 5;     for (i = 1; i &lt;= table; i++)     { // outer loop         for (j = 0; j &lt;= max; j++)         { // inner loop             printf("%d x %d = %d\n", i, j, i*j);         }         printf("\n"); /* blank line between tables     */     } }</pre>	<pre>1 x 0 = 0 1 x 1 = 1 1 x 2 = 2 1 x 3 = 3 1 x 4 = 4 1 x 5 = 5  2 x 0 = 0 2 x 1 = 2 2 x 2 = 4 2 x 3 = 6 2 x 4 = 8 2 x 5 = 10</pre>
--	--

Program 2.2.15

The Program 2.2.15 shows nesting of for loops. It can be extended to any level. Similarly you can do nesting of any loops.

## Recap

- ◆ Decision making or control statements are used to choose one of many paths depending on the outcome of the evaluated expression.
- ◆ 'C' uses if, if-else structures for decision-making statements.
- ◆ When several paths must be checked, we can nest if-else statements within one another.
- ◆ When we need to search different options based on the expression's outcome, we use the else-if ladder.
- ◆ A switch case is a control flow construct in C.
- ◆ A switch is used in problems that require you to decide from several choices for decision.
- ◆ A switch must have a test expression that can be executed.
- ◆ A break keyword must be used in each event.
- ◆ Case labels must be consistent and distinct.
- ◆ The default clause is optional.
- ◆ Switch statements may be nested one within the other.
- ◆ In C, a collection of looping statements is executed a number of times until the condition is false.
- ◆ 'C' programming provides us
  - while
  - do-while
  - for loop.
- ◆ goto is an unconditional jump command.
- ◆ goto will not be able to return to previous execution lines
- ◆ break is used to exit from a code block



## Objective Type Questions

1. What is the output of the following c program ?

```
int main()
{
    if( 10 > 9 )
        printf("Singapore\n");
    else if(4%2 == 0)
        printf("England\n");
        printf("Poland");
    return 0;
}
```

2. How many choices are possible while using a single if else statement?
3. What is the output of the following code snippet?

```
int sum = 14;
if ( sum < 20 )
    printf("Under ");
else
    printf("Over ");
    printf("the limit.");
```

4. What is the output of the code fragment shown below ?

```
int sum = 14;
if ( sum < 20 )
    printf("Under ");
else
{
    printf("Over ");
    printf("the limit.");
}
```

5. What is the output of the code block shown below ?

```
int a=3;
switch(a)
{
    case 2: printf("ZERO ");
        break;
    case default: printf("RABBIT ");
}
}
```

6. Predict the output of the code shown below:

```
void main()
{
    int a;
    switch(a)
    {
        printf("Apple ");
    }
    printf("Banana");
}
}
```

7. What is the output of the code block ?

```
int main()
{
    int a;
    switch(a);
    {
        printf("DEER ");
    }
    printf("LION");
}
}
```

8. Guess the output of the code block

```
void main()
{
    static int a=5;
    switch(a)
    {
        case 0: printf("ZERO");break;
        case 5: printf("FIVE");break;
        case 10: printf("DEER");
    }
    printf("LION");
}
```

9. What is the output of the code shown below ?

```
void main()
{
    char code='K';
    switch(code)
    {
        case 'A': printf("ANT ");break;
        case 'K': printf("KING "); break;
        default: printf("NOKING");
    }
    printf("PALACE");
}
```

10. Loop controls supported by C are \_\_\_\_\_
11. \_\_\_\_\_ is the syntax of while loop
12. \_\_\_\_\_ is the output of the following C program

```
int main()
{
    while(true)
    {
        printf("RABBIT");
        break;
    }
    return 0;
}
```

13. Predict the output of the C program

```
int main()
{
    int a=5;
    while(a==5)
    {
        printf("RABBIT");
        break;
    }
    return 0;
}
```

14. What is the output of the C program ?

```
int main()
{
```

```

int a=25;

while(a <= 27)

{

    printf("%d\t", a);

    a++;

}

return 0;

}

```

15. Predict the output of the following program

```

int main()

{

    int i;

    for(i=0;i<10;i++);

    printf("%d", i);

    return 0;

}

```

16. Which command is used to exit immediately from the loop?

17. \_\_\_\_\_ is the output of the C program

```

int main()

{

    int a=0, b=0;

```

```

while(++a < 4)

    printf("%d ", a);

while(b++ < 4)

    printf("%d ", b);

return 0;

}

```

18. Which command is used for coming out of a code block ?
19. In which command is the destination specified using a label ?
20. \_\_\_\_\_ command changes the control flow of code to other statement without any condition

## Answers to Objective Type Questions

1. Singapore  
Poland
2. Two
3. Under the limit
4. Under
5. Compilation error/ syntax error; there is no case default, it is 'default'.
6. Banana (Take note of the absence of any CASE or DEFAULT statements. Even then, the compiler supports it. However, if the CASE statement is not present, nothing will be printed inside of SWITCH.)
7. DEER LION (there is a ';' after switch so printf("DEER ") is outside the switch block.)
8. FIVELION
9. KING PALACE
10. For, while and do-while
11. while(condition){ statement(s);}
12. Compilation error
13. RABBIT

14. 25      26      27

15. 10

16. break

17. 1 2 3 1 2 3 4

18. break

19. goto

20. goto

## Assignments

1. Explain decision making statements in C programming.
2. Write a C program to find the largest among three numbers using nested if.
3. Explain switch statement in C program.
4. Write a C program to perform basic arithmetic operations using switch statements based on a user's choice.
5. Explain entry controlled loop and exit controlled loop.
6. Write a C program to check whether a given string is palindrome or not.
7. Write a C program to check whether a given number is armstrong or not.
8. Write a C program to find the factorial of a given number.
9. Explain break, continue and go statements in C programming.

## Reference

1. Balagurusamy, E. Programming in C. Tata McGraw-Hill Education, 2008.
2. Forouzan, Behrouz A. "Computer Science, A Structured Programming Approach Using C, Brooks." Cole. California, 2001.
3. Kernighan, Brian W., and Dennis M. Ritchie. The C programming language. Prentice Hall, Englewood Cliffs, N.J., 1988.
4. Gottfried, Byron S. Schaum's Outline of Programming in Structured Basic. McGraw-Hill Professional, 1992.



## Arrays and Strings

### Learning Outcomes

After the successful completion of the unit, the learner will be able to:

- ◆ design and use arrays in C program to solve real-world problems
- ◆ know about pointers in C Programming
- ◆ introduce representation of strings in C Programming
- ◆ identify the usage of string with pointers.
- ◆ familiarize the concept of built-in string functions.

### Prerequisites

If you want to go to a place 10km away from your hometown and alone, you will probably choose a two-wheeler as your mode of transport. Nevertheless, if you are going with a group of 20 people, two-wheelers would be a wrong choice. A bus or a van will be more convenient. In the same way, in programming, primitive data types like int or float may be inadequate. In such situations, we use derived data types that can store multiple values at a time. In this section, we discuss arrays. “An array is a collective name given to a group of similar quantities.”

Let's consider an example:

```
1. #include <stdio.h>
2. #include <conio.h>
3. void main()
4. {
5.     int x=7;
6.     x = 12;
7.     printf( “ x = %d \n”, x);
8.     getch();
9. }
```



What will be the output of the above-given program? No doubt, the program will print the value of x as 12. In line 6, when we assign the value 12, the initial value of x will be written i.e., 7. int variables cannot hold more than one value at a time. Here we can use arrays.

As an example, let us say we want to rank 100 students' percentage marks in ascending order. We have two choices for storing these marks in memory :

Create 100 variables to hold the percentage marks earned by 100 different students, with each variable holding one student's marks.

Construct a single variable (also known as an array) that can store or hold all of the hundred values.

Surely, the second choice is preferable. One easy explanation for this is that managing one variable is much simpler than managing 100 different variables. Which means an array is a group of identical elements. These related elements may all be int, float, or chars. A string is a collection of characters, while an array of int or float is simply referred to as an array. Remember that all elements of an array must be of the same type; for example, an array of 10 numbers cannot contain 5 ints and 5 floats. Arrays are composed of contiguous memory locations. The first element has the lowest address, and the last element has the highest.

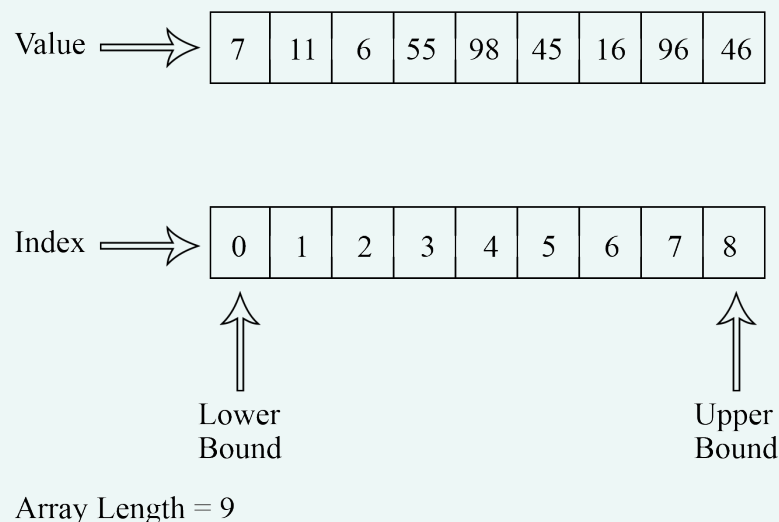


Fig 2.3.1 Array

In a C program, the minor individual units are termed as C tokens. C has six types of tokens. C programs are written using these tokens and the syntax of the language. Six tokens are namely Keywords, Identifiers, Constants, Strings, Operators, and Special symbols.

Every C word is considered as either a keyword or an identifier. All keywords have fixed meanings, and these meanings cannot be changed. Keywords serve as basic building

blocks for program statements. All keywords must be written in lowercase. Examples include auto, else, register, break, enum, return, unsigned, char, float, void, case, etc.

Identifiers guide the names of variables, functions, and arrays. These are user-defined names and comprise a sequence of letters and digits, with a mandate that a letter should be the first character. Both uppercase and lowercase letters are permitted. However, lowercase letters are generally used. The underscore character is also permitted in defining an identifier. Moreover, keywords and white space cannot be used as an identifier.

An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables. C operators can be classified into a number of categories. They include Arithmetic operators, Relational operators, Logical operators, etc.

Constants in C refer to fixed values that do not change during the execution of a program. C mainly supports two types of constants: character constants and numeric constants. A variable in C is a data name that may be used to store a data value. Unlike constants that remain unchanged during the execution of a program, a variable may take different values at different times during execution.

## Key Concepts

One dimensional arrays, Multidimensional arrays, array index

## Discussion

### 2.3.1 Declaration and initialization of arrays

To begin, an array, like other variables, must be declared so that the compiler knows what kind of array we want and how big it should be. This is what we've done in our programme with the following statement:

```
datatype arrayName [ arraySize ];
```

**Example:** `int arr[10];`

In this case, int specifies the variable's type, just as it does with ordinary variables, and the word 'arr' specifies the variable's name. The number 10 indicates how many integer elements will be in our array. This value is commonly referred to as the array's "dimension." The square bracket ( [ ] ) indicates to the compiler that we are working with an array.

Let us see how individual elements in an array can be referred to after it has been

declared. Subscript; the number after the array name in brackets, is used to accomplish this. The position of the element in the array is determined by this number. The array elements are numbered from zero to n-1.

Let us consider a C program (refer to Program 2.3.1) to understand it

```
/* program to find the average of given five numbers*/
#include<stdio.h>
void main()
{
    int i, avg, num[5] = { 10,20,32,50,26},sum=0;
    for ( i = 0 ; i <5 ; i++ )
        sum = sum + num[ i ] ; /* read data from an array by using index
value          or position value */
    avg = sum / 5 ;
    printf ( "Average marks = %d\n", avg ) ;
}
```

Program 2.3.1

We use the variable i as a subscript to refer to different elements of the array in the above code. This attribute may have several values, allowing it to apply to the array's various elements in turn. The ability to interpret subscripts with variables is what makes arrays so useful.

Let us see how to enter elements into arrays

```
#include<stdio.h>
voidmain()
{
    int i, num[5];
    printf ( "Enter any 5 numbers " ) ;
    for ( i = 0 ; i <5 ; i++ )
    {
        scanf ( "%d", &num[ i ] ) ;
    }
}
```

In the above code snippet, the **for** loop repeats the process of asking the user to input any number, five times. Since *i* has a value of 0 the first time around the loop, the `scanf( )` function would cause the value typed to be stored in the array element `num[ 0 ]`, the array's first element. This process will be repeated before *i* attain the value of 5.

We used the “address of ” operator (&) on the array element `num[i]`, just like we did on other variables, in the `scanf( )` function (&*rate*, for example). We are passing the address of this particular array element to the `scanf( )` function, rather than its value, as is required by `scanf( )`.

The **for** loop in the following code snippet is similar, except that the body of the loop now adds each number to a running total stored in a variable called *sum*. After adding up all of the numbers, divide the total by 5, the number of students, to get the average.

```
for ( i = 0 ; i < 5 ; i++ )  
  
sum = sum + num[ i ] ; /* read data from an array*/  
  
avg = sum / 5 ;  
  
printf ( “Average marks = %d\n”, avg ) ;
```

### 2.3.2 Array elements in memory

Take a look at the array declaration below:

```
int array[5];
```

When we make this declaration, what happens in the computer memory? 10 bytes are reserved in memory right away, 2 bytes for each of the five integers. Since the array isn't initialised, all of the values in it are garbage.

Regardless of the initial values, all array elements will still be present in contiguous memory locations. This memory array element arrangement is depicted below.

Include image of array with 5 elements and contiguous memory location

Note: The programmer must ensure that the you can't exceed the size of the array

### 2.3.3 Multi-dimensional arrays

In the previous section, we looked at arrays with only one dimension. Arrays of two or more dimensions are also possible. This session explains how to build and manipulate multidimensional arrays in C. The following is an example of a multidimensional array declaration in its most basic form.

```
Datatype arrayName[size1][size2]...[sizeN];
```

The two-dimensional array is the most basic type of multidimensional array. A two-dimensional array is essentially a list of one-dimensional arrays. The following is the syntax for a two-dimensional array:

`Datatype arrayName[size1][size2];`

Let's consider an example Program 2.3.2 to understand initialization and accessing the two dimensional array elements

```
#include <stdio.h>
#include <conio.h>
int main () {
    /* an array with 5 rows and 2 columns*/
    int a[5][2];
    int i, j;
    for (i=0;i<5;i++)
    {
        for (j=0;j<2;j++)
        {
            printf("Enter a[%d][%d]: ",i,j);
            scanf("%d",&a[i][j]);
        }
    }
    /* output each array element's value */
    for ( i = 0; i < 5; i++ ) {

        for ( j = 0; j < 2; j++ ) {
            printf("a[%d][%d] = %d\n", i,j, a[i][j] );
        }
    }
    getch();
    return 0;
}
```

Program 2.3.2

The above program has two parts. We read-in the values in the first part using a for loop, and we print them out in the second part using another for loop. The array elements were stored row-by-row and accessed row-by-row in our example program. You can also view the array elements by column. Since array elements are traditionally stored and accessed row-by-row, we will use the same approach.

The array elements are contained in one continuous chain in memory, whether it's a one-dimensional or two-dimensional array.

### 2.3.4 What are Strings?

The C language provides a capability that enables the user to design a set of similar data types, called arrays. In the same way a group of integers can be stored in an integer array, a group of characters can be stored in a character array. Character arrays are also called strings. Character arrays or strings are used by programming languages to manipulate text, such as words and sentences.

A string constant is a one-dimensional array of characters terminated by a null ( '\0' ). For example,

```
char name[ ] = { 'H', 'A', 'E', 'S', 'L', 'E', 'R', '\0' } ;
```

Each character in the array occupies 1 byte of memory and the last character is always '\0'. What is this character? '\0' is called a null character. However '\0' and '0' are not the same. The ASCII value of '\0' is 0, whereas the ASCII value of '0' is 48. Figure 2.3.2 below shows the way a character array is stored in memory. Note that the elements of the character array are stored in contiguous memory locations. The terminating null ('\0') is important because it is the only way the functions that work with a string can know where the string ends. In fact, a string not terminated by a '\0' is not a string, but it is just a collection of characters.



Fig 2.3.2 String as an Array

However, we would often use strings and hence C programming provides a shortcut for initializing strings. For example, the above example of string termed “HAESLER” could be initialized as,

```
char name[ ] = “HAESLER”;
```

In this method of declaration '\0' is not necessary. C automatically inserts the null character.

### 2.3.5 More about Strings

In what way are character arrays different from numeric arrays? In C, elements in a

character array be accessed in the same way as the elements of a numeric array? Do we need to take any special care of '\0'? Why do numeric arrays don't end with a '\0'? Declaring strings is okay, but how do we manipulate them? Let us find answers to the above questions with the help of a few sample programs.

### Manipulating the String or Character Arrays using :

#### 2.3.5.1 Word count

```
/* Program to demonstrate printing of a string */  
  
#include <stdio.h>  
  
int main( )  
{  
    char name[ ] = "Hellions" ;  
    int i = 0 ;  
    while ( i <= 7 )  
    {  
        printf ( "%c", name[ i ] ) ;  
        i++ ;  
    }  
    printf ( "\n" ) ;  
    return 0 ;  
}
```

#### Output:

Hellions

Program 2.3.3

Here in the above program (refer to Program 2.3.3), a character array is initialized, and then printed out the elements of this array iterating through a while loop. Can we limit the looping iteration without using the hardbound count of 7?

#### 2.3.5.2 End Delimiter

It is possible as every string or character array always ends with a '\0'. The Program 2.3.4 illustrates this conditional check:

```

/* Program to demonstrate printing of a string */
#include <stdio.h>
int main( )
{
    char name[ ] = "Hellions" ;
    int i = 0 ;
    while (name[i] != '\0')
    {
        printf ( "%c", name[ i ] ) ;
        i++ ;
    }
    printf ( "\n" ) ;
    return 0 ;
}

```

**Output:**

Hellions

Program 2.3.4

The above-given program doesn't rely on the string length (number of characters in the string) to print out its contents. Hence it provides a more general solution than the first example.

The printf( ) function has got a simple way of printing strings as shown below. Also printf( ) doesn't print the '\0'.

```

#include <stdio.h>
int main( )
{
    char name[ ] = "Klinsman" ;
    printf ( "%s", name ) ;
}

```

**Output:**

Klinsman

Program 2.3.5



The `%s` used in the above `printf( )` statement (Program 2.3.5) is a format specification for printing out a string. The same specification can be used to receive a string from the keyboard, as shown below.

```
#include <stdio.h>

int main( )
{
    char name[ 25 ] ;
    printf ( "Enter your name " ) ;
    scanf ( "%s", name ) ;
    printf ( "Hello %s!\n", name ) ;
    return 0 ;
}
```

**Output:**

```
Enter your name Debashish
Hello Debashish!
```

Program 2.3.6

The declaration in the example Program 2.3.6, `char name[ 25 ]` sets aside 25 bytes under the array `name[ ]`, whereas the `scanf( )` function fills in the characters typed at keyboard into this array until the Enter key is hit. Once enter is hit, `scanf( )` places a `'\0'` in the array. Naturally, we should pass the base address of the array to the `scanf( )` function.

While entering the string using `scanf( )`, we must be cautious about two things:

(a) The length of the string should not exceed the dimension of the character array. This is because the C compiler doesn't perform bounds checking on character arrays. Hence, if you carelessly exceed the bounds, there is always a danger of overwriting something important, and in that event, you would have nobody to blame but yourselves.

(b) `scanf( )` is not capable of receiving multi-word strings. Therefore, names such as 'Debashish Roy' would be unacceptable. The way to get around this limitation is by using the function `gets( )`. The usage of functions `gets( )` and its counterpart `puts( )` is shown below.

```

#include <stdio.h>

int main( )
{
    char name[ 25 ] ;
    printf ( "Enter your full name: " ) ;
    gets ( name ) ;
    puts ( "Hello!" ) ;
    puts ( name ) ;
    return 0 ;
}

```

**Output:**

Enter your full name: Debashish Roy

Hello!

Debashish Roy

Program 2.3.7

In the above Program 2.3.7 puts( ) displays only one string at a time, hence two puts ( ) are used in the above program. Also, on displaying a string, puts( ) places the cursor on the next line. However, gets( ) is capable of receiving only one string at a time; the advantage of gets( ) is that it can receive a multi-word string. scanf( ) can be used to accept multi-word strings by writing it as given below.

```

char name[ 25 ] ;

printf ( "Enter your full name " ) ;

scanf ( "%[ ^\n ]s", name ) ;

```

Here, [^\n] indicates that scanf( ) will keep receiving characters into name[ ] until \n is encountered.

### 2.3.6 String Functions

The string functions in C language are built-in functions, which can be used for various string manipulation operations.

Function	Use
strlen	Finds length of a string
strlwr	Converts a string to lowercase
strupr	Converts a string to uppercase
strcat	Appends one string at the end of another
strncat	Appends first n characters of a string at the end of another
strcpy	Copies a string into another
strncpy	Copies first n characters of one string into another
strcmp	Compares two strings
strncmp	Compares first n characters of two strings
strcmpi	Compares two strings by ignoring the case
stricmp	Compares two strings without regard to case (identical to strcmpi)
strnicmp	Compares first n characters of two strings without regard to case
strdup	Duplicates a string
strchr	Finds first occurrence of a given character in a string
strrchr	Finds last occurrence of a given character in a string
strstr	Finds first occurrence of a given string in another string
strset	Sets all characters of string to a given character
strnset	Sets first n characters of a string to a given character
strrev	Reverses string

Fig 2.3.3 String Functions

The figure 2.3.3 above shows some commonly used functions along with their purpose. From the above list given in Fig 2.3.3. more commonly used functions like `strlen()`, `strcpy()`, `strcat()` and `strcmp()` will be illustrated below and shows how the library functions in general handle strings. Let us explore these functions one-by-one.

### 2.3.6.1 `strlen()`

This function counts the number of characters present in a string. Its usage is illustrated in the following program (refer to Program 2.3.8):

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

int main( ) {
    char arr[ ] = "Bamboozled" ;
    int len1, len2 ;
    len1 = strlen ( arr ) ;
    len2 = strlen ( "Humpty Dumpty" ) ;
    printf ( "string = %s length = %d\n", arr, len1 ) ;
    printf ( "string = %s length = %d\n", "Humpty Dumpty", len2
) ;

    return 0 ;
}
```

#### **Output:**

string = Bamboozled length = 10

string = Humpty Dumpty length = 13

Program 2.3.8

You may notice that in the first call to the function `strlen()`, we are passing the base address of the string, and the function returns the length of the string. While calculating the length, it doesn't count `'\0'`. In the second call as in the above example,

```
len2 = strlen ( "Humpty Dumpty" ) ;
```

Input to the `strlen()` function is the address of the string and not the exact string itself.

### 2.3.6.2 `strcpy()`

This function copies the contents of one string into another. The base addresses of the

source and target strings should be supplied to this function. An example of strcpy( ) is given in the below program (refer to Program 2.3.9)

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

int main( )
{
    char source[ ] = "Sayonara" ;
    char target[ 20 ] ;
    strcpy ( target, source ) ;
    printf ( "source string = %s\n", source ) ;
    printf ( "target string = %s\n", target ) ;
    return 0 ;
}
```

And here is the output...

source string = Sayonara

target string = Sayonara

Program 2.3.9

On supplying the base addresses, strcpy( ) goes on copying the characters in source string into the target string till it encounters the end of source string ('\0'). It is our responsibility to make sure that the target string's dimension is big enough to hold the string being copied into it. Thus, a string gets copied into another, piece-meal, character-by character.

### 2.3.6.3 strcmp()

This function is used to compare two strings. It stands for "string compare". The function takes two string arguments and compares them character by character until it finds a difference or reaches the end of one of the strings.

**Syntax: strcmp(str1,str2)**

The strcmp() function compares string1 with string 2 and returns:

0 if both strings are equal.

A negative value if string1 is lexicographically less than string2.

A positive value if string1 is lexicographically greater than string2

```
#include <stdio.h>

#include <string.h>

int main()
{
    char str1[] = "rice";
    char str2[] = "wheat";
    int result = strcmp(str1, str2);
    if (result == 0)
    {
        printf("Strings are equal.\n");
    }
    else if (result < 0)
    {
        printf("str1 is less than str2.\n");
    }
    else
    {
        printf("str1 is greater than str2.\n");
    }
    return 0;
}
```

### Output

str1 is less than str2

In the above example, since "rice" comes before "wheat" in lexicographic order, the output would be "str1 is less than str2."

### 2.3.6.4 strcat()

This function is used to concatenate (i.e., append) one string to the end of another string. It stands for “string concatenation”.

#### Syntax: strcat(str1,str2)

This function appends a copy of the null-terminated string str2 to the end of the null-terminated string str1. The str1 argument should be large enough to hold the concatenated result. The str2 argument remains unchanged.

```
#include <stdio.h>

#include <string.h>

int main()
{
    char str1[50] = "Good ";
    const char str2[] = "Morning";
    strcat(str1, str2);

    printf("Concatenated string: %s\n", str1);

    return 0;
}
```

In this example, str1 is "Good" and str2 is "Morning". After calling strcat(str1,str2), the contents of str1 become “Good Morning ”.

## Recap

- ◆ An array is a collection of same data type stored in contiguous memory locations.
- ◆ The array’s first element is numbered 0, so the last element is one less than the array’s length.
- ◆ Each element in the array is accessed by its index.
- ◆ An array’s form and dimension must be declared before it can be used.
- ◆ An array’s bounds are not verified by the compiler.
- ◆ Since array elements are stored in contiguous memory locations, pointers can be used to access them.
- ◆ It is possible to construct multidimensional arrays.

- ◆ A 2-D array is a collection of several 1-D arrays.
- ◆ About the concept of strings
- ◆ Manipulating strings
- ◆ String functions

## Objective Type Questions

1. Array is an example of \_\_\_\_\_ type memory allocation.
2. A one dimensional array A has indices 0....75. Each element is a string and takes up three memory words. The array is stored at location 1120 decimal. The address of A[49] is ?
3. What will be the address of the arr[2][3] if arr is a 2-D long array of 4 rows and 5 columns and the starting address of the array is 2000?
4. What is the maximum number of dimensions an array in C can have?
5. Which standard string function will you use to join two words?
6. Which standard string function is used for the Copying function?
7. The \_\_\_\_\_ function appends not more than n characters.
8. What will strcmp() function do?
9. What is a String in C Language?
10. What is the Format specifier used to print a String or Character array in C?

## Answers to Objective Type Questions

1. Compile time
2.  $1120 + (49-1) \times 3 = 1264$
3. 2052
4. Theoretically there are no limits. It purely depends on system memory and compiler
5. strcat()
6. strcpy and memcpy
7. strncat()
8. compare two strings and return an integer value that indicate the relationship between the 2 strings being compared



9. String is an array of Characters with null character as the last element of array
10. %s

## Assignments

1. Write a simple program to search an element in 1D array.
2. How to make the search easy imagine that the array elements are sorted in order.
3. What is an array? Explain its significance in representation of data.
4. What are the different ways to declare and initialise an array?
5. Explain a multi dimensional array.
6. How to represent strings in the C programming language.
7. Write a C program to declare an array of 10 elements and find the largest element in an array.

## Reference

1. Balagurusamy, E. Programming in C. Tata McGraw-Hill Education, 2008.
2. Forouzan, Behrouz A. "Computer Science, A Structured Programming Approach Using C, Brooks." Cole. California, 2001.
3. Kernighan, Brian W., and Dennis M. Ritchie. The C programming language. Prentice Hall, Englewood Cliffs, N.J., 1988.
4. Gottfried, Byron S. Schaum's Outline of Programming in Structured Basic. McGraw-Hill Professional, 1992.



## Pointers and Dynamic Memory Allocation

### Learning Outcomes

After the successful completion of the unit, the learner will be able to:

- ◆ understand the need for storing memory addresses
- ◆ recall accessing a variable through pointers
- ◆ familiarize with malloc(), calloc(), free() and realloc() functions

### Prerequisites

In memory, every stored data item occupies one or more contiguous memory cells. The number of memory cells required to store a data item depends on its type (char, int, double, etc.). Whenever we declare a variable, the system allocates memory location(s) to hold the value of the variable. Since every byte in memory has a unique address, this location will also have its own (unique) address.

Consider the statement

```
int x = 50;
```

This statement instructs the compiler to allocate a location for the integer variable x, and put the value 50 in that location. Suppose that the address location allocated is 100. During the execution of the program, the system always associates the name x with the address 100. The value 50 can be accessed by using either the name x or the address 100. Since memory addresses are simply numbers, they can be assigned to some variables which can be stored in memory. Such variables that hold memory addresses are called pointers. Pointers enable us to access the values of a variable using their memory address.

### Key Concepts

Pointers, dynamic memory allocation, malloc, calloc, realloc, free

## Discussion

### 2.4.1 Pointers in C

A pointer is a variable whose value is the address of another variable, i.e., the direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is

**datatype \*ptr**

An example pointer “p” that holds the address of an integer variable or holds the address of a memory whose value can be accessed as integer values through “p”

```
int *p;
```

#### 2.4.1.1 Working of pointers

Consider the declaration:

```
int i=3;
```

This declaration tells the C compiler to

- ◆ Reserve space in memory to hold the integer value.
- ◆ Associate the name i with this memory location.
- ◆ Store the value 3 at this location.

We may represent i’s location in memory by the memory map shown in Figure 2.4.1

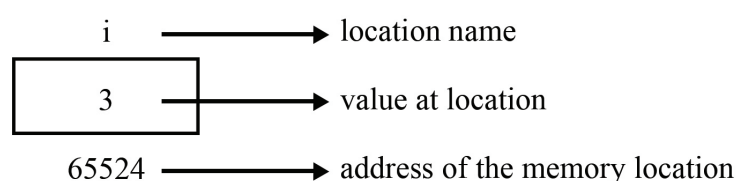


Fig 2.4.1: Memory map for the declaration `int i=3`

We see that the computer has selected memory location 65524 as the place to store the value 3. The memory address 65524 is not a number to be relied upon, because at some other time the computer may choose a different location for storing the value 3. The important point is, i’s address in memory is a number. We can print this address through the following program:

```
#include <stdio.h>

int main( )
{
    int i = 3 ;

    printf ( "Address of i = %u\n", &i ) ;

    printf ( "Value of i = %d\n", i ) ;

    return 0 ;
}
```

The output of the above program would be:

Address of i = 65524

Value of i = 3

Look at the first printf( ) statement carefully. ‘&’ used in this statement is C’s ‘address of’ operator. The expression &i returns the address of the variable i, which in this case happens to be 65524. Since 65524 represents an address, there is no question of a sign being associated with it. Hence it is printed out using %u, which is a format specifier for printing an unsigned integer. We have been using the ‘&’ operator all the time in the scanf( ) statement.

The other pointer operator available in C is ‘\*’, called the ‘value at address’ operator. It gives the value stored at a particular address. The ‘value at address’ operator is also called ‘indirection’ operator.

Consider the following program

```
#include <stdio.h>

int main( )
{
    int i = 3 ;

    printf ( "Address of i = %u\n", &i ) ;

    printf ( "Value of i = %d\n", i ) ;

    printf ( "Value of i = %d\n", *( &i ) ) ;
}
```

```
return 0 ;  
}
```

The output of the above program would be:

Address of i = 65524

Value of i = 3

Value of i = 3

Note that printing the value of `*( &i )` is the same as printing the value of `i`. The expression `&i` gives the address of the variable `i`. This address can be collected in a variable, by saying,

```
j = &i ;
```

But remember that `j` is not an ordinary variable like any other integer variable. It is a variable that contains the address of another variable (`i` in this case). Since `j` is a variable, the compiler must provide it with space in the memory. Once again, the memory map shown in Figure 2.4.2 would illustrate the contents of `i` and `j`.



Fig 2.4.2: Memory locations and contents of the variables `i` and `j`

As you can see, `i`'s value is 3 and `j`'s value is `i`'s address. But wait, we can't use `j` in a program without declaring it. And since `j` is a variable that contains the address of `i`, it is declared as,

```
int *j ;
```

This declaration tells the compiler that `j` will be used to store the address of an integer value. In other words, `j` points to an integer. How do we justify the usage of `*` in the declaration,

```
int *j ;
```

Let us go by the meaning of `*`. It stands for 'value at address'. Thus, `int *j` would mean, the value at the address contained in `j` is an `int`.

Here is a program that demonstrates the relationships we have been discussing.

```

#include <stdio.h>

int main( )
{
    int i = 3 ;

    int *j ;

    j = &i ;

    printf ( "Address of i = %u\n", &i ) ;
    printf ( "Address of i = %u\n", j ) ;
    printf ( "Address of j = %u\n", &j ) ;
    printf ( "Value of j = %u\n", j ) ;
    printf ( "Value of i = %d\n", i ) ;
    printf ( "Value of i = %d\n", *( &i ) ) ;
    printf ( "Value of i = %d\n", *j ) ;

    return 0 ;
}

```

The output of the above program would be:

Address of i = 65524

Address of i = 65524

Address of j = 65522

Value of j = 65524

Value of i = 3

Value of i = 3

Value of i = 3

Consider the following declarations.

int \*alpha ;

char \*ch ;

float \*s ;



Here, alpha, ch and s are declared as pointer variables, i.e., variables capable of holding addresses. The declaration float \*s does not mean that s is going to contain a floating-point value. What it means is, s is going to contain the address of a floating-point value. Similarly, char \*ch means that ch is going to contain the address of a char value. Or in other words, the value at the address stored in ch is going to be a char.

Pointer, is a variable that contains the address of another variable. This variable itself might be another pointer. Thus, we now have a pointer that contains another pointer's address. The following example should make this point clear:

```
#include <stdio.h>

int main( )
{
    int i = 3, *j, **k ;
    j = &i ;
    k = &j ;

    printf ( "Address of i = %u\n", &i ) ;
    printf ( "Address of i = %u\n ", j ) ;
    printf ( "Address of i = %u\n ", *k ) ;
    printf ( "Address of j = %u\n ", &j ) ;
    printf ( "Address of j = %u\n ", k ) ;
    printf ( "Address of k = %u\n ", &k ) ;
    printf ( "Value of j = %u\n ", j ) ;
    printf ( "Value of k = %u\n ", k ) ;
    printf ( "Value of i = %d\n ", i ) ;
    printf ( "Value of i = %d\n ", * ( &i ) ) ;
    printf ( "Value of i = %d\n ", *j ) ;
    printf ( "Value of i = %d\n ", **k ) ;

    return 0 ;
}
```

Consider the memory locations for variables i, j, and k in the above program as shown in Figure 2.4.3.

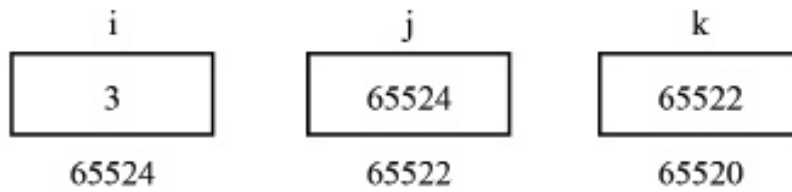


Fig 2.4.3: Pointer to pointer example

The output of the above program would be:

Address of i = 65524

Address of i = 65524

Address of i = 65524

Address of j = 65522

Address of j = 65522

Address of k = 65520

Value of j = 65524

Value of k = 65522

Value of i = 3

Value of i = 3

Value of i = 3

Value of i = 3

In the above program i, j and k are declared as

```
int i, *j, **k ;
```

Here, i is an ordinary int, j is a pointer to an int (often called an integer pointer), whereas k is a pointer to an integer pointer.

## 2.4.2 Dynamic Memory Allocation

Since C is a structured language, it has some fixed rules for programming. One of them includes changing the size of an array. An array is a collection of items stored at contiguous memory locations.

Consider the array in Figure 2.4.4. The size of the array is 9. But what if there is a requirement to change this length (size). If there is a situation where only 5 elements are needed to be entered in this array. The remaining 4 indices will be just wasting memory in this array. So there is a requirement to lessen the length (size) of the array from 9 to 5. Take another situation. In this, there is an array of 9 elements with all 9 indices filled.



But there is a need to enter 3 more elements in this array. In this case, 3 more indices are required. So the length (size) of the array needs to be changed from 9 to 12. This procedure is referred to as Dynamic Memory Allocation in C.

Dynamic Memory Allocation can be defined as a procedure in which the size of a data structure (like an Array) is changed during the runtime. C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under `<stdlib.h>` header file to facilitate dynamic memory allocation in C programming. They are:

- ◆ `malloc()`
- ◆ `calloc()`
- ◆ `free()`
- ◆ `realloc()`

#### 2.4.2.1 malloc()

The “malloc” or “memory allocation” method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It doesn't initialize memory at execution time so will have the default garbage value initially.

##### Syntax:

**`ptr = (cast-type*) malloc(byte-size)`**

For Example:

```
ptr = (int*) malloc(100 * sizeof(int));
```

Since the size of int is 4 bytes, the above statement will allocate 400 bytes of memory. The pointer ptr holds the address of the first byte in the allocated memory. If space is insufficient, allocation fails and returns a NULL pointer.

#### 2.4.2.2 calloc()

“calloc” or “contiguous allocation” method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. It is very much similar to malloc() but it is different in two points and they are:

It initializes each block with a default value '0'.

It has two parameters or arguments as compared to malloc().

##### Syntax:

**`ptr = (cast-type*) calloc(n, element-size);`**

here, n is the no. of elements and element-size is the size of each element.

For Example:

```
ptr = (float*) calloc(25, sizeof(float));
```

This statement allocates contiguous space in memory for 25 elements each with the size of the float. If space is insufficient, allocation fails and returns a NULL pointer.

#### 2.4.2.3 free()

“free” method in C is used to dynamically de-allocate the memory. The memory allocated using functions malloc() and calloc() is not de-allocated on its own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce the wastage of memory by freeing it.

##### Syntax:

```
free(ptr);
```

#### 2.4.2.4 realloc()

“realloc” or “re-allocation” method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to dynamically re-allocate memory. re-allocation of memory maintains the already present value and new blocks will be initialized with the default garbage value.

##### Syntax:

```
ptr = realloc(ptr, newSize);
```

where ptr is reallocated with new size ‘newSize’.

### 2.4.3 Manipulating Strings using Pointers

A string can be declared using a pointer similar to a character array. We can also use pointer variable instead of an array variable. The string may be initialised when it is declared, the string will be stored in memory and string pointer will be assigned with the address of the first character of the string. The below program uses a pointer to access the array elements.

```
#include <stdio.h>

int main( )
{
    char name[ ] = "Jennath" ;

    char *ptr ;

    ptr = name ; /* store base address of string */
}
```

```

while ( *ptr != '\0' )
{
    printf ( "%c", *ptr ) ;

    ptr++ ;
}

printf ( "\n" ) ;

return 0 ;
}

```

Program 2.3.5

We get the base address (address of the zeroth element) of the array by mentioning the name of the array. This base address is stored in the variable ptr using,

```
ptr = name ;
```

Once the base address is obtained in ptr, \*ptr would yield the value at this address, which gets printed promptly through,

```
printf ( "%c", *ptr ) ;
```

Then, ptr is incremented to point to the next character in the string. This derives from two facts: array elements are stored in contiguous memory locations and on incrementing a pointer, it points to the immediately next location of its type. This process is carried out until ptr points to the last character in the string, that is, '\0'.

In fact, the character array elements can be accessed exactly in the same way as the elements of an integer array. Thus, all the following notations refer to the same element:

- ◆ name[ i ]- Accesses the character at index i directly
- ◆ \*( name + i )- Accesses the character at index i using pointer Arithmetic
- ◆ \*( i + name )- Equivalent to \*(name + i)
- ◆ i[ name ]- Equivalent to name[i]- accessing characters using array indexing syntax

There are a lot of ways as shown above to refer to the elements of a character array.

## 2.4.4 More on character pointer

Suppose we wish to store "Hello". We could either store it in a string or we may ask the C compiler to store it at some memory location and assign the address of the string in a

char pointer. This is shown below.

```
char str[ ] = "Hello" ; // Storing in string (character array)
```

```
char *p = "Hello" ; // charcater pointer
```

There is a huge difference in the usage of these two forms of character arrays. For a defined String, we cannot assign another string (see the example below) whereas, for a char pointer we can assign another char pointer. See the example program 2.3.9 for the above mentioned scenario:

```
int main( )
{
    char str1[ ] = "Hello" ;
    char str2[ 10 ] ;
    char *s = "Good Morning" ;
    char *q ;
    str2 = str1 ; /* error */
    q = s ; /* works */
    return 0 ;
}
```

Program 2.3.9

Also, once a string has been defined, another set of characters cannot be assigned to it. Unlike strings, such an operation is perfectly valid with char pointers.

```
int main( )
{
    char str1[ ] = "Hello" ;
    char *p = "Hello" ;
    str1 = "Bye" ; /* error */
    p = "Bye" ; /* works */
}
```

Program 2.3.10

## Custom String Function Implementation

The code snippet below (refer to Program 2.3.11) initiates(custom program) the standard string library function strlen( ).

```
/* A look-alike of the function strlen( ) */

#include <stdio.h>

int xstrlen ( char *str) ;

int main( )
{
    char arr[ ] = "Bamboozled" ;

    int len1, len2 ;

    len1 = xstrlen ( arr ) ;

    len2 = xstrlen ( "Humpty Dumpty" ) ;

    printf ( "string = %s, length is %d\n", arr, len1 ) ;

    printf ( "string = %s, length is %d\n", "Humpty Dumpty", len2 ) ;

    return 0 ;

}

int xstrlen ( char *s )
{
    int length = 0 ;

    while ( *s != '\0' )
    {
        length++ ;

        s++ ;
    }

    return ( length ) ;

}
```

Output:

string = Bamboozled, length is 10

string = Humpty Dumpty, length is 13

#### Program 2.3.11

The function `xstrlen()` is fairly straightforward and simple. What it does primarily is to keep counting the characters till the end of the string. Or it keeps counting characters till the pointer points to the character `'\0'`.

Let us now attempt to mimic `strcpy()`, via our own string copy function, which we will call `xstrcpy()`.

```
#include <stdio.h>

void xstrcpy ( char *, char * ) ;

int main( )
{
    char source[ ] = "Sayonara" ;
    char target[ 20 ] ;
    xstrcpy ( target, source ) ;
    printf ( "source string = %s\n", source ) ;
    printf ( "target string = %s\n", target ) ;
    return 0 ;
}

void xstrcpy ( char *t, char *s )
{
    while ( *s != '\0' )
    {
        *t = *s ;
        s++ ;
        t++ ;
    }
}
```

```
*t = '\0' ;  
}
```

The output of the program is

source string = Sayonara

target string = Sayonara

Note that having copied the entire source string into the target string, it is necessary to place a ‘\0’ into the target string, to mark its end. If you look at the prototype of strcpy ( ) standard library function would be

```
strcpy ( char *t, const char *s ) ;
```

Even though we haven't used the keyword const in our custom version of xstrcpy ( ), the function has worked correctly. So what is the need of the const qualifier? What would happen if we add the following lines beyond the last statement of xstrcpy ( )?

```
s = s - 8 ;
```

```
*s = 'K' ;
```

This would change the source string to “Kayonara”. Can we not ensure that the source string doesn't change even accidentally in xstrcpy ( )? We can, by changing the definition as follows:

```
void xstrcpy ( char *t, const char *s )  
{  
    while ( *s != '\0' )  
    {  
        *t = *s ;  
  
        s++ ;  
  
        t++ ;  
    }  
    *t = '\0' ;  
}
```

By declaring char \*s as const, we are declaring that the source string should remain constant forever. Thus the const qualifier ensures that our program does not inadvertently

alter a variable that you intended to be a constant. It also reminds anybody reading the program listing that the variable is not intended to change. Let us understand the difference between the following two statements:

```
char str[ ] = "Quest";
```

```
char *p = "Quest";
```

Here str acts as a constant pointer to a string, whereas, p acts as a pointer to a constant string. As a result, observe which operations are permitted, and which are not:

```
str++;          /* error, constant pointer cannot change */
```

```
*str = 'Z';     /* works, because string is not constant */
```

```
p++;           /* works, because pointer is not constant */
```

## Recap

- ◆ Pointers are variables which hold addresses of other variables.
- ◆ A pointer to a pointer is a variable that holds the address of a pointer variable.
- ◆ The & operator fetches the address of the variable in memory.
- ◆ The \* operator lets us access the value present at an address in memory with an intention of reading it or modifying it.
- ◆ Dynamic Memory Allocation: a procedure in which the size of a data structure is changed during the runtime.
- ◆ The “malloc” or “memory allocation” method in C is used to dynamically allocate a single large block of memory with the specified size.
- ◆ “calloc” or “contiguous allocation” method in C is used to dynamically allocate the specified number of blocks of memory of the specified type.
- ◆ “free” method in C is used to dynamically deallocate the memory.
- ◆ “realloc” or “re-allocation” method in C is used to dynamically change the memory allocation of a previously allocated memory.
- ◆ Arrays are a kind of data structure that can store a fixed-size sequential collection of elements of the same type.

- ☐ Declaring Arrays : type arrayName [ arraySize ];

- ☐ Initializing Arrays :double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};



□ Accessing Array Elements : double salary = balance[9];

- ◆ Example program using Array- Declaration, Initialisation, Accessing Stored values from arrays

```
#include <stdio.h>

int main () {

int n[ 10 ]; /* n is an array of 10 integers */

int i,j;

/* initialize elements of array n to 0 */

for ( i = 0; i < 10; i++ ) {

n[ i ] = i + 100; /* set element at location i to i + 100 */

}

/* output each array element's value */

for ( j = 0; j < 10; j++ ) {

printf("Element[%d] = %d\n", j, n[j] );

}

return 0;

}
```

- ◆ Notations that refer same element in an array, name[ i ], \*( name + i ), \*( i + name ) and i[ name ].

## Objective Type Questions

1. What is a variable that stores the address of another variable?
2. Which operator is used to get value at the address stored in a pointer variable?
3. What is the output of the following code?

```
#include <stdio.h>

int main() {

int a=3, *b = &a;

printf("%d",a*b);

}
```

4. Which operator fetches the address of a variable in memory?
5. What is the output of printxy(1,1) in the following code?

```
void printxy(int x, int y)
{
    int *ptr;
    x = 0;
    ptr = &x;
    y = *ptr;
    *ptr = 1;
    printf(“%d,%d”, x, y);
}
```

6. What is the return type of malloc() or calloc()?
7. What is the usage of malloc() and calloc()?
8. Which method is used to dynamically allocate a single large block of memory with the specified size?
9. Which method in C is used to dynamically allocate the specified number of blocks of memory of the specified type?
10. Which method in C is used to dynamically de-allocate the memory.
11. “realloc” or “re-allocation” method in C is used to dynamically change the memory allocation of a previously allocated memory.
12. What is the procedure in which the size of a data structure is changed during the runtime?
13. Write the output of the following code:

```
#include <stdio.h>

int main () {
    char greeting[6] = {‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’};
    printf(“Hi your message is : %s\n”, greeting );
    return 0;
}
```

14. Write the output of the following code:

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

```

#include <string.h>

int main () {
    char str1[12] = "Hi";
    char str2[12] = "dear";
    char str3[12];
    int len ;

    /* copy string1 into string3 */
    strcpy(str3, str1);

    printf("strcpy( str3, str1) : %s\n", str3 );

    /* concatenates string1 and string2 */
    strcat( str1, str2);
    printf("strcat( str1, str2): %s\n", str1 );

    /* total length of string1 after concatenation */
    len = strlen(str1);

    printf("strlen(str1) : %d\n", len );

    return 0;
}

```

15. Write the output of the program given below

```

#include <stdio.h>

int main () {
    int *ptr = NULL;
    printf("The value of ptr is %x\n", ptr );
    return 0;
}

```

## Answers to Objective Type Questions

1. Pointer
2. \*
3. Invalid operands
4. &
5. 1,0
6. Void \*
7. Dynamic memory allocation
8. malloc()
9. calloc()
10. free()
11. realloc()
12. Dynamic memory allocation
13. Hi your message is: Hello
14. strcpy( str3, str1) : Hi  
    strcat( str1, str2): Hiear  
    strlen(str1) : 6
15. The value of ptr is 0

## Assignments

1. Write a program in C to calculate the sum of n numbers entered by the user.
2. Explain pointers in C programming.
3. What is dynamic memory allocation in the C programming language?
4. Differentiate between the \* and & operators with examples in C.
5. Explain different string functions in C programming.

## Reference

1. Balagurusamy, E. Programming in C. Tata McGraw-Hill Education, 2008.
2. Forouzan, Behrouz A. "Computer Science, A Structured Programming Approach Using C, Brooks." Cole. California, 2001.
3. Kernighan, Brian W., and Dennis M. Ritchie. The C programming language. Prentice Hall, Englewood Cliffs, N.J., 1988.
4. Gottfried, Byron S. Schaum's Outline of Programming in Structured Basic. McGraw-Hill Professional, 1992.

## BLOCK 3

# Functions, Structures and Union

```
#include "KMotionDef.h"
```

```
int main()
```

```
{
```

```
    ch0->Amp = 250;
```

```
    ch0->output_mode=MICROSTEP_MODE;
```

```
    ch0->Vmax=7000f;
```

```
    ch0->Jerk =2000f;
```

```
    ch0->Lead=0.0f;
```

```
    EnableMilestone(0.0);
```

```
    ch1->Amp = 250;
```

```
    ch1->output_mode=MICROSTEP_MODE;
```

```
    ch1->Vmax=7000f;
```

```
    ch1->Accel=1000f;
```

```
    ch1->Jerk =2000f;
```

```
    ch1->Lead=0.0f;
```

```
    EnableMilestone(0.0);
```

```
    DefineCoordSystem(0,1,-1,-1);
```

```
    return 0;
```

```
}
```





# Functions

## Learning Outcomes

After the successful completion of the unit, the learner will be able to:

- ◆ learn modular programming architecture in C
- ◆ achieve skills to implement a function
- ◆ familiarise nested functions and its usage in C programs

## Prerequisites

We learned how to write a program in C in the previous blocks. We will now have a quick look through the concepts.

**Data types:** We need to declare a variable to store data in memory, and have to specify the type of data to be represented by the variable, right?, and you know the data types permitted in C are char, int, float and double.

**Input/output statements:** We have familiarized ourselves with the use of various statements to receive data from the user, and, statements to display results on the screen.

**Control Structures:** The control flow in the program sequence can be managed using control structures. Now try to remember the use of if-then-else to manage conditional execution. Also, we know how to frame looping structures using for, while, or do-while structures.

All the programs we wrote were inside a block named `main()`, right? We called it the main function. Did you ever feel it complex to express every statement in a single main function? When we need to write long programs, consisting of multiple tasks, the program becomes lengthy, doesn't it? At times, we need to write the same group of statements in many portions.

What if we could divide it into separate blocks, instead of enlisting everything under the `main()`?

Why do people use assistants while doing a job? It will be easier to get the work done, right? Think of a mason doing building construction. Let it be a single wall. If he is to do everything, say mixing the sand, going to take suitable bricks, bringing water, measuring levels, everything by himself, Will the job be over in time?

What if the supervisor brings an assistant to the scene where the assistant will bring the suitable brick when the mason asks. If a big one is required, ask him to bring it. If you need half the size of a brick, ask him, he will break it and bring it. Just specify what you need, and the assistant will bring it on time.

The scene becomes brighter, doesn't it? The work goes smoothly now, the service of the assistant can be used by other masons also, if required!

Here, we were redefining the work of the main mason with assistants, or even other masons to help him.

Similarly, problems can also be divided into subproblems and solved independently. Proper execution of the subproblems leads the way to success.

We learned that C is a structured programming language. What do you understand about structured programming? As the name suggests, it is a programming methodology that makes use of control flow statements, looping statements, and subprograms. These features improve the quality and development time of a program. In the previous units, we learned about control flow statements(if/else) and looping statements(for/while/do-while). What about subprograms?

In this unit, we have one of the important concepts of structured programming called subprograms.

## Key Concepts

Modular Programming, Subprograms, Library functions, User-defined functions, Function prototype, Parameters, Function header, Nested functions, Code reusability

## Discussion

### 3.1.1 Modular Programming

We are all familiar with the positivity and success rate of teamwork. A group of persons combines individually with their

skill sets to complete a difficult task. In the following sections, we are going through a programming methodology known as modular programming which beautifully narrates the teamwork of small programs





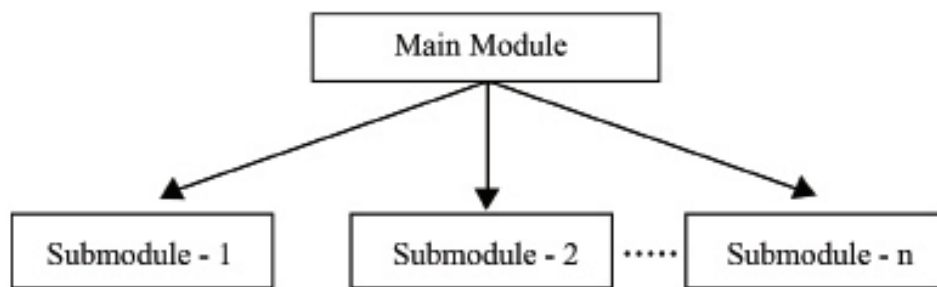


Fig 3.1.1 Modular Programming

within a single large program.

**Modular Programming** is a kind of teamwork where a program is divided into subprograms or modules. Each subprogram can perform specific functionality. By combining the subprograms into a single program, the desired output gets produced. That is each module that performs a specific task combines with other modules to complete the main module (Fig. 3.1.1).

On campus, it is a common practice to celebrate college day. For the successful running of this event, generally, committees will be set up. For example, the reception committee invites a chief guest and other delegates, the recreational committee, the technical-event committee, and so on. Each committee should be assigned specific tasks and guidelines to perform the task. Coordination and successful running of subcommittees lead

to the success of the entire event.

The above-mentioned strategy is similar to the modular programming concept, where each committee resembles subprograms. The efficient implementation of subprograms/modules leads to a successful outcome.

Modular programming is a general programming methodology.

For example, consider the most familiar program, the program to implement a calculator. Basically, there are four operations- addition, subtraction, multiplication, and division. Instead of writing a single module that performs the entire task, we can move into modular programming by subdividing the large single program into four subprograms - addition, subtraction, multiplication, and division (Fig 3.1.2).

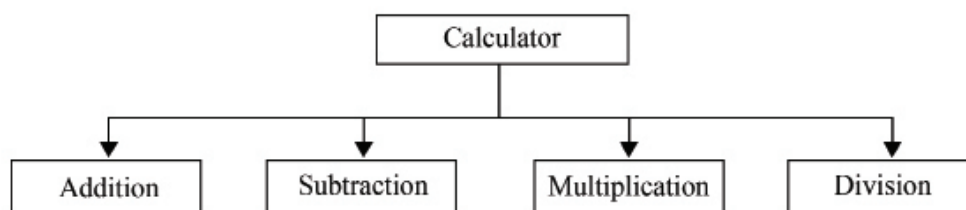


Fig 3.1.2 Subprograms of calculator

These subprograms or modules are also known as **functions**.

A function is a block of organized, reusable code that is used to perform a single, related task.

Functions are roles assigned to do a specific job. We can see it all around us. For the proper functioning of a department in an office, each person will be assigned a specific job. Each person works independently there, but the coordination of the staff and their functions makes the department operations easy.

There are two types of functions in C programming

- ◆ Built-in functions
- ◆ User-defined functions

### 3.1.2 Built-In Functions

Built-in functions are predefined functions in C. It is always available to the programmer and is also known as standard library functions.

The concept of built-in functions is similar to department sections in an office. Consider an attestation section; all persons coming for attestation purposes will be directed to this section without any

uncertainty. Likewise, built-in functions have specific jobs and are grouped together in a common place called header files.

For instance, library functions **printf()** and **scanf()** are associated with header file **stdio.h**. Before using the specified functions within the program, the header file should be linked to the program. The characteristics and definitions of built-in functions are defined within the related header file.

Syntax : `#include<header_filename.h>`

ex: `#include<stdio.h>`

### 3.1.3 User-Defined Functions

In C programs, users can define their own functions to perform a specific task. These are code segments written by the programmer itself.

Each programming function should have input and output. The function should contain instructions or statements to process these inputs to generate the desired output. Like any other program, the input and output can be in any form, such as integers, floating values, characters, arrays, etc. Different types of functions are shown in Fig 3.1.3.

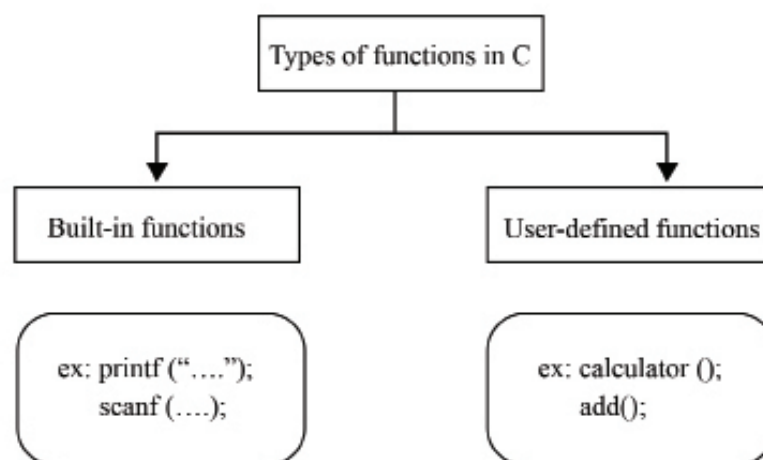


Fig 3.1.3 Classification of functions

Before writing a user-defined function, we have to discuss some of its basic characteristics.

A user-defined function has three phases

- ◆ function declaration
- ◆ function definition
- ◆ function call

### 3.1.4 Function Declaration

Remember, before we use a variable in a program, we must declare it. Similarly, a function declaration statement is necessary before using a function.

Note that the purpose of the function declaration is to identify the function name, number and type of input parameters (*that is why parameter names are not necessary*) and return type of the function, by the compiler. Function declaration statements are also known as function prototypes.

Did you notice the term parameter? Are you familiar with this? Parameters are the variables that are used during a function declaration or definition. To illustrate parameters, consider the admission procedure in a school. All the processing will be done in the office section; after that, the name list of the admitted

students for each course is forwarded to the corresponding department. Here, the parameter or data is the name list prepared by the office and used by another department.

A similar concept is applicable to the concept of functions.

For example, a function to add two integers can be declared as

ex: `int add (int x, int y);`

The above statement declares a function with the function name 'add', accepts two parameters of type integer (x and y) and returns an integer value.

It can also be written as,

`int add(int, int);`

Syntax: return-type function-name (list of parameters);

### 3.1.5 Function Definition

After function declaration, the code of the function has to be defined. Function definition contains the block of code to perform a specific task.

The function definition has two parts- The Function header and the Function body.

```
int add(int a,int b)           // function name - add, input parameters - a and b,
{                               return type of the function - int
    int c;
    c=a+b;
    return(c);                 //output parameter c
}
```

program 3.1.1

The function header includes the function name, return type and parameter list.

The Function body is the set of instructions defined in the function definition.

Consider the function **add()** as an example. The function adds two integer numbers (a,b) and returns another integer c as output.

In precise, the above code segment defines a function with function name **add()** and the input parameters given are integer variables a and b. In the function definition, parameters and the result assigned variable c are all integer type.

Note: Return type of a function is the type of the variable returned by the function.

### 3.1.6 Function call

In the above two sections, we discussed function declaration and function definition. However to invoke a function to perform a specific task, the function call is needed.

Calling a function means invoking a function to do a specific task.

eg: `add (5,7);`

The above function call sends the values 5 and 7 to the function definition. Variables a and b (function definition) accept these values and function execution occurs.

#### Flow of working of a function

```
#include<stdio.h>

int add(int,int);

int add(int a, int b)
{
    int c;
    c=a+b;
    return(c );
}

void main()
{
    int sum;
    sum = add(5,7);
    printf("Sum is %d",sum);
}
```

program 3.1.2

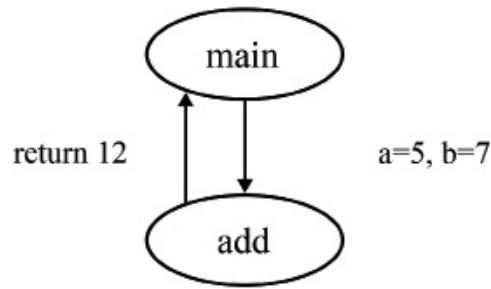


Fig 3.1.4 Working of function

How is a function implemented? What are the components? How are they interconnected? All these questions can be explained using program 3.1.2 and Fig 3.1.4.

When a program execution begins, as we know, the compiler starts execution from main (). Execution proceeds till it reaches the function call statement (Step 1). When a function is called, program control is transferred to the function definition (step 2), parameters from the function call are

copied to the function definition, and the code inside the function gets executed. After the execution of the return statement (Step 3) the program control is transferred to the calling function .

Note: A function can be defined either before the main() or after the main().

### 3.1.7 Examples of functions

The following program illustrates the implementation of function

program 3.1.3

**Example 1:** Program to implement a function for addition of 3 integers

```

#include<stdio.h>

int add (int, int ,int);           //function declaration

int add (int a, int b,int c)       //function definition
{
    int d;
    d=a+b+c;
    return(d);
}

void main ()
{
    int w,x,y,z;
    printf ("Enter three numbers");
  
```

```

scanf("%d%d%d",&w,&x,&y);

z=add(w,x,y);           //function call

printf("\nThe sum is %d",z);

}

```

### Output

```

Enter three numbers 1 2 4

The sum is 7

```

The above program defines the function add() which adds three integers and returns an integer value to the calling function.

### program 3.1.4

#### Example 2: Program to implement calculator

```

#include<stdio.h>

float add(float,float);           //function declaration

float sub(float,float);

float mul(float,float);

float div(float,float);

float add(float num_1, float num_2)    //function definition
{
float ans;
ans= num_1 + num_2;
return(ans);
}

float sub(float num_1, float num_2)
{
float ans2= num_1 - num_2;
return(ans2);
}

float mul(float num_1, float num_2)

```

```

{
float ans3= num_1 * num_2;
return(ans3);
}

float div(float num_1, float num_2)
{
float ans4= num_1 / num_2;
return(ans4);
}

void main()                                //Program execution begins here
{
float b,c;
float a,s,m,d;
printf("Calculator\n-----\n");
printf("Enter two numbers to perform operations\n");
scanf("%f %f",&b,&c);
a=add(b,c);
printf("The result of addition is %f\n",a);
s=sub(b,c);
printf("The result of subtraction is %f\n",s);
m=mul(b,c);
printf("The result of multiplication is %f\n",m);
d=div(b,c);
printf("The result of division is %f\n",d);
}

```

The code defines an operation for addition, subtraction, multiplication and division on two variables. The functions send two float values to the function definition and execution occurs there. The results are sent back to the called functions.

## Output

Calculator

-----

Enter two numbers to perform operations

2 3

The result of addition is 5.000000

The result of subtraction is -1.000000

The result of multiplication is 6.000000

The result of division is 0.666667

Concept of functions permits to divide a program into simple and smaller tasks. It makes the code to be called many times and implements the code reusability. For example, a function to find the sum of marks can also be used by a program to find the grade of students.

### Advantages of using functions

- ◆ The complexity of the program gets reduced by division of work
- ◆ Dividing a program into subprograms enables easier error handling
- ◆ A large program being divided into subprograms makes it easy to update
- ◆ Code reusability – Same code segment can be used in different programs

### 3.1.8 Nested Functions

**Functions defined within other functions are called nested functions**

#### Syntax:

```
fun1()
{
    fun2();
}
```

The concept of nested functions can be explained through an example. Previously, we discussed the function 'add' to sum three integers (Example 1). A program that finds out the average of three numbers has to perform the same sequence of steps additionally and a division operation.



Therefore, the function to find the average can use the function 'add' to calculate the average.

```
int avg(){  
    int k, result;  
    k=add();  
    result=k/3;  
    return(result);  
}
```

Function definition of 'average' calls the function add () and stores the sum in variable 'k'. The value of 'k' is divided by 3 and assigned as the average in variable 'result'.

Here, the function add() gets reused to find the average of given numbers.

#### program 3.1.5

**Example:** Program to find the average of 5 marks

```
#include<stdio.h>  
  
float avg();  
int add();  
float avg()                //Nested Function  
{  
    int sum=add();          //function call within a function  
    float avrg=sum/5;  
    return(avrg);  
}  
int add()  
{  
    int i,a[5],mark=0;  
    printf("Compute average marks of 5 subjects\n-----  
-----\n");  
    printf("Enter marks of 5 subjects\n");  
    for(i=0;i<5;i++)  
    {
```

```

        scanf("%d",&a[i]);
        mark=mark+a[i];
    }
    printf("Total mark is %d\n",mark);
    return(mark);
}
void main()
{
    float result;
    result=avg();
    printf ("The average of marks is %f", result);
}

```

### Output

Compute average marks of 5 subjects

-----

Enter marks of 5 subjects

20 30 89 90 50

Total mark is 279

The average of marks is 55.000000

## 3.1.9 Application Examples

### 3.1.9.1 Checking whether a number is odd or even using a function.

program 3.1.6

```

#include<stdio.h>

int odd_even(int);           /* Function Declaration or prototype*/

void main()
{
    int n,flag=0;
}

```



```

printf("\nGoing to check whether a number is even or odd");
printf("\nEnter the number: ");
scanf("%d",&n);
flag = odd_even(n);                                /* Function Calling*/
if(flag == 0)
{
    printf("\nThe number is odd");
}
else
{
    printf("\nThe number is even");
}
}
int odd_even(int n)                                /* Function Definition*/
{
    if(n%2 == 0)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

```

### Output

Going to check whether a number is even or odd

Enter the number: 4

The number is even

### Explanation of the program

- ◆ Input an integer number for checking odd or even numbers
- ◆ Then make a function call to the function odd\_even(int n) with an integer parameter.
- ◆ It will direct to the function definition
- ◆ In function definition taking the modulus of the number by 2 , if it is giving zero as input

### 3.1.9.2 Sorting using function

program 3.1.7

```
#include<stdio.h>

void sortarray(int a[], int n); /*function prototype or declaration*/

void main()
{
    int a[20],n,i;
    printf("Sort an array of numbers\n-----\n");
    printf("Enter the size of the array\n");
    scanf("%d",&n);
    printf("Enter the array elements\n");
    for(i=0;i<n;i++)
    {
        scanf("%d", &a[i]);
    }
    sortarray(a,n); /* function calling to function sortarray() */
}

void sortarray(int a[],int n)/*function definition for sorting*/
{
    int i,j,temp;
    for(i=1;i<n;i++)
    {
```

```

for(j=0;j<n-i;j++)
{
if(a[j]>a[j+1])
{
temp=a[j];
a[j]=a[j+1];
a[j+1]=temp;
}
}
}

printf("Elements after sorting\n");
for(i=0;i<n;i++)
{
printf("%d\t",a[i]);
}
}

```

### Output

Sort an array of numbers

-----

Enter the size of the array

5

Enter the array elements

5 3 1 3 4

Elements after sorting

1      3      3      4      5

### 3.1.9.3. Find the largest and second largest element in an array using function

program 3.1.8

```
#include<stdio.h>

void sortarray(int a[],int n); /*function prototype or declaration*/

void main()
{
    int a[20],n,i;
    printf("Enter the size of the array\n");
    scanf("%d",&n);
    printf("Enter the array elements\n");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);

    sortarray(a,n);          /* function calling to function sortarray() */
    printf("Largest element is %d \n",a[n-1]);
    printf("second largest element is %d \n",a[n-2]);
}

void sortarray(int a[],int n)          /*function definition*/
{
    int i,j,temp;
    for(i=1;i<n;i++)
    {
        for(j=0;j<n-i;j++)
        {
            if(a[j]>a[j+1])
            {
                temp=a[j];
                a[j]=a[j+1];
                a[j+1]=temp;
            }
        }
    }
}
```

```

    }
}
}

```

Output

Enter the size of the array

5

Enter the array elements

1 6 5 4 3

Largest element is 6

second largest element is 5

### 3.1.9.4 Matrix addition using functions

program 3.1.9

```

#include <stdio.h>

// function to add two matrix
void add_matrix(int a[10][10], int b[10][10],
               int c[10][10], int row, int column)
{
    for(int i=0; i< row; ++i)
    {
        for(int j=0; j< column; ++j)
        {
            // add & store to matrix C
            c[i][j] = a[i][j] + b[i][j];
        }
    }
}

// function to read matrix

```

```

void read_matrix(int matrix[10][10], int row, int column)
{
    for (int i = 0; i < row; ++i)
    {
        for (int j = 0; j < column; ++j)
        {
            scanf("%d", &matrix[i][j]);
        }
    }
}

// function to display matrix
void display_matrix(int matrix[10][10], int row, int column)
{
    for (int i = 0; i < row; ++i)
    {
        for (int j = 0; j < column; ++j)
        {
            printf("%d ", matrix[i][j]);
        }
        printf("\n"); // new line
    }
}

// main function
int main()
{
    // declare matrix matrix A, B, & C
    int a[10][10]; // first matrix

```



```

int b[10][10]; // second matrix
int c[10][10]; // resultant matrix
// read the size of matrices
int row, column;
printf("Enter Row and Column Sizes: ");
scanf("%d %d", &row, &column);
// read matrix A and B
printf("Enter Matrix-1 Elements: \n");
read_matrix(a, row, column);
printf("Enter Matrix-2 Elements: \n");
read_matrix(b, row, column);
// add both matrix A and B
add_matrix(a, b, c, row, column);
// display resultant matrix
printf("Resultant Matrix: \n");
display_matrix(c, row, column);
return 0;
}

```

### Output

Enter Row and Column Sizes: 2 3

Enter Matrix-1 Elements:

1 2 3 4 5 6

Enter Matrix-2 Elements:

2 4 6 8 10 12

Resultant Matrix:

3 6 9

12 15 18

### 3.1.9.5 Matrix multiplication using functions

program 3.1.10

```
#include<stdio.h>

void multiply(int r1, int c1, int r2, int c2);

int main()
{
    int i,j,k,r1,c1,r2,c2;
    printf("Enter row and column of first matrix\n");
    scanf("%d%d", &r1, &c1);
    printf("Enter row and column of second matrix\n");
    scanf("%d%d", &r2, &c2);
    multiply(r1,c1,r2,c2);
    return 0;
}

void multiply(int r1, int c1, int r2, int c2)
{
    int i,j,k;
    float a[10][10], b[10][10], mul[10][10];
    if(c1==r2) // condition check for matrix multiplication.
    {
        printf("Enter elements of first matrix:\n");
        for(i=0;i< r1;i++)
        {
            for(j=0;j< c1;j++)
            {
                printf("a[%d][%d]=",i,j);
                scanf("%f", &a[i][j]);
            }
        }
    }
}
```

```

printf("Enter elements of second matrix:\n");
for(i=0;i< r2;i++)
{
    for(j=0;j< c2;j++)
    {
        printf("b[%d][%d]=",i,j);
        scanf("%f", &b[i][j]);
    }
}
for(i=0;i< r1;i++)
{
    for(j=0;j< c2;j++)
    {
        mul[i][j] = 0;
        for(k=0;k< r2;k++)
        {
            mul[i][j] = mul[i][j] + a[i][k]*b[k][j];
        }
    }
}
printf("Multiplied matrix is:\n");
for(i=0;i< r1;i++)
{
    for(j=0;j< c2;j++)
    {
        printf("%f\t", mul[i][j]);
    }
    printf("\n");
}

```

```

    }
    else
    {
        printf("Dimension do not match for multiplication.");
    }
}

```

### Output:

Enter row and column of first matrix

2 3

Enter row and column of second matrix

3 2

Enter elements of first matrix:

a[0][0]=1

a[0][1]=2

a[0][2]=3

a[1][0]=4

a[1][1]=3

a[1][2]=2

Enter elements of second matrix:

b[0][0]=1

b[0][1]=2

b[1][0]=3

b[1][1]=4

b[2][0]=5

b[2][1]=6

Multiplied matrix is:

22.000000    28.000000

23.000000    32.000000

## Recap

- ◆ In a nutshell, functions are self-contained program segments. Library functions are built-in functions placed in a common place called the C standard library.
- ◆ User-defined function is provided by the developer of the program.
- ◆ In user-defined functions, when a function call occurs, the program control is transferred to the called function (function definition). The called function performs its specific task.
- ◆ When the function end is reached, the control will return to the calling function.
- ◆ Defining a function within another function increases the rate of code reusability, and also it implements efficient space utilization.
- ◆ Nesting of functions does not limit to a number of functions or types of data. However, the data passing techniques and visibility of data matter in implementation.

## Objective Type Questions

1. What is the default return type of function definition?
2. Which is the keyword used to send output obtained in a function to a called function?
3. What is the output of the following code?

```
void main()
{
    void f1(),f2();
    f2();
}
void f1()
{
    printf("5");
}
void f2()
```

```

{
printf("3");
fl();
}

```

4. What is the output of the following function?

```

void show() {
printf("abc");
show();
}

void main() {
printf("pqr");
return 10;
}

```

5. What are the two types of functions in C?  
 6. Every C program should contain which function?  
 7. What is the output of the following C program?

```

#include<stdio.h>

int main() {
int a = 20;
printf("HELLO ");
return 1;
printf("WORLD");
return 1;
}

```

## Answers to Objective Type Questions

1. int
2. return

3. 35
4. pqr program finished with exit code
5. Built-in functions and user-defined functions
6. main
7. HELLO

## Assignments

1. Write functions for subtraction and division for integer and float type variables
2. Write a program to find the factorial of a number using function.
3. Write a program to print the Fibonacci series using function.
4. Write the program to implement calculator using function (Use switch to accept a choice and perform a single operation based on the choice)
5. Discuss some real-time applications of nested functions.
6. Write a program to create a progress report of a student using nested functions. (total marks, average, grade)
7. Explain different types of functions in C programming.
8. How to declare and define a function in the C programming language.
9. Write a C programme to find sum and difference of two numbers using sum() and sub() functions.
10. Write a C programme to find the largest and smallest of two numbers using large() and small() functions

## Reference

1. Balagurusamy, E. Programming in C. Tata McGraw-Hill Education, 2008.
2. Forouzan, Behrouz A. "Computer Science, A Structured Programming Approach Using C, Brooks." Cole. California, 2001.
3. Kernighan, Brian W., and Dennis M. Ritchie. The C programming language. Prentice Hall, Englewood Cliffs, N.J., 1988.
4. Gottfried, Byron S. Schaum's Outline of Programming in Structured Basic. McGraw-Hill Professional, 1992.



## Recursion

### Learning Outcomes

After the successful completion of unit, the learner will be able to:

- ◆ achieve skills to implement recursive functions
- ◆ familiarise the operation of arrays on functions
- ◆ learn the working of pointers on arrays and functions

### Prerequisites

If a problem is too complex, we can adopt a method to break it down into simpler ones and combine the results. We do it in real life constantly.

Consider you have bus tickets of Rs. 12/- that you took in the previous month and now you need to count how much you have spent on. Since the number is large, you might ask for the help of your two friends and divide the tickets into three. When you finish counting, you add up the individual outcomes and get the final result.

In the above method, a complex task has been broken down into simple problems. Similarly, in programming, a single problem can be solved by dividing the problem into subparts and combining the results iteratively.

The method of breaking down a larger problem into subproblems and solving is implemented in C language by the use of functions, which we have studied already. The concept of storing similar data types in a single variable name – array, is also familiar to you.



## Discussion

### 3.2.1 Recursion

Generally, a function is called by another function. Have you thought about a function that calls itself? A function that calls itself is known as a recursive function and the process of calling a recursive function is known as recursion.

*How does recursion work?*

```
void function_1()
{
    function_1();
}

void main()
{
    function_1();
}
```

A typical recursive function will work as shown above. In main(), 'function\_1' is called and the control goes to the function definition. Inside the function definition, the function is again called and the procedure repeats. Have you noted any issues? When will it end? Recursive calls execute indefinitely until some condition is encountered to prevent it. To stop infinite recursion, an explicit condition should be specified within the function itself. An exit condition should be defined to stop the repeated function call, otherwise, it will enter into an infinite loop.

Recursion can be applied to programs like Fibonacci series, factorial, sum of n numbers, etc. The following is an example to find out the factorial of a number using a recursive function.

Factorial of  $n = 1 * 2 * 3 * \dots * n$

The same statement can be represented as

Factorial of  $n = n * (n-1) * (n-2) * \dots * 2 * 1$

e.g.:  $7! = 7 * 6 * 5 * 4 * 3 * 2 * 1$

The above steps can be represented in a recursive formula,

$7! = 7 * 6!$

**$n! = n * (n-1)!$**

Note that the factorial of a negative number doesn't exist, the factorial of 0 is 1. Now

let us move to the implementation of recursive program to find factorial of a number,

```
int fact(int n)
{
    if(n>1)
    {
        return(n*fact(n-1)); //Recursive call
    }
    else
    {
        return (1);    //exit - condition
    }
}
```

Statement  $n * \text{fact}(n-1)$  performs the recursive operation on function call. When the value of  $n$  is 3, the recursive call becomes  $\text{return}(3 * \text{fact}(3-1))$ ; ,which is equivalent to

$\text{return}(3 * \text{fact}(2))$ ; . -----(1)

To find the value of  $\text{fact}(2)$ , the function is again called with 2 as an argument. The else block is executed and becomes  $\text{return}(2 * \text{fact}(2-1))$ ;, which is equivalent to

$\text{return}(2 * \text{fact}(1))$ ;----- (2)

To find out the factorial of 1, the function is again called with 1 as an argument. Now the condition in the if statement gets executed and will return 1 as the value. Then equation (2) becomes  $\text{return}(2 * 1)$  and it returns value 2 to equation (1). The equation (1) will become  $\text{return}(3 * 2)$  which is equal to 6. The execution of the function is delayed till it reaches 1 which is the termination condition. When the function gets the last return value, it returns to the previous function calls.

The execution of the above program is shown in Fig. 3.2.1.

$5 * \text{fact}(4)$
$5 * 4 * \text{fact}(3)$
$20 * 3 * \text{fact}(2)$
$60 * 2 * \text{fact}(1)$
$120 * 1 = 120$

Fig: 3.2.1 Working of function evaluation

**Example:** Program to print Fibonacci series up to nth term.

```
#include<stdio.h>

int fibonacci(int);

int fibonacci(int k)
{
    if(k <= 1)
        return k;
    else
        return (fibonacci(k-1) + fibonacci(k-2));
}

int main() {
    int n, i;

    printf("\nDisplay first n Fibonacci numbers");
    printf("\nEnter n:");
    scanf("%d", &n);
    printf("\nFirst %d Fibonacci numbers are: ", n);
    for(i = 0; i < n; i++) {
        printf("%d ", fibonacci(i));
    }
}
```

### Output

Display first n Fibonacci numbers

Enter n:7

First 7 Fibonacci numbers are: 0 1 1 2 3 5 8

In the Fibonacci series, the next number is generated by adding the previous two numbers. That is, 2 is generated by adding (1+1), 3 by adding (1+2), and so on. In the above program, function fib() is called iteratively till output is obtained. In the function definition, if the value of the parameter is zero or one, then the value of parameter itself is returned. Otherwise, recursively call the function fib() with values (n-1) and (n-2).

For input 6,

fib(0) prints 0

fib(1) prints 1

fib(2) performs fib(1)+fib(0) and prints 1

fib(3) performs fib(2)+fib(1) and prints 2 and so on.

Recursive functions cannot be implemented on all functions, but only to problems that can be solved iteratively.

### 3.2.2 Types of Recursions

Recursive functions can be classified on the basis of whether the function calls itself directly or indirectly.

- ◆ Direct Recursion
- ◆ Indirect Recursion

#### Direct Recursion

When a function explicitly calls itself, it is called a direct recursive function. Function implementations (Factorial, Fibonacci) that we discussed previously are examples for direct recursion.

e.g.: - void fib(int k)

```
{  
    if(k <= 1)  
        return k;  
    else  
        return(fib(n-1)+fib( n-2));  
}
```

Here, the function fib() calls itself for all values greater than zero.

#### Indirect Recursion

In this method, the function invokes another function which again causes the original

function to be called again.

**Example:**

```
int func1(int n)
{
    if(n==0)
        return 0;
    else
        return(func2(n-1));
}

int func2(int n2)
{
    return (func1(n2-1));
}
```

In the above example, func1() calls func2(), which again calls func1(). It is called indirectly recursive or mutually recursive.

**Example:** Program to print numbers from 1 to 20 using indirect recursion.

```
#include<stdio.h>

int n = 1,N=20;

int fun_1()
{
    if (n <= N)
    {
        printf("%d\t", n);
        n++;
    }
}
```

```

        fun_2();
    }
    else
        return(0);
}

int fun_2()
{
    if (n <= N)
    {
        printf("%d\t", n);
        n++;
        fun_1();
    }
    else
        return(0);
}

void main()
{
    fun_1();
}

```

The program illustrates the execution of an indirect function in which fun\_1() calls fun\_2() and fun\_2() calls fun\_1() recursively till the condition gets satisfied.

### Output

1	2	3	4	5	6	7	8	9	10	11
12	13	14	15	16	17	18	19	20		

### 3.2.3 Tail recursion

It is a type of direct recursion. If the recursive call is the last statement in the function, then it is called tail recursion. After the recursive call, the recursive function performs nothing.

Example:

```
void fib(int k)
{
    if(k=0)
        return 0;
    else if(k=1)
        return 1;
    else
        return(fib(n-1)+fib( n-2))
}
```

### 3.2.4 Comparison between Recursion and Iteration

Recursion	Iteration
Always applied to a function	Always applied to a set of instructions which are repeatedly executed.
A function calls itself repeatedly	Loop repeatedly executes until the controlling condition becomes false
Terminates when a base condition is encountered	Terminates when the loop condition fails
Slower in execution	Faster in execution
Shorter code	Lengthier code

### 3.2.5 Advantages & Disadvantages of Recursion

#### Advantages

- ◆ Recursive problems are solved easily
- ◆ Lesser code for complex problems

#### Disadvantages

- ◆ Difficult to understand
- ◆ Execution speed decreases because of repeated function call

### 3.2.6 Arrays in Functions

We have already discussed the concepts of functions and arrays separately in previous units. Do you think it is possible to pass an array to a function? Yes, it is possible to pass an array in function... How can it be done? Is it similar to normal variables? The following sections discuss the implementation of arrays in functions.

#### Passing One-Dimensional Array to function

An entire array can be passed to a function by passing the array name without any subscripts and the size of an array, as arguments.

**Syntax : function\_name (array\_name, size);**

Consider a function to find the sum of marks.

```
sum(marks, arr-len);
```

In function call sum(), the array name is sent as a parameter to the function definition (array name itself points to the base address of an array). The corresponding function header is

```
int sum(int m[ ], size);
```

The function sum is defined to take two arguments- the array name and size of the array. The bracket pair informs the compiler that the argument m is an array. Function declaration for the above function is as follows.

```
int sum(int [ ], int);
```

Let us see how the entire program can be implemented.

```
#include<stdio.h>

int sum (int [ ], int);

void main()
```





```

{
    int result;
    int marks[5]={38,42,35,45,48};
    result=sum(marks,5);//base address sends as parameter
    printf ("The result is %d",result);
}
int sum(int m[],int size)
{
    int i,totalmarks=0;
    for(i=0;i<size;i++)
        totalmarks+=m[i];
    return(totalmarks);
}

```

### Output

The result is 208

The program consists of main and sum functions. main() reads the elements of the array "marks," and calls the function "sum" to print the sum of the array elements. The function is called by passing the array name and size. In the function definition, the formal parameter m must be an array type; the size of the array does not need to be specified in the subscripts. In a function prototype, it must show that the argument is an array.

It should be noted that when an entire array is passed to a function, the address details of the array elements are actually passed to the function, not the contents of the array. So any changes in array elements in the function are also reflected in the original array in the calling function.

Another method to *pass an array to a function is using pointers*. The base address of the array is sent to a pointer variable in the function.

Consider an integer array,

```
int a[3]={3,6,9};
```

```
int *p;           //Declaration of an integer pointer variable  
p=a;             // assigns the base address of the array to the pointer variable
```

Now the pointer variable 'p' points to the base address of the array.

```
printf("%d", *p);           //output 3  
printf("%d",*(p+1)); //output 6
```

Now, let us go to an example program to implement a function using pointers in arrays. The following program finds the sum of 5 marks stored in an array.

```
#include<stdio.h>  
  
int sum (int *,int);           //function declaration  
  
void main()  
{  
    int result,*p;  
    int marks[5]={38,42,35,45,48};  
    p=marks;//assigning base address to a pointer variable  
    result=sum(p,5);           //base address sends as parameter  
    printf ("The result is %d",result);  
}  
  
int sum(int *m,int size)       //function definition  
{  
    int i,totalmarks=0;  
    for(i=0;i<size;i++)  
        totalmarks+=m[i];  
    return(totalmarks);  
}
```

In the above program, function call sends base address and size of the array as parameters. At the same time, the function header defines a pointer variable (\*p) that accepts the base address as a formal argument. Function declaration specifies int \* to denote a pointer variable as an argument.

### 3.2.7 Two-Dimensional arrays

Fig 3.2.2 illustrates the representation of the following 2-D array.

e.g.: `int a[2][3]= {{1,2,3},{4,5,6}};`

<code>a[0][0]</code>	<code>a[0][1]</code>	<code>a[0][2]</code>
1	2	3
<code>a[1][0]</code>	<code>a[1][1]</code>	<code>a[1][2]</code>
4	5	6

Fig: 3.2.2 2-D Array representation

2-D arrays can be passed to a function by specifying the array name, row and column size. While specifying an array in functions, we must indicate that the array has two dimensions by indicating two sets of brackets, the first dimension need not have to be specified, and the second dimension should be given. See below given example programs.

The program given below prints 2-D matrix elements using function

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

```
int n;
```

```
void print(int a[][n],int,int);    //function declaration
```

```
void main()
```

```
{
```

```
    int a[10][10],r,c,i,j;
```

```
    printf("Enter row and column size:");
```

```
    scanf("%d %d",&r,&c);
```

```
    printf("Enter matrix elements:");
```

```
    for(i=0;i<r;i++)
```

```
    for(j=0;j<c;j++)
```

```
        scanf("%d",&a[i][j]);
```

```
    print(a,r,c);    //function call
```

```
}
```

```
void print(int arr[][10],int i,int j)//function definition
```

```
{  
    int c,d;  
    for(c=0;c<i;c++)  
    {  
        for(d=0;d<j;d++)  
        {  
            printf("%d\t",arr[c][d]);  
        }  
        printf("\n");  
    }  
}
```

### Output

Enter row and column size:2 3

Enter matrix elements:1 2 3 4 5 6

1	2	3
4	5	6

## Recap

- ◆ Recursion means the function is repeating or recurring its own execution.
- ◆ A function is called recursive if a statement within the body of a function calls the the same function.
- ◆ A recursive function performs a task by dividing it into subtasks.
- ◆ It is a method of solving a complex problem where the solution depends on sub solutions of smaller instances of the same problem.
- ◆ Recursion makes the program elegant.
- ◆ Recursive problems can also be solved by loop statements.
- ◆ Recursive code is shorter than iterative code.
- ◆ An array can be passed to a function by sending the array name(without subscripts) and size.
- ◆ Any changes made to an array passed as arguments reflect to the function call.
- ◆ For two-dimensional arrays, the function call specifies the array name( without subscripts), and row and column sizes.
- ◆ In function definition using arrays, along with rows and column sizes, the arguments should contain an array name with two pair of brackets
- ◆ Function declaration and function header should be similar

### Objective Type Questions

1. What will be the output?

```
void main()
{
    printf("Hai");
    main();
    return 0;
}
```

2. What will happen if an exit condition is absent in a recursive function?
3. Iteration requires less system memory than recursion. State True or False.
4. When is a function called direct recursive?
5. What is the core difference between iteration and recursion?
6. What will be the output?

```
void fun_1(int*);

void main()
{
    int i = 12, *p = &i;
    fun_1(p++);
}

void fun_1(int *p)
{
    printf("%d\n", *p);
}
```

7. What are the outputs?

```
void fun(int *);

void main()
{
    int i = 100, *p = &i;
    fun(&i);
    printf("%d", *p);
}

void fun(int *p)
{
    int j = 2;
    p = &j;
```

```
    printf("%d ", *p);
}
```

8. What is the maximum number of arguments that can be passed in a single function?
9. What will be the output?

```
void display(int*);
```

```
void display(int *p)
```

```
{
```

```
    int i = 0;
```

```
    for(i = 0; i < 5; i++)
```

```
        printf("%d\t", p[i]);
```

```
}
```

```
void main()
```

```
{
```

```
    int a[5] = {6, 5, 3};
```

```
    display(&a);
```

```
}
```

10. What will be the output?

```
void fun(int p, int q)
```

```
{
```

```
    printf("%d %d\n", p, q);
```

```
}
```

```
void main()
```

```
{
```

```
    int a = 6, b = 5;
```

```
    fun(a);  
}
```

11. What will be the output?

```
void change(int[]);  
  
void main()  
{  
    int a[3] = {50,55,56};  
  
    change(a);  
  
    printf(“%d %d”, *a, a[0]);  
}  
  
void change(int a[])  
{  
    a[0] = 86;  
}
```

## Answers to Objective Type Questions

1. Prints hai infinite number of times (Here main() function is called repeatedly without any exit condition)
2. Runs infinity and run into out of memory error
3. True
4. If it calls the same function recursively
5. Iteration will repetatively executes code block until the conditions is unmet, but recursion, the function calls itself in its body to solve the problem.
6. 12 (Hint: p points to i)
7. 2 100
8. No limits in number of arguments



9. 6      5      3      0      0      (Hint: array indexing)
10. Compile-time error      (Hint : too few arguments to function 'fun')
11. 86      86 (Hint : Uses call by reference, hence changes reflected)

## Assignments

1. Write a program to print n natural numbers using recursion.
2. Write a program to find the sum of n numbers using recursion.
3. Write a program to print the Fibonacci series up to a number using recursion.
4. Write a program to read two 1-D arrays and display their sum using a function.
5. Rewrite the above program using pointers.
6. Write a program to read two matrices and display the sum using a function.
7. Write a program to read two strings and concatenate without using the library function. (Use user-defined functions)
8. What is recursion? Write significance of recursion in programming?
9. Write a C program to find factorials of a number using recursive functions.
10. Explain different types of recursion in C programming.
11. How to pass an array in a function.
12. Write a note comparing recursion and iteration in programming.
13. Write a C program to print fibonacci series using recursive functions.

## Reference

1. Balagurusamy, E. Programming in C. Tata McGraw-Hill Education, 2008.
2. Forouzan, Behrouz A. "Computer Science, A Structured Programming Approach Using C, Brooks." Cole. California, 2001.
3. Kernighan, Brian W., and Dennis M. Ritchie. The C programming language. Prentice Hall, Englewood Cliffs, N.J., 1988.
4. Gottfried, Byron S. Schaum's Outline of Programming in Structured Basic. McGraw-Hill Professional, 1992.



## Call by Value and Call by Reference

### Learning Outcomes

After the successful completion of the unit, the learner will be able to:

- ◆ understand core concepts of data passing to functions
- ◆ understand parameter sharing methods
- ◆ learn to identify local and global variables

### Prerequisites

Data passing is the practice of transferring or sharing data used by one entity to other entity. Data can be of any type like contact numbers, addresses, location information, documents etc. It is a common procedure to share data by using various methods. In an office, approaches like direct sharing (person to person), emails, WhatsApp, written documents etc. are used as methods for data passing.

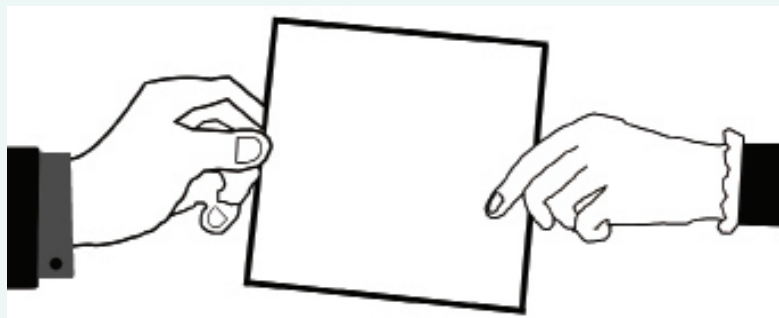


Fig 4.3.1 Data Sharing

In C, Modular Programming is an interesting concept and functions are a kind of top-down approach in modular programming. A large program is divided into subprograms called functions. In the previous unit, we learned about the concept of function, the basic components of a function, and flow of function execution etc.

Have you thought of how data from the calling function are shared or passed to the sub-routine? In this unit, we are going to understand how data is shared between functions, and various methods of data passing while dealing with functions.

## Key Concepts

Parameters/Arguments, Actual parameters, Formal Parameters, Local variables, Global Variables, Parameter Passing, Call by Value, Call by Reference

## Discussion

### 3.3.1 Basics of Parameter passing

In C, functions exchange information through parameters. In a function, variables present in the function call are passed to the function definition and vice versa. These variables are called parameters.

Arguments are actual values of the variables that get passed to the function.

To illustrate the difference between parameters and arguments, let us go to a real-time example;

When sending an e-mail, the mail sent is the parameter, and the content specified within the mail is the argument.

Likewise, consider a function `add()` which adds two integers and returns an integer value;

```
int add (int x, int y)
{
    int z = x + y;
    return (z);
}

void main ()
{
    int c;
    c = add (3, 7);
    printf("%d",c);
}
```

Output: 10

In the above code , values 3 and 7 are the arguments and variables x and y are the parameters.

*Function call invokes the function definition and passes the values to the parameters in the function header. This is known as **parameter passing**.*

There are two types of parameters in C

- ◆ **Formal Parameters:** The variables that appear in the function definition are formal parameters.
- ◆ **Actual Parameters:** The variables corresponding to the formal parameters that appear in the function call are actual parameters.

Consider the following example program,

```
#include<stdio.h>

int add(int ,int );

int add(int x,int y ) // x and y are formal parameters
{
    int z;

    z=x+y;

    return(z);
}

void main()
{
    int c,a,b;

    printf("Enter two integer values\n");
    scanf("%d %d",&a,&b);

    c=add(a,b);      //Here, a and b are the actual parameters
    printf("The sum is %d",c);
}
```

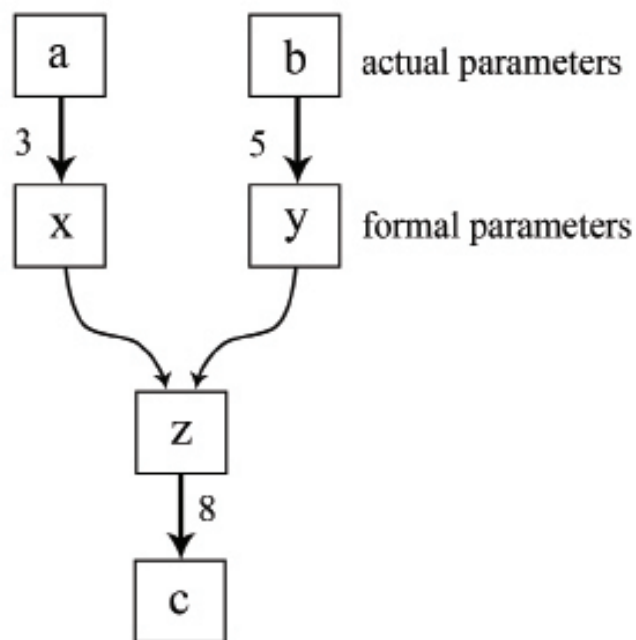
### Output

Enter two integer values

2      3

The sum is 5

Refer Fig 3.3.2 to understand the Call by Value method.



**Fig 3.3.2 Call by Value method**

Actual parameters are in the function call statement and when a function is called, the control will transfer to the function definition. The value of actual parameters is copied to the parameters present in the function definition and are called formal parameters.

In the above program, variables a and b are actual parameters and x and y are formal parameters

### **Local variables and Global variables**

Based on accessibility, variables can be classified into two- Local and Global variables.

Local variables are declared within a function and are accessible to that function only. Global variables are defined outside a function and are accessible to all functions in a program.

Let us discuss this concept of global variables and local variables with the help of some examples.

```
#include<stdio.h>

int num=5;

void main()
```

```
{  
printf("The value of num is %d",num);  
}
```

### **Output**

The value of num is 5

In the above program, variable num is globally defined. Since it is global, the variable is accessible throughout the program.

Consider another code segment,

```
#include<stdio.h>  
int num= 5;  
void main()  
{  
    int num= 2;  
    printf("The value of num is %d", num);  
}
```

### **Output**

The value of num is 2

In the above program, variable 'num' is assigned with two different values, one outside main() and one within main().

### **Output**

The value of num is 2

The value assigned within the main function gets printed because the variable 'num' is local to main().

Consider another example,

```

#include<stdio.h>

int num_1=5;           //Global variables

int num_2=12;

void main()
{
    int num_1= 2;       //local variable
    printf("Value of num_1 is %d",num_1);
    printf("\nValue of num_2 is %d",num_2);
}

```

### Output

```

Value of num_1 is 2
Value of num_2 is 12

```

Thinking!!Why did this happen?

In the program, num\_1 is declared both outside (globally) and inside the function (locally). When trying to print the variable within the function, the local value gets printed.

In the case of num\_2, it is not locally declared. Therefore, the global value is obtained as output. In the absence of local variables, we get global variables.

**Example:** Program to demonstrate the use of local variables and global variables

```

#include<stdio.h>

int num_1=5;           //Global variables

int num_2=12;

int incr(int);

void main()
{
    int num_1= 2;       //local variables

    int x,z;

    int decr(int);
}

```

```

printf("\nValue of num_1 is %d",num_1);
printf("\nValue of num_2 is %d",num_2);
z=incr(num_2);
printf("\nIncremented value of num_2 is %d",z);
x=decr(num_1);
printf("\nDecrement value of num_1 is %d",x);
}

int incr(int a)      // variable a is local to incr()
{
    a++;
    return(a);
}

int decr(int b)      //variable b is local to decr()
{
    b--;
    return(b);
}

```

### Output

Value of num\_1 is 2

Value of num\_2 is 12

Incremented value of num\_2 is 13

Decrement value of num\_1 is 1

### 3.3.2 Parameter Passing methods

Parameter passing entails information sharing. In our daily life, data can be passed using different strategies. Sharing information with employees in an organization can be done through emails, memos, notices, shared folders, etc. Each method follows different techniques and impacts implementation.



In C programming, variables can share information using assignment operators (a=b;). Likewise, variables in a function can pass information from the called function to the calling function and vice versa. Data sharing in functions can be done through different methods.

There are **two techniques** in C to send values to function definition from function call, from actual parameters to formal parameters.

- ◆ Pass by value
- ◆ Pass by reference

### 3.3.2.1 Pass by Value/ Call by value

This is the simplest and default method for parameter passing. In this method, the value of the actual parameter is copied to the formal parameters.

In all programming examples we discussed in previous sections, we were using call by value method for parameter passing.

Let us discuss a code segment to explain the call by value method.

```
#include<stdio.h>

void incr(int x,int y )
{
    x+=y;
    printf("\nThe value of x in function incr is %d",x);
}

void main()
{
    int a,b;

    printf("Enter two values\n");
    scanf("%d %d",&a,&b);
    printf("\nThe value of a in main is %d",a);
    incr(a,b);
}
```

## Output

Enter two values

2 3

The value of a in main is 2

The value of x in function incr is 5

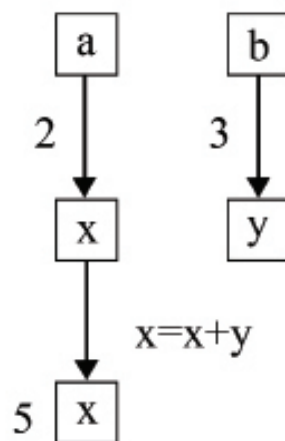


Fig 3.3.3 Call by Value

Here, values of variables 'a' and 'b' are copied to 'x' and 'y' (Fig 3.3.3). Function code executes on formal variables 'x' and 'y' and all changes are made on these variables. Hence, any change made in the formal parameter will not get reflected in the actual parameters.

### 3.3.2.2 Pass by reference/Call by reference

Passing of parameters in functions involves sending input parameters to functions and receiving output parameters from functions. One method for parameter passing is *call by value*, where the values of actual parameters are copied to formal parameters. Meanwhile, the change in actual parameter values will not get reflected in formal parameters.

In call by reference method, passing arguments to a function definition passing the address of an argument into the formal parameter

In this method, the changes in values of formal parameters are reflected to the actual parameters. For this, *the address of actual parameters is being sent to the formal parameters*.

Consider the following example to demonstrate this,

```
a=20;
```

```
b= 32;
```

**function call -**

```
reset (&a, &b);           //address of the parameters are sent
```

**function definition -**

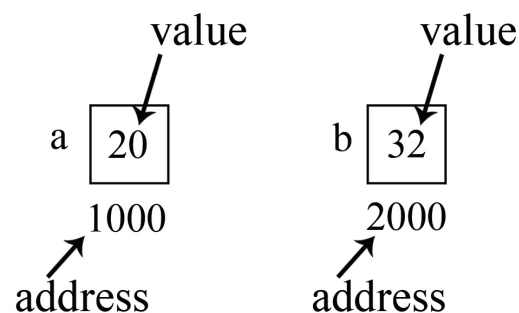
```
void reset (int *x, int *y)  //address copied to pointer variables
```

```
{
```

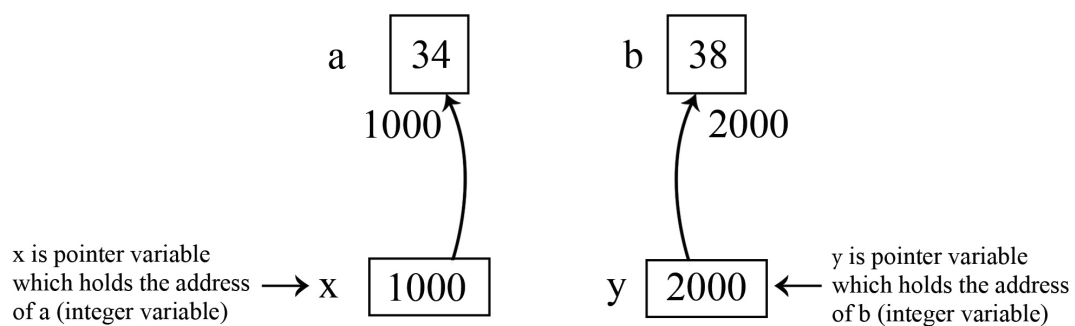
```
    *x=34;
```

```
    *y=38;
```

```
}
```



Before function call `reset (&a , &b)`



After function call `reset (&a , &b)`

Fig: 3.3.4 Call by Reference

Both the parameters (`a` and `x`) are pointed to the same memory location. Therefore, changes in the value of memory get reflected in the other pointed variables (Fig 3.3.4). ie, changes made in variables `x` and `y` are also reflected in formal parameters, `a` and `b`. Now the value of `a` became 34 and `b` became 38.

**Example:** Program to swap two numbers

```
#include<stdio.h>

void swap(int *,int*);           // function declaration

void main()
{
    int x,y;
    printf("Enter two numbers: ");
    scanf("%d %d",&x,&y);

    printf("\nBefore swapping x and y %d %d",x,y);
    swap(&x,&y);                 //function call
    printf("\nAfter swapping x and y %d %d",x,y);
}

void swap(int *a,int *b)        //function definition
{
    int t;

    t=*a;
    *a=*b;
    *b=t;
}
```

### Output

Enter two numbers: 2 3

Before swapping x and y 2 3

After swapping x and y 3 2

swap (int\*, int\*) declares a function that accepts pointer variables as parameters. In the main function, swap (&x,&y) calls the function named swap with arguments address of x and y. Function definition accepts these memory addresses as contents in formal arguments(\*a,\*b). Addresses get swapped, meanwhile, the contents of the addresses are also exchanged (Fig. 3.3.5).

Call by reference changes the contents of both actual arguments and formal arguments, whereas in the call by value, the contents of formal arguments are not changed.

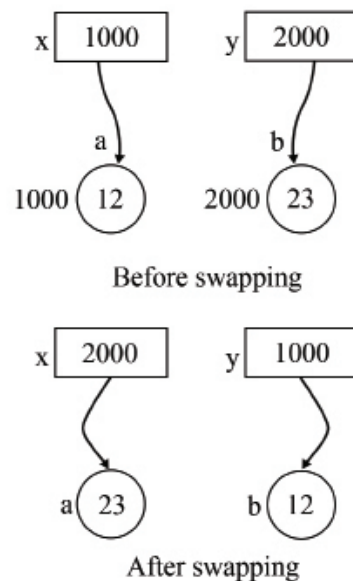


Fig. 3.3.5 Call by reference method

## Recap

- ◆ In a nutshell, an element declared within a function is local to that function.
- ◆ The value of the variable is accessible within that function only. Such variables are called local variables.
- ◆ Global variables are globally defined and are accessible to all functions within a program.
- ◆ Local variables are created during the execution of a function and its scope ends when the function terminates.
- ◆ Global variables are created when the program starts execution and come to an end when the program terminates.
- ◆ When a variable is locally and globally defined, the global variable gets shadowed. If it is not defined locally, global values are used.
- ◆ Execution of functions requires proper data transfer. Call by value and call by reference are two methods for data sharing.
- ◆ The first method copies the value of arguments from function call to function definition, whereas the second method passes a reference of the arguments from the called function to the calling function.

- ◆ In the call-by-reference method, any changes made to the reference variable are passed to the actual arguments.
- ◆ Normal functions can return a single value; meanwhile, using call by reference method, a function can return multiple values.

## Objective Type Questions

1. Arguments passed to a function are called \_\_\_\_ arguments
2. State True or False. In a function, the number and type of arguments should be the same when sending and receiving.
3. If a local variable with the same name as a global variable exists, which one will be used?
4. The default parameter passing technique is \_\_\_\_
5. What will be the output?

```
void main()
{
    auto int x;
    print("%d",x);
}
```

6. void fctn(int c)

```
{
c=90;
}
```

```
void main()
```

```
{
    int d=60;
    fctn(d);
}
```

What will be the value of variable 'd' after execution?

7. For the above code, the value of 'c' will be \_\_\_\_.
8. int &num; it is used in which method?
9. State True/False. Call by Value cannot change the value of actual parameters
10. What is the Output of the following code segment?

```
#include<stdio.h>

void incr(int *var)
{
    *var=*var+1
}

void main()
{
    int q=23;
    incr(&q);
    printf("The value of variable after function call is %d",&q);
}
```

## Answers to Objective Type Questions

1. Actual arguments
2. True
3. Local variable (Preference is given to the local variable within the function, global variable is shadowed.)
4. Call by Value
5. Some random number will be generated
6. 60 //pass by value method.
7. 90
8. Pass by reference
9. True
10. 24

## Assignments

1. Write a program to print the square and factorial of a number using functions. (use switch..case to select option)
2. Write a program to print n prime numbers using function (Pass 'n' as an argument)
3. List out the advantages and disadvantages of Pass by value and Pass by reference method.
4. Write a program to swap two numbers with function using both methods and discuss what changes will reflect in both program outputs.
5. Explain global variable and local variable.
6. Write a program to illustrate pass by value method and pass by reference method.
7. What is a parameter in a function declaration? What are the different types of parameters? Explain parameter passing mechanism in C programming.
8. Explain call by value method and call by reference method with examples.
9. Write a c program to find the largest among two numbers using call by value method and call by value reference method.

## Reference

1. Balagurusamy, E. Programming in C. Tata McGraw-Hill Education, 2008.
2. Forouzan, Behrouz A. "Computer Science, A Structured Programming Approach Using C, Brooks." Cole. California, 2001.
3. Kernighan, Brian W., and Dennis M. Ritchie. The C programming language. Prentice Hall, Englewood Cliffs, N.J., 1988.
4. Gottfried, Byron S. Schaum's Outline of Programming in Structured Basic. McGraw-Hill Professional, 1992.





## Structures and Union

### Learning Outcomes

After the successful completion of the unit, the learner will be able to:

- ◆ familiarize the way to write programs using structure
- ◆ discuss the concept of Nested structures
- ◆ introduce concepts of functions in structures
- ◆ define structure and union
- ◆ introduce the concept of implementing structure pointers to functions.
- ◆ introduce the concept of implementing union in a program

### Prerequisites

People handle a variety of data for different purposes. Each data belongs to different categories. These data can have the same or different properties. For example, data can take the form of a name, date, number etc.

In C language, data can be integers, characters, strings, floating point numbers and so on. In previous units, we discussed array handling, which stores only same type of data items.

For some applications, this might not be enough. For example, consider the address data of an organization. The address contains values like house number, street name, pin number, state etc. How to represent this kind of data? Real-time applications contain various types of data.

In this unit, we are familiarizing a new user-defined data type called structure which treats related data items which fall under different data types as a single entity. Modularity in programming is inevitable in all programming languages. Dividing a program into modules or subfunctions increases the code reusability, efficiency and readability.

This unit also discusses how the concept of functions is carried out in programs with different types of data.

‘Structure’ has many advantages and at the same time, it has some disadvantages too, especially in terms of memory allocation and utilization.

All of us are familiar with prepaid taxis and shared taxis. In prepaid taxis, you pay in advance for the cab, depending on the distance. But shared taxis take passengers on a fixed or semi-fixed route without a schedule.

The concept of structures is similar to prepaid taxis. When we create a structure variable, memory is allocated for all the member variables defined. Even if you are a single person traveling, the taxi seats are reserved for you. Likewise, memory is allocated for all the member variables, even if it is not used.

Do you know any method to overcome this?

The concept of union suggests solutions for disadvantages in structure, even though it inherits many properties of structure implementation.

## Key Concepts

User-defined data types, Structure, Nested Structure, Structure Pointers, Union, Union of Structures, Enumerated data type, Type casting

## Discussion

### 3.4.1 Concept of Structure

C programming allows defining variables that include several data items of the same type, however the structure is another user-defined data type which allows to include data items of different types. Structures are used for the record of data. For example, a student record not only consists of a name, it also includes roll\_no, marks, class, grade etc. Data types like int, and float, char, arrays are not sufficient for handling data items like this. **Structure is a user defined data type which can hold data items of different types under a single name.**

### 3.4.2 Define a structure

Structure is a heterogeneous collection of elements. Keyword ‘**struct**’ is used to define a structure. The struct statement defines a new data type, with more than one element.

Syntax – struct struture name

```
{  
  
    data-type member_element1;  
  
    data-type member_element2;  
  
    .....  
  
} structure variables(optional);
```

Note that, structure variables are optional and can initialize later and a struct definition should end with a semicolon.

e.g : struct student

```
{  
  
    char name[10];  
  
    int class;  
  
    int roll_no;  
  
    int marks[6];  
  
    float percent;  
  
} s1,s2;
```

Data items included in a structure definition are known as *member variables or member elements*. The above code defines a structure student with 5 member variables, one character array, two integer variables, one integer array and one float variable. In C, two methods are there to create structure variables. It can be declared either with the structure definition or like a basic data type declaration. The first method is demonstrated above, where structure variables (s1,s2) are defined along with the structure definition.

It can be also created using the keyword 'struct' and structure name like a basic data type from the calling function. Refer to the code below where struct vaiables are defined inside main function

e.g: struct student

```
{  
  
    char name[10];  
  
    int class;
```

```

        int roll_no;

        float percentage;
    };

void main()
{
    struct student s1,s2;
}

```

### 3.4.3 Structure elements in memory

When a structure is defined, no memory is allocated. We need to create structure variables to allocate the memory of a given structure type. The data elements of a structure are always stored in contiguous memory locations.

For example, for the above structure, memory allocation is as follows (in older versions, the size of an int was 2 bytes, but now it is 4 bytes. However, here we are considering size of an integer as 2 bytes).

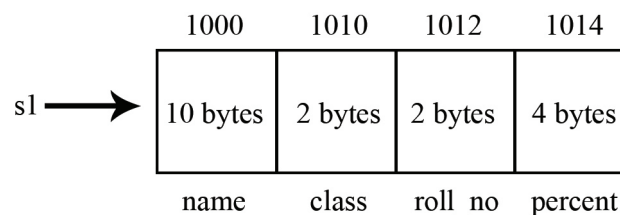


Fig: 3.4.1 Memory allocation for student structure

- ◆ Structure member elements are stored in contiguous memory locations (Fig 3.4.1).
- ◆ Size of a structure variable is the sum of size of data elements. For example, in student structure, size of structure = size(name) + size(class) + size(roll\_no) + size(percent) = 10 + 2 + 2 + 4 = 18 bytes

### 3.4.4 Accessing data members

Members of a structure can be accessed using two operators.

- ◆ Dot(.) operator is the member access operator, to normally access member variables.
- ◆ Arrow(->) operator to access member variables using pointers.
- ◆ Both operators are used to refer to individual data elements of a structure.

- **Accessing member variables using member access operator dot (.)**

**Member access operator(.)** works as a connector between the structure variable and member variables.

**Syntax: structure variable name.member variable name**

e.g.: s1.name

This statement access name of a student of s1

Structure members are initialized as follows,

```
s1.name="Ram";
```

The above statement assigns value 'Ram' to the name of student of s1

Another method is **designated initialization**,

```
struct student s1={name ="Ram", class = 11, roll no=13, mark = 32, 26, 42, Percent = 30.00};
```

This method allows you to specify which elements of a structure are to be initialized by the values. All members that are not initialized are zero-initialized. On the other hand, all data elements can be assigned in a single assignment.

e.g. : struct student s = {"Ram",11,13,79.3};

Remember, in the above initialization method, the values assigned should be in order with respect to the structure definition.

**Example:** Program to store data of a student using structure.

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

struct student           //structure definition
{
    char name[10];        //member variables
    int class;
    int roll_no;
    int marks [5];
} stud;                  // structure variable stud

void main()
{
```

```

    int i;

    printf("\nEnter name: ");
    scanf("%s",stud.name);
    printf("\nEnter class: ");
    scanf("%d",&stud.class);
    printf("\nEnter roll number: ");
    scanf("%d",&stud.roll_no);
    printf("Enter marks of 5 subjects: ");
    for(i=0; i<5;i++)
    {
        scanf("%d",&stud.marks[i]);
    }
    printf("Student Details \n");
    printf("Name:%s\nClass:%d\nRollnumber:%d",stud.name,stud.class,
stud.roll_no);
    printf("\nMarks\n");
    for(i=0;i<5;i++)
    {
        printf("%d\t",stud.marks[i]);
    }
}

```

### Output

Enter name: Tom

Enter class: 10

Enter roll number: 21

Enter marks of 5 subjects: 78 89 90 97 90

Student Details

Name:Tom

Class:10

Rollnumber:21

Marks

78      89      90      97      90

Note: Structure definition can also be included within the main() function.

- **Accessing member variable using arrow operator (->)**

Member variables are also accessed **using the arrow operator(->)**. The arrow operator is commonly used with a pointer to a structure variable.

How do pointers work through structures? What are the differences made compared to the conventional method!!

### 3.4.5 Pointers to Structure

A structure variable groups variables of different types under a single name. Each structure variable is associated with a block of memory which contains the entire data defined within the structure definition. Pointer makes a reference to the address of the memory location that stores a structure variable. Pointer which points to the address of the memory block that stores a structure is known as a Structure pointer.

e.g:-struct stud

```
{
    char name[10];
    int roll_no;
}s;

void main()
{
    struct stud *ptr= &s;    //ptr is a structure pointer points to
the structure variable S
}
```

Now 'ptr' points to the structure variable 's'. Memory representation of a structure pointer is demonstrated in Fig 3.4.2.

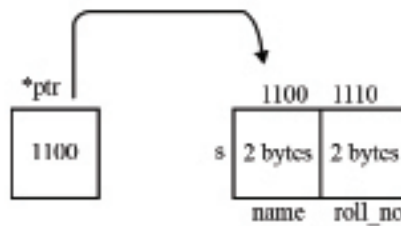


Fig. 3.4.2 Structure pointer

The following program implements the previous program of student details using pointers.

```
#include<stdio.h>

struct student {           //structure definition
    char name[10];         //member variables
    int class;
    int roll_no;
    int marks [5];
} stud;

void main()
{
    int i;

    struct student *p = &stud;    //initialization of structure pointer
    printf("Enter name\n");
    scanf("%s",&p->name);
    printf("\nEnter class\n");
    scanf("%d",&p->class);
    printf("\nEnter roll number\n");
    scanf("%d",&p->roll_no);
    printf("Enter marks of 6 subjects\n");
```



```

    for(i=0; i<6; i++) {
        scanf("%d",&p->marks[i]);
    }
    printf("Student Details \n");
    printf("Name:%s\t Class : %d\tRollnumber%d\n", p->name, p->class,p-
>roll_no);
    printf("Marks\n");
    for(i=0; i<6; i++) {
        printf("%d\t",p->marks[i]);
    }
}

```

### Output

Enter name

Tom

Enter class

10

Enter roll number

21

Enter marks of 6 subjects

89 98 56 78 56 87

Student Details

Name:Tom      Class : 10      Rollnumber21

Marks

89      98      56      78      56      87

In this example, the address of stud is stored in pointer p, p=&stud. Then you can access members of the stud using pointer p.

### 3.4.6 Array of Structure

In application level programming, it is not enough to operate a single record using structure. In a class, there are a number of students and each student has their own details. Likewise, employee details in an organization include details of many employees. In the above example, data of a single student is manipulated. In real applications, groups of students and records have to be manipulated. Array of structure is a finite collection of structure variables of same type. In array of structures, each element of an array is of type "structure".

An array of structure is declared as follows:

```
e.g :-    struct student
           {
           char name[10];
           int class;
           int roll_no;
           int marks[6];
           float percent;
           }stud[10];           //array of 10 students
```

stud[10] creates a collection of student variables, stud[0],stud[1],....stud[9], each storing different data. Each student structure variable holds associated data members and can be accessed through dot(.) operator with array index .

An individual element is referred to as,

```
stud [0].name="Ram";

stud[0].class= 12;
```

It assigns values to member variables "name" and "class" of the first student in the structure array, stud[0]

To generalize, the assignment operation can be done as follows,

```
stud[i].class=12;
```

The above statement assigns the value to class for i-th students in the structure array. The program given below read details of a set of students in a class.

```
#include<stdio.h>

#include<string.h>

struct student
```

```

{                                //structure definition
    char name[10];              //member variables
    int class;
    int roll_no;
    int marks[6];
    int totalmarks;
} stud[40];                      // structure variable for 40 students

void main ()
{
    int i, num, j;
    printf ("Enter number of students\n");
    scanf ("%d", &num);
    for (i = 0; i < num; i++)
    {
        printf("Enter details of student%d", i+1);
        printf ("Enter name\n");
        scanf ("%s", stud[i].name);
        printf ("\nEnter class\n");
        scanf ("%d", &stud[i].class);
        printf ("\nEnter roll number\n");
        scanf ("%d", &stud[i].roll_no);
        printf ("\nEnter marks of 6 subjects\n");
        for (j = 0; j < 6; j++)
        {
            scanf ("%d", &stud[i].marks[j]);
            stud[i].totalmarks += stud[i].marks[j];
        }
    }
}

```

```

for (i = 0; i < num; i++)
{
    printf("Details of student %d", i+1)
    printf ("\nName:%s\tClass:%d\tRoll no:%d\tTotal marks:%d\t",
            stud[i].name, stud[i].class, stud[i].roll_no,
            stud[i].totalmarks);
}
}

```

### Output

Enter number of students

2

Enter details of student 1

Enter name

Tom

Enter class

10

Enter roll number

21

Enter marks of 6 subjects

90 99 98 97 96 96

Enter details of student 2

Enter name

Jerry

Enter class

10

Enter roll number

22

Enter marks of 6 subjects

90 89 87 96 95 86

Details of Student 1

Name:Tom    Class:10    Roll no:21    Total marks:576

Details of Student 2

Name:Jerry    Class:10    Roll no:22    Total marks:543

Assignment operator can assign or copy the value of a structure variable to another structure variable which is of the same type. The following expressions demonstrate different methods to copy value of structure variables to another.

```
struct student s1,s2;

s1.age = s2.age;           // age of s2 gets copied to age of s1

strcpy(s1.address,s2.address);

s1=s2;                     //copying all elements to another variable
```

### 3.4.7 Nested Structures

A structure may consist of structures inside it and is known as nested structure. For example, consider an employee record which has data members name, employee id, address, phone number. The attribute address has subparts like house number, street number, city, state, pin code. Therefore, addresses have to be stored in separate structures. Nested structures are also known as embedded structures. We can define the employee record as follows.

```
struct employee
{
    char name[20];
    int emp_id;
    struct address           //Nested structure definition
    {
        int house_no;
        int street_no;
        char city[10];
        char state[10];
        int pin_no;
    }ads ;
}
```

```
int phone_no;

} emp;
```

The members of the nested structure can be accessed using dot(.) operator.

**Syntax: outer structure variable.inner structure variable.member element;**

e.g:-emp.ads.pin\_no=691302;

The following code shows the implementation of employee record using nested structure.

```
#include <stdio.h>

struct employee
{
    char name[20];
    int emp_id;
    struct address          //Nested structure definition
    {
        int house_no;
        int street_no;
        char city[10];
        char state[10];
        int pin_no;
    }ads ;
    int phone_no;
} emp;

void main()
{
    printf("Enter name:");
    scanf("%s",emp.name);
```

```

    printf("Enter employee id:");
    scanf("%d",&emp.emp_id);
    printf("Enter address\n");
    printf("Enter house number:" );
    scanf("%d",&emp.ads.house_no);
    printf("Enter street number:");
    scanf("%d",&emp.ads.street_no);
    printf("Enter city:");
    scanf("%s",&emp.ads.city);
    printf("Enter State:");
    scanf("%s",&emp.ads.state);
    printf("Enter pin number:");
    scanf("%d",&emp.ads.pin_no);
    printf("The entered details are\n");

    printf("Name :%s\n id: %d\nAddress : \n House NO : %d\n Street no :
%d\nCity : %s \nState : %s\n Pin no %d", emp.name, emp.emp_id, emp.ads.
house_no,emp.ads.street_no,emp.ads.city, emp.ads.state,emp.ads.pin_no);
}

```

### Output

Enter name:Tom

Enter employee id:21

Enter address

Enter house number:2

Enter street number:5

Enter city:Kollam

Enter State:Kerala

Enter pin number:691001

The entered details are

Name :Tom  
id: 21  
Address :  
House NO : 2  
Street no : 5  
City : Kollam  
State : Kerala  
Pin no 691001

Employee record comprises data name, employee id and address, where address covers multiple fields like house number, street number, city etc. Structure definition for 'employee' is defined as a nested structure which contains structure 'address' within it. 'emp.member\_variable' accesses the structure data members which are directly defined within the structure, whereas, 'emp.ads.member\_variable' accesses the nested structure members which are defined in structure address nested in structure employee.

### 3.4.8 Passing Structures in Functions

To pass a structure to a function, a structure variable is passed as a parameter to the function.

Consider the student structure defined below,

```
struct student
{
    char name[10];
    int class;
    int roll_no;
    int marks[6];
}stud;
```

The following function displays the student details and the function gets the student variable stud as formal parameter. Same structure type variable s receives the argument and performs the operations.



```

void print(struct student); // function declaration

void print(struct student s) // function definition
{
    int i;
    printf("Student Details \n");
    printf("Name:%s\t Class : %d\tRollnumber%d\n", s.name,
           s.class,s.roll_no);
    printf("Marks\n");
    for(i=0;i<6;i++)
    {
        printf("%d\t",stud.marks[i]);
    }
}

void main()
{
    int i;
    printf("Enter name\n");
    scanf("%s",stud.name);
    printf("\nEnter class\n");
    scanf("%d",&stud.class);
    printf("\nEnter roll number\n");
    scanf("%d",&stud.roll_no);
    printf("Enter marks of 6 subjects\n");
    for(i=0;i<6;i++)
    {
        scanf("%d\n",&stud.marks[i]);
    }

    print(stud); //function call
}

```

Note that, Structure type is necessary in function declaration. Individual Structure elements can also be passed to functions.

### 3.4.9 Concepts of Union

Union is a user-defined data type that is used to store different types of data elements. A union can define many members, but only one member can contain a value at any given time. Union provides an efficient way of using memory by storing different data types in the same memory location.

#### 3.4.10 Define a Union

The syntax to define a union follows the same as that of structure, but the keyword 'union' is used in place of the struct.

```
Syntax:      union u_name
               {
               member definitions;
               } union variables;
```

In union definition, u\_name tag is optional and member definitions are normal variable definitions. You can specify one or more union variables before the final semicolon, but it is optional. If the union variable is not specified with the definition, it can be defined separately within the main() function.

```
e.g.:      union dimen
               {
               int x;
               char c;
               float f;
               } d;
```

or

```
union dimen d;
```

This declares variable d of type union dimen. The union contains three members each with a different data type. But only one can be used at a time due to the fact that only one location is allocated for a union variable, irrespective of its size. This is a major difference between structure and union in terms of storage. We know, in structures, each member has its own memory location, but all the members of the union share the same memory location.

*Memory allocation* for union type occurs at the time of union variable creation. For a single variable (here 'd'), the same memory location can be used to store multiple types of data (Fig 3.4.1). For the above union definition, the size of d=4 (largest size

among the data members). This is because, in the above example, the maximum space is occupied by float value. The memory occupied by the union is large enough to hold the largest member of the union. (in older versions, the size of an int was 2 bytes, but now it is 4 bytes. However, here we are considering size of an integer as 2 bytes)

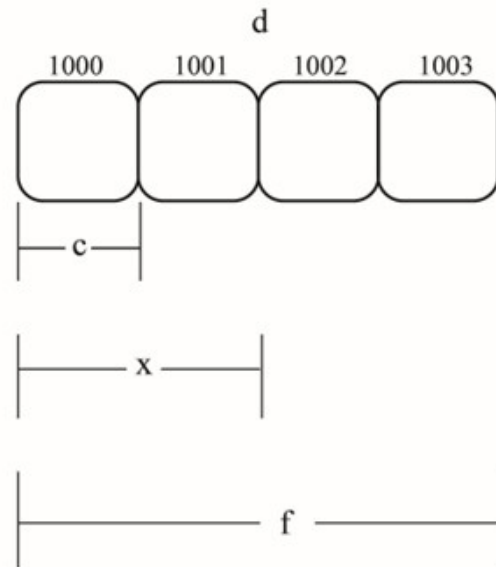


Fig 3.4.1 Memory Allocation in Union

The compiler allocates storage that is large enough to hold the largest variable type specified in union. Fig 3.4.1 shows how all the members share the same address. This implies that, even if a union may contain many elements of different data types, it can handle only one member at a time.

### 3.4.11 Accessing Union Members

Like structure, member access dot(.) operator is used by the union data type to access data elements.

e.g.: `d.x=5;`

`d.f=8.3;`

During accessing, we must ensure that we are accessing the member whose value is currently stored. Union can store only the last entered data, it overwrites the previously stored information. This is because the maximum memory allocated is the size of the largest data type of member variables.

Let us discuss the features of union with the following code,

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

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

```

union student
{
    int roll_no;
    char name[20];
}s;
void main()
{
    s.roll_no=101;
    printf( "Student Roll number : %d\n", s.roll_no);
    strcpy(s.name, "Sam");
    printf( "Student name : %s\n", s.name);
    printf("Size of union variable s is %d",sizeof(s));
    printf("\nRoll number after name assignment%d",s.roll_no);
}

```

### Output

Student Roll number : 101

Student name : Sam

Size of union variable s is 20

Roll number after name assignment <garbage value>

In the above program, the memory space of the roll number in the final print statement got replaced by the data element name, hence some garbage value gets printed. So, when a different member is assigned with a new value, the new value supersedes the previous allocated member's value.

Union members can be initialized only with a value of the same type as the first union member. For example, the following declaration is valid.

```
union student roll_no={12};
```

But the declaration, union student roll\_no={12.00} is invalid because the type of first member is int.

Example: Program to illustrate memory allocation for union members.

```
#include<stdio.h>

union u
{
    int i;
    char c[2];
};

void main()
{
    union u ul;
    ul.i=1024;
    printf("ul.i=%d\n",ul.i);
    printf("ul.c[0]=%d\n",ul.c[0]);
    printf("ul.c[1]=%d\n",ul.c[1]);
    ul.c[1]=5;
    printf("ul.c[0]=%d\n",ul.c[0]);
    printf("ul.c[1]=%d\n",ul.c[1]);
    printf("ul.i=%d\n",ul.i);
}
```

### Output

ul.i=1024

ul.c[0]=0

ul.c[1]=4

ul.c[0]=0

ul.c[1]=5

ul.i=1280

Representation of the above data is shown in Fig. 3.4.2. The union occupies 2 bytes in memory. Here, int has 2 bytes of memory. The same memory locations used for u1.c[0] and u1.c[1] are used by u1.i also. Initially, u1.i is assigned with an integer value 1024, a 2-byte number. The binary equivalent of 1024 is 0000 0100 0000 0000. From Fig 3.4.2, it is clear that the last eight bits are also occupied by u1.c[0] and the first eight are occupied by u1.c[1]. Therefore, u1.c[0]=0 and u1.c[1]= 4. Next, u1.c[1] is reassigned with value 5(binary equivalent- 0101). Therefore, the content of u1.i also changed to 0000 0101 0000 0000(equal to 1280).

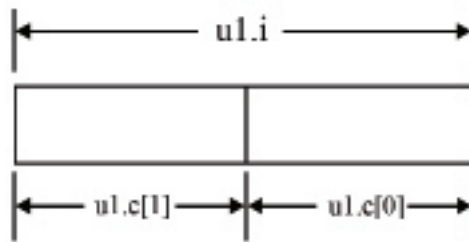


Fig. 3.4.2 . Union data in memory

The representation of structure for the above data is demonstrated in Fig 3.4.3

```
struct s
{
    int i;
    char c[2];
}s;
```

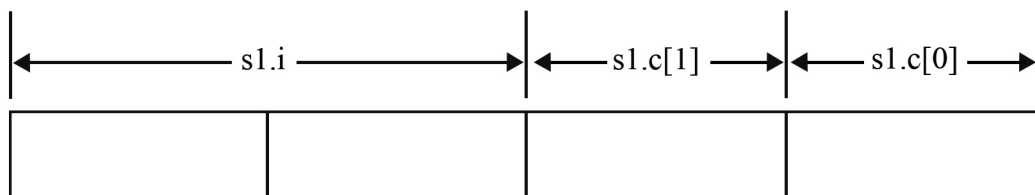


Fig. 3.4.3 Structure data in memory

It occupies 4 bytes in memory, 2 bytes for s.i and 2 bytes for s.c[0] and s.c[1].

### 3.4.12 Union of Structures

Similar to nested structures, union and structures can be implemented in nested form. The notation to access union and structure members which are in nested form remains

the same as for the nested structures. The following program is an example of structures nested in a union.

```
struct s
{
    int i;
    char c[2];
};

struct t
{
    int j;
    char d[2];
};

union u    //union with structure members
{
    struct s s1;
    struct t t1;
};
```

In the above program, two structure variables s1 and t1 are defined as members within a union. Assigning value for a struct variable also assigns a value for the next structure because both are variables of a union type.

```
union u ul;

ul.s1.i=5;
ul.t1.d[0]='w';
ul.t1.d[1]='r';

printf("\n%d",ul.s1.i);    //Output    5
printf("\n%d",ul.t1.j);    //Output    5
printf("\n%c",ul.t1.d[0]);    //Output    w
printf("\n%c",ul.t1.d[1]);    //Output    r
printf("\n%c",ul.s1.c[0]);    //Output    w
printf("\n%c",ul.s1.c[1]);    //Output    r
```

Fig. 3.4.4 shows the memory allocation of structure variables defined above(here int size is 2 bytes).

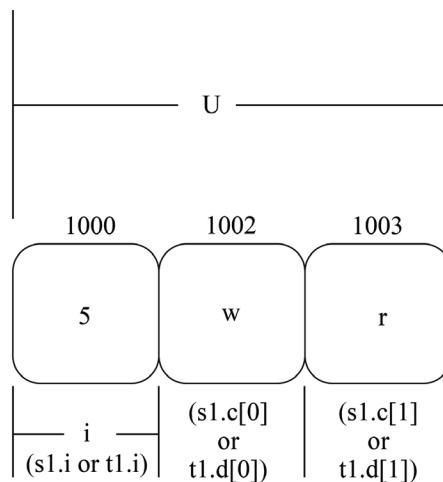


Fig. 3.4.4 Sharing of memory locations by structure variables defined in union

The following statement reassigns the location d[0] with c[0]. Memory allocation after this reassignment is shown in Fig. 3.4.5

```
u1.s1.c[0]='y';
printf("\n%c",u1.s1.c[0]);    //Output y
printf("\n%c",u1.t1.d[0]);    //Output y
```

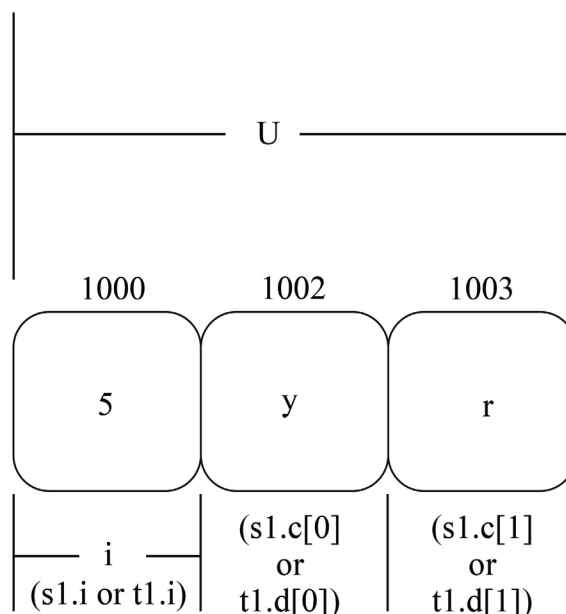


Fig. 3.4.5 Memory reallocation of structure variables defined in union



### 3.4.13 Pointers to Union

In C, pointers to union are similar to the concept of pointers in structure. It can also access the members using the arrow(->) operator.

The program code given below shows how a union type pointer variable is created and the base address of the union variable is assigned with union variable 'd'. *Arrow operator(->) is used with a pointer variable to access member variables of union.*

```
#include<stdio.h>

union dimen
{
    int x;
    float y;
};

void main()
{
    union dimen d;
    d.x=10;
    union dimen *p;
    p=&d;    // union variable base address is assigned to same type pointer
variable
    printf("%d %f",p->x,p->y);
}
```

#### Output

10 0.000000

### 3.4.14 Advantages of Union in application programs

Consider an application that stores information about employees in an organization. The information includes items like name, employee id, age, and division. Suppose division is classified into two- technical and non-technical.

```
if (division == technical)
```

```
    department
```

```
        credit card number
    if (division == non-technical)
        date of contract
        vehicle number
```

The representation of the above data in a structure is as follows

```
struct emp
{
    char name[2];
    int emp_id;
    int age;
    char div[15];
    int yoc;
    int ccn;
    char dept;
    char vn[10];
};
```

The structure definition meets all the requirements of an employee record, but there are some disadvantages. Depending on the division, each employee uses different data. All sets of fields would never be used. It would result in a wastage of memory with every structure variable because all structure variables would have all four fields. This disadvantage can be eliminated by using a union in place of structure.

```
struct div1
{
    int ccn;
    char dept;
};

struct div2
{
    int yoc;
    char vn[10];
};
```

```

union division
{
    struct div1 d1;
    struct div2 d2;
};

struct emp
{
    char name[2];
    int emp_id;
    int age;
    char div[15];
};

```

The above code gets rid of the disadvantage that we mentioned previously. It defines employee details in memory based on the requirement only and reduces wastage of memory.

### 3.4.15 Comparison between Structure and Union

No	Structure	Union
1	Keyword 'struct' is used	Keyword 'union' is used
2	Allocates memory for each element	Allocates memory on the size of an element with the largest size
3	Size of structure variable is equal to the size of the sum of the size of all data elements	Size of union is equal to the size of largest data element
4	Value assignment on one variable will not affect other variables	Value assignment on one variable changes content on other variables
5	Each element can be accessed at a time	Only one element can be accessed at a time
6	All elements can be initialized at once	Only one element can be initialized

### 3.4.16 More user-defined data types

#### 3.4.16.1 Enumerated Data Type

Enumerated data type is a user-defined data type. It **assigns names to integral constants** and the names are given to make the program easy to read and understand.

Keyword 'enum' is used to define and use an enumerated data type.

Syntax: enum identifier {value1, value2, ..., valuen};

The 'identifier' is a user-defined enumerated data type which is used to declare variables which are enclosed within the braces (*known as enumeration constants*).

e.g.: enum days {mon, tues, wed, thurs, fri, sat, sun};

After enum definition, we can declare variables of enum type as below:

**Syntax: enum identifier v1, v2...;**

e.g: enum days day1, day2;

Default values can also be assigned by the compiler beginning with 0 to all the enum constants. The default value of the enumerated data type starts from 0, and then increments continuously. It is also possible to assign values to defined constants explicitly.

***Assign Values to the Variables***

day1=mon;

day2=tues;

Let us discuss it in more detail

e.g.: **enum boolean {fal, tru};** //Default Values starts from 0

printf("\n Boolean values assigned are %d %d", fal,tru);

Boolean values assigned are                      0            1

e.g.:**enum day {sun=1,mon,tues,wed,thurs,fri,sat};**

//initial value assigned is 1

printf("\nThe value of enum day is %d\t %d\t %d\t %d\t %d\t %d\t %d\t",sun,mon,tues,wed,thurs,fri,sat);

enum day d1,d2;

d1=sun;

d2=mon;

printf("\n Value of d1 and d2 is %d,%d", d1,d2);

The constant sun is assigned with value 1. The remaining constants are assigned values that increase continuously by 1.

The value of enum day is      1          2          3          4          5          6          7

Value of d1 and d2 is 1, 2

e.g.: **enum name {sam,ram,jan=5,prin,hyran,quin=23,ann};**

//assigned with different values

```
printf("\nThe value of names given are %d\t %d\t %d\t %d\t %d\t %d\t %d\t",
sam,ram,jan,prin,hyran,quin,ann );
```

The value of names given are 0      1          5          6          7          23      24

In enum data type, values not given in the original declaration cannot be used. Size of the enum variable is the size of the integer. The definition and declaration of enum variables can be done in a single statement.

```
enum day{sun, mon,...., sat} day1, day2;
```

### Advantages of Enumerated data type

- ◆ Enumerated constants can be generated automatically.
- ◆ Errors can be easily detected.
- ◆ Programs become more readable and understandable.

### 3.4.16.2 Type definition

Type definition allows users to define an identifier that would represent an existing data type. Keyword 'typedef' can be used to define a new name to an existing data type.

#### Syntax – typedef data\_type identifier;

Where data\_type refers to an existing data type and identifier is the new name which is meaningful given to the data\_type. The existing data\_type can be of any type including the user-defined ones.

e.g: typedef int marks;

Now the identifier marks can be used as type int.

```
marks m1=5;
```

```
printf("%d",m1);
```

User-defined data types can also be defined using 'typedef'. For example, using typedef with structure, you can define a new data type and use that data type to define structure variables.

e.g.: struct student

```

{
    char name[10];
    int roll_no;
};
typedef struct student stud;
stud s;
or
typedef struct student
{
    char name[10];
    int roll_no;
}stud;
stud s;

```

### 3.4.16.3 Type Conversion

Through type conversion, variables of one type can be changed to another.

**Two types of type conversions are there**

- ◆ Implicit Type Conversion
- ◆ Explicit Type Conversion

***Implicit type conversions* are done by the compiler automatically.**

Consider the following program code,

```

int x=12,s;

char c='a';

s=x+c;

printf("Value after addition is %d",s);

```

In the above example, character type variable 'c' (a) is added with integer type variable 'x'(12) and the result is assigned to integer variable 's'. ASCII value of 'a' (97) is added. The compiler automatically converts character type to ASCII before the addition operation. Such conversions are called **implicit type conversions**.

## Output

Value after addition is 109

Type conversions **done by users explicitly** are called *explicit Type conversions*.

e.g.: - float x=8.3;

```
int a= (int)x+2;//explicit conversion
```

```
printf("Value of a is %d",a); // output 10
```

*Explicit type conversions* are also called *typecasting* because the user can cast a data type to another.

## Recap

- ◆ Structure is a user-defined data type.
- ◆ It is a collection of one or more variables, maybe of different data types, assembled under a single name.
- ◆ Structure variables are real-time entities and member variables are data associated with it.
- ◆ Direct handling of data is not possible in structures; therefore, special operators ( . and ->) are used to access data members.
- ◆ Concept of pointer to structure makes data handling efficient and simple.
- ◆ Array of structures allows us to store many data records of similar type.
- ◆ Array is a finite collection of similar variables.
- ◆ Nested structure is defining structure inside another structure.
- ◆ It is possible to pass and return structure variables as arguments to functions.
- ◆ Unions are derived data types; the way structure is.
- ◆ Union is used to collect a number of different types of items together.
- ◆ Union allows a method for a section of memory to be managed as a variable of one type on one occasion, and as a different variable, of different type on another occasion.
- ◆ Union performs better in terms of memory utilization compared to the 'Structure.'
- ◆ It is likely to define any built-in or user-defined data types inside a union based on the requirement.

- ◆ The size of a union variable at any instance is equal to the size of largest element among all the elements in the union.
- ◆ At any time, only one member of the union can occupy the memory.
- ◆ Enumerated data types consist of a set of named values called enumerators.
- ◆ Enum variables are usually identifiers that behave like integral constants.
- ◆ Explicit and implicit value assignment is possible.
- ◆ typedef create meaningful data type names which increases the readability of the program
- ◆ Typecasting is a technique by which you can change the data type of a variable, regardless of how it was originally defined.

## Objective Type Questions

1. struct student

```

{
    char name[10];
    int class;
    int roll_no;
};

void main()
{
    structure student.class=10;
    print("%d",student.class);
}

```

What will be the output?

2. State whether True or False. Structure elements are stored in contiguous memory locations



3. What is another term for user-defined data type?
4. How can you find out the size of a structure?
5. Function cannot be a structure member. State True/False
6. State True or False. It is possible to create an array of structure
7. What is the size of the given union?

```
union rect
{
    int x;
    char a[10];
};
```

8. #include<stdio.h>

```
union rect
{
    int x;
    int y;
};
```

```
void main()
{
    union rect r=30;
    printf(“%d %d”,r.x,r.y);
}
```

What will be the output?

9. What will be the output of the following code

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

```

union student
{
    int no=5 ;
    char name[20];
};

void main()
{
    union student u;

    u.no = 8;

    printf("hello");

}

```

10. Which element determines the size of union?

11. What is the output?

```

void main(){
    union {int i1; int i2;} myVar;

    myVar.i2 =100;

    printf("%d %d",myVar.i1, myVar.i2);

}

```

12. Consider the following statements

```

struct stu
{
    int a[5];

    union
    {

```

```

        float x,

        double y;

    }u;

}s;

```

Suppose that int, float, double occupy 2, 4, 8 bytes rep. What do you think about the memory requirement?

13. #include<stdio.h>

```

enum san

{

    a,b,c

};

enum san g;

void main()

{

    g++;

    printf(“%d”,g);

}

```

What will be the output?

14. What will be the output of the following code?

```

void main()

{

    typedef int a;

    a i=4,j=8,k;

    k=(i*2)/2+j;

```

```
printf(“%d”,k);

}
```

15. What is the size of array1 in the following code?

```
typedef char x[10];

x array1[4];
```

16. Which type conversion is called automatic type conversion?

17. What will be the output?

```
void main()

{

    int i = 10;

    char j = ‘m’;

    i = i + j;

    float k = i + 1.0;

    printf(“i = %d, k = %f”, i,k);

}
```

## Answers to Objective Type Questions

1. Compilation error (structure instead of struct) and structure variable not defined
2. True
3. Aggregate data type or derived data type
4. Sum of the size of all data elements
5. True
6. True
7. 10(Hint: Largest member variable)
8. Compile-time error (rect r = 30 - invalid initializer, should have been rect r.x={30})

9. Compile-time error (Hint :int no=5; )
10. size of the biggest member in the union
11. 100 100 (Hint: share same memory)
12. 18(Hint: 8 for y and 10 for a[5])
13. 1(Hint: Default value starts from 0)
14. 12(Hint:a is equivalent to int)
15. 40/(Hint:10\*4 )
16. Implicit type conversion
17. 119 120.000000 (Hint: Type casting)

## Assignments

1. Write a program to read details of N students (Name, Class, Marks of 5 subjects), calculate the percentage of marks and display the details.
2. Write a program to find the area of a rectangle using structures with pointers.
3. Write a program to implement a book with its associated properties using structure.
4. Write a program to implement employee record for a group of employees (Nested structure and array of structures)
5. Write a program to implement a library (book name, publisher, price, author, date of publishing) using an array of structures.
6. Discuss the advantages and disadvantages of structure.
7. Write a program to implement a student record using union. Discuss what are the changes noted in the same program using structure.
8. Write programs for employee records using both structure and union and compare the features.
9. Write a program to display months using the concept of enumerated data type.
10. Compare type conversion and type casting.
11. What is a structure? How to declare and define structure? How to create structure variables?
12. How to access elements of structures in C programming.
13. Create a structure Student with field student name, id number, course, address and date of birth. Write a C program to read and display students' details.
14. Compare array and structure.
15. Explain nested arrays with examples.

16. Create a structure Employee with following fields name, id, address and date of birth.
17. Create another structure within the employee, DateOfBirth with following fields date, month and year.
18. Explain pointers in structure .
19. How to pass structure in a function? Explain with an example.

## Reference

1. Balagurusamy, E. Programming in C. Tata McGraw-Hill Education, 2008.
2. Forouzan, Behrouz A. "Computer Science, A Structured Programming Approach Using C, Brooks." Cole. California, 2001.
3. Kernighan, Brian W., and Dennis M. Ritchie. The C programming language. Prentice Hall, Englewood Cliffs, N.J., 1988.
4. Gottfried, Byron S. Schaum's Outline of Programming in Structured Basic. McGraw-Hill Professional, 1992.

```
#include "KMotionDef.h"
```

```
int main()
```

```
{
```

```
    ch0->Amp = 250;
```

```
    ch0->output_mode=MICROSTEP_MODE;
```

```
    ch0->Vel=70.0f;
```

```
    ch0->Amp=500.0f;
```

```
    ch0->Jerk=0.0f;
```

```
    ch0->Limit=0.0f;
```

```
    EnableAxisDest(0,0);
```

```
    ch1->Amp = 50;
```

```
    ch1->output_mode=MICROSTEP_MODE;
```

```
    ch1->Vel=70.0f;
```

```
    ch1->Amp=500.0f;
```

```
    ch1->Jerk=0.0f;
```

```
    ch1->Limit=0.0f;
```

```
    EnableAxisDest(1,0);
```

```
    DefinedDays=0;
```

```
    return 0;
```

```
}
```

## BLOCK 4

# Storage Classes, Files, and Preprocessors



# Storage Classes

## Learning Outcomes

After the successful completion of the unit, the learner will be able to:

- ◆ understand the concepts of the scope of elements in a program
- ◆ learn different storage classes in C
- ◆ achieve skills to implement efficient programs using bitwise operators.

## Prerequisites

Generally, in the declaration of every variable, we need to specify the data type and variable name.

e.g.: `int x;`

In addition to data type and name, the above statement specifies the size of the variable, and the amount of memory allocated. These properties are applicable to functions also. Functions also have a data type, name, return type, etc. Actually, the properties of elements of a program do not limit to these.

For example, consider you are admitted to a college which is under a university. At the time of admission, you will definitely get an admission number and identity card. The identification card includes details like student name, class, course, admission number, college name, year of validity, etc. Using the identity card, you can access college facilities like a library, gymnasium, canteen, etc.

Is it possible for you to access the same facilities in another college under the same university using the same id for you?

Obviously, No. Why?

You have not been admitted to that college. The id card is allotted for the college that you are admitted to. Even if you are part of a university, your accessibility for the above-mentioned facilities is limited to your college. After all, the validity of the id card expires after years of study. You can't access the college facilities, even if you have the





```

}

void main()
{
    int n, result;

    printf("Enter number\n");
    scanf("%d",&n);
    result= n*n;
    printf("\nSquare of the number is %d", result);
}

```

When we compile the program, there will be an error as `n` is declared only in the main function and not in the square function

The scope of a variable is within that part of the program code in which it can be used. In the above program, the scope of variable '`n`' is within the main function because it is declared within that function. It is not possible to use that variable outside the function. This is known as local scope.

The concept of availability or accessibility of variables or functions are referred to as their scope and lifetime. Scope of an element in a program denotes a region in a program where an element can be accessed after its declaration. Scope defines the visibility of an identifier in a program. Visibility refers to the accessibility of a variable from the memory.

If the variable is declared before the main function and not within any other function, the scope of the variable is through the entire program. That is, the variables can be used at any place in the program. This scope is known as a global scope.

The life of a variable declared within a function ends with the last statement of the function. The variables used as formal arguments and variables declared within a function have local scope.

Similar to variables, functions also have scope. Functions can be used within the function where it is declared. Then the function is said to have a local scope. If it is declared before the main function and not within any other function, it has a global scope. A function which is declared inside the body of another function is called a local function. A function which is declared outside the body of any other function is called a global function.

Storage Classes define the scope and lifetime of a variable or a function. It helps to trace the existence of a variable during the runtime of a program.

## 4.1.2 Types of Storage classes

In C, four types of storage classes are there. They are

- ◆ Automatic
- ◆ Static
- ◆ Register
- ◆ External

### 4.1.2.1 Automatic

Automatic variables are declared inside a function in which they are to be used. Variables under the automatic storage class are local variables or internal variables. That is, the visibility and scope are limited within the block in which they are defined. These variables are created when the function is called and is deleted automatically when the function is exited. Keyword 'auto' is used to define it.

e.g.: `auto int age;`

Automatic storage class is the default storage class for all elements defined within a function or block, hence it is rarely used in C programs. By default, auto variables are initialized with some garbage value.

`int i=3, j;        // by default under category auto variable`

In the above statement, variable 'i' is initialized with value 3 and no initialization value for variable j. Hence it would be initialized with some garbage value. Memory allocated gets free after the completion of the execution of the corresponding block.

**Example :** Program to demonstrate 'auto' variables.

```
#include <stdio.h>

void display();
void display()
{
    int b;        //default auto variable
    printf("\n\n function value of b is \n%d",b);
}

void main()
{
```

```

    auto int a,b=5;    //Auto variables
    char c;            //default auto variables
    int d;              //default auto variables
    printf("%d\t%d\t%c\t%d",a,b,c,d);
    display();
}

```

In the above program, variable b is declared both in main() and display(). The values taken by variable b are different in these functions. In main(), it is initialized with value 5, and in display(), it is not initialized, hence a garbage value is assigned. The compiler treats variable b in two blocks as totally different variables since they are defined in different blocks. Variables a, c, and d are automatic variables and are initialized with some garbage value. Remember to initialize the automatic variables properly, otherwise, you will get unexpected results.

### Output

0      5      <garbage>      0

In function value of b is

22017

#### 4.1.2.2 Static

Static variables are single-valued local variables and they preserve the value throughout the entire program. These variables are declared once and exist at the end of the program. Keyword 'static' is used.

e.g.: static int yearofb=1990;

A static variable can be an internal or external type depending on the place of declaration. Internal static variables have scope till the end of the function in which it is defined. So it is similar to auto variables, but remains in existence throughout the program. Hence static internal variables can retain values between functions. An external static variable is declared outside all functions and is accessible to all the functions in the program. Default value of a static variable is 0. In its lifetime, it is initialized a single time.

**Example:** Program to demonstrate static variables.

```
#include<stdio.h>

void add ()
{
    static int a = 5;          //static variables
    int b = 11;                // auto variables
    printf ("%d %d \n", a, b);

    a++;
    b++;
}

void main ()
{
    int i;
    for (i = 0; i < 3; i++)
    {
        add ();
    }
}
```

Output

```
5      11
6      11
7      11
```

The above program consists of two functions main() and add(). The function add() gets called from main() three times. In add(), variable 'a' is static and variable 'b' is automatic. Static variable a is initialized to 5 and it is never initialized again. During the first function call, a is incremented to 6 (a is static and the value remains). For the next function call, the process repeats. In the static storage class, the statement static int a=1 is executed only once, irrespective of how many times the same function is called.

For auto variable b, when each time add() is called, it is re-initialized to 11. In effect, no matter how many times add() is called, b is initialized to 11 every time.

#### 4.1.2.3 Register

Register variables are the same as auto variables, but the difference is that these variables are usually stored in processor registers which makes faster accessibility. Since register access is much faster than memory access, the frequently accessed data kept in registers make the program execution faster. If a processor did not find available free register, then the content is stored in memory. Keyword 'register' is used.

e.g: register int pixel=2;

Default initial values of the register variable is 0. Scope and visibility of the variable are limited to the function or block.

```
#include <stdio.h>

void main ()
{
    register int a;           // The initial default value of a is 0.
    printf ("%d", a);
}
```

#### Output

0

A variable in the register storage class can be accessed faster than any other storage class. It is better to use a register storage class for variables used at many places in a program.(e.g: variables in the loop). The number of CPU registers is limited and may be reserved for some other task. Therefore, it cannot be sure that the value of a register variable would be stored in a CPU register.

#### 4.1.2.4. External

External Variables are declared outside of all functions which makes it access to all functions in the program. They are alive and active throughout the entire program. These variables are also known as global variables. Value change in an external variable gets reflected in all functions within the program and it can be changed in any block. Keyword 'extern' is used.

e.g.: extern average=45;



Default initial value of external variables is zero.

**Example:** Program to demonstrate extern variables

```
#include <stdio.h>

extern int a;           //extern variables

int a = 10;

void display ();

void main ()
{
    int a = 5;          //local variable
    printf ("In main function %d", a);
    display ();
}

void display ()
{
    printf ("\nIn display function %d", a);
}
```

In the above program, variable a is declared a global variable initially. Within main() function, it is again declared and is considered as local to that function. When function display() is trying to access variable a, control goes to global variable, not to the local variable. This is because local variable a is local to main() and is not accessible outside main().

### Output

In main function 5

In display function 10

Table 4.1.1 Comparison of various storage classes

Storage Class	Storage Specifier	Scope	Life	Initial Value
<b>Automatic</b>	Auto	Within block	End of block	Garbage
<b>Static</b>	Static	Within block	End of program	Zero
<b>Register</b>	Register	Within block	End of block	Garbage
<b>External</b>	Extern	Entire program	Entire program	Zero

## Recap

- ◆ A variable's storage class provides the following information:
  - Where the variable would be stored.
  - What will be the initial value assigned
  - Lifespan of the variable
  - Scope of the variable.
- ◆ Automatic storage class - visibility and scope are limited within the block in which variables are defined
- ◆ Static storage class - variables are declared once and exist till at the end of the program
- ◆ Register storage class - variables are usually stored in processor registers
- ◆ External storage class - declared outside of all functions which makes access to all functions in the program



## Objective Type Questions

1. What will be the output?

```
void main()

{

int x=10;

static int y=x;

if(x==y)

printf("\n Equal");

else

printf("\n Unequal");

}
```

2. State True or false. Static storage class cannot be used with function parameters.
3. What will be the output?

```
#include<stdio.h>

static int wt;

int num;

void main()

{

printf("%d\t",wt);

printf("%d",num);

}
```

4. What will be the output?

```

void main()
{
    register num1=7;
    {
        register num1=10;
        register num2=20;
        printf("%d\n %d\n",num1,num2);
    }
    printf("%d",num1);
}

```

5. What is the output of the following program?

```

#include <stdio.h>

void main()
{
    register int i = 10;
    int *ptr = &i;
    printf("%d", *ptr);
}

```

6. What is the scope of the extern class specifier?  
 7. What will be the output?

```

void main()
{

```

```

auto int x=10;

{

    auto int x=20;

    printf("%d\t",x);

}

printf("%d",x);

}

```

8. What will be the output?

```

void main()

{

    register x=10;

    printf("%d",&x);

}

```

## Answers to Objective Type Questions

1. Compile time error (Hint: static variables can hold only constant literals)
2. True
3. 0      0      (Hint: Default values)
4. 10      20      7
5. Compile-time error      (Hint: Address of register variable i requested)
6. Global Multiple files
7. 20      10
8. compile-time error

(Hint: register is part of the CPU and accessing it using & is not possible)

## Assignments

1. Write programs to add three numbers using different storage classes and discuss the changes noted during usage of different storage classes
2. Explain the differences between automatic, static, register, and external storage classes in C, providing examples for each.
3. Write a C program that demonstrates the usage of automatic variables and discusses their scope and lifetime within the program.
4. Develop a C program showcasing the usage of static variables and analyze how their values are preserved throughout the program's execution.
5. Create a C program illustrating the usage of register variables and discuss their benefits in terms of program optimization.
6. Write a C program demonstrating the usage of external variables and explain how they can be accessed across different functions within the program.
7. Explain the purpose and functionality of bitwise operators in C, including AND, OR, XOR, left shift, right shift, and NOT.
8. Develop a C program that implements bitwise operations such as AND, OR, and XOR on two given integers, providing the output for each operation

## Reference

1. Balagurusamy, E. Programming in C. Tata McGraw-Hill Education, 2008.
2. Forouzan, Behrouz A. "Computer Science, A Structured Programming Approach Using C, Brooks." Cole. California, 2001.
3. Kernighan, Brian W., and Dennis M. Ritchie. The C programming language. Prentice Hall, Englewood Cliffs, N.J., 1988.
4. Gottfried, Byron S. Schaum's Outline of Programming in Structured Basic. McGraw-Hill Professional, 1992.



## Managing Files

### Learning Outcomes

After the successful completion of the unit, the learner will be able to:

- ◆ understand various types of files
- ◆ learn file handling functions and their implementation
- ◆ achieve skills to manage files using C programming

### Prerequisites

In the previous units, we learned of programs implemented on the assumption that input to a program originates from the standard input device. Usually, it is not enough to just display the data on the screen.

Consider a program to read student details from the keyboard using the concept of structure. By using such a program, we can read and display the student details. Does the job get completed? What do you think? Obviously, No. In an academic institution, when we enter student details, then they should be stored in permanent storage space for future purposes. Similarly, many real-life problems involve a large amount of data and in such cases console I/O operations are insufficient. In such situations, it becomes necessary to store the data in a manner that can be seen later and exhibited either in part or whole.

How can data be read from a disk? How can data be stored on a disk?

A common solution is data on disk. It is a more flexible approach where data can be stored on the disk and read whenever needed. It employs the concept of files to store data. A file is a common storage unit in a computer. All of us are familiar with different types of files; for example, documents, worksheets, PowerPoints, etc. Is it possible to operate files through C programs? Have you heard anything about this? Does C language provide any direct input-output functions for this?

## Key Concepts

FILE, file pointer, fopen, fclose

## Discussion

### 4.2.1 Data Organization in Files

Files are a collection of data. It can store data permanently without data loss. Two types of files are there -Text files (character streams) and binary files (binary data). We are mostly concerned with character data. In text files, data are in human-readable form. It includes numbers, alphabets, and other special symbols. In binary files, data is stored in the form of a sequence of bytes.

Data organized in a file can be manipulated easily through C file operations and in-built functions.

### 4.2.2 File Operations

C supports a set of functions that have the ability to perform basic file operations, which are the following:

1. Creating a new file
2. Opening an existing file
3. Closing a file
4. Reading from a file
5. Writing to a file
6. Seeking specific location in a file

While storing data in a file, the following things should be specified about the file

- ◆ Filename
- ◆ Data Structure
- ◆ Purpose

The Filename is a string of characters that should be a valid file name for the Operating System. It usually contains two parts – a primary name and an extension (optional). For example, add.c, computer.docx, program, etc. Data structure of the file is defined as FILE which is the pointer type. All files should be declared as type FILE before use. While opening the file the purpose of opening the file should also be specified .

### 4.2.3 Creating and opening a file

Before doing any operation in a file, we must create it. To create a new file and to open an existing file fopen() function is used.



Syntax:        `FILE *fp;`  
              `fp= fopen("file1","mode");`

The initial statement declares a data type called FILE and a pointer variable fp to file type. File pointer can hold an address of a C structure which stores the type of file operation, memory location of current read and write operations. The second statement opens the file named file1 and assigns an identifier to fp. It also specifies the purpose or mode of opening the file. "mode" denotes access mode and can take one of the following values.

Various modes of opening files are shown in Table 4.2.1.

Table 4.2.1: Modes of opening files in C

Mode	Explanation
r	opens an existing text file for reading operation. If the file does not exist, the function returns a NULL
w	opens a text file for a write operation. If the file specified does not exist a new file is created with the same file name.
a	opens a text file to append data. If the file specified does not exist a new file is created with the same file name
r+	opens a text file for both read and write operations
w+	opens a text file for both read and write operation. If the file specified does not exist a new file is created with the same file name
a+	opens a text file for both read and write operation. If the file specified does not exist a new file is created with the same file name. Read operation will start from the beginning, but writing content will append to the existing content.

Filename and mode are strings; therefore they should enclose in double quotes. For binary files, instead of the above mentioned modes rb, wb, ab, rb+, wb+, ab+ are used respectively.

fopen() performs three important steps when a file is opened.

Initially, it searches for the file to be opened.

Then it loads the file from the disk into a buffer (a temporary storage place in memory).

Then the character pointer points to the corresponding character in the buffer.

#### 4.2.4 Closing a file

The file should be closed after completing its operation. `fclose()` function is used to close files. This ensures that all information associated with the file is flushed out from the buffers.

**Syntax :** `fclose(file_ptr);`

This statement would close the file associated with the FILE pointer `file_ptr`. The `fclose()` function returns a zero on success. If there is an error in closing the file EOF will be returned. `fclose()` function releases all memory used for the file.

#### 4.2.5 Reading and writing to a file

Once a file is opened, reading from or writing to a file is accomplished using standard I/O functions.

##### A. `fprintf()` and `fscanf()` functions

These functions are similar to `printf()` and `scanf()`, but for file-handling functions, a file pointer is required to handle files. `fprintf()` sends formatted output to the file stream.

**Syntax :**      `fprintf(fp, "control string",list);`  
                  `fscanf(fp, "control string",list);`

where 'fp' is a file pointer, control string contains input-output specifications for the items in the list, the list may contain variables, constants and strings.

e.g: `fscanf(fptr,"%s %d",name,roll_no);`

`fprintf(fptr,"Name %s Roll number%d", name,roll_no);`

Example: Program to read data from a file.

```
#include <stdio.h>

#include <stdlib.h>

void main()
{
    int num;

    FILE *fptr;           // file pointer

    if ((fptr = fopen("C:\\program.txt","r")) == NULL)

        /*Program exit if the file pointer returns null*/
}
```



```

    {

        printf("Error! opening file");

        exit(1);

    }

    fscanf(fp, "%d", &num); //reads data from file program.c

    printf("Value of n=%d", num);

    fclose(fp);

}

```

The program displays the contents of the file program.txt on the screen. Before a read or writing information, we must open the file. To open the file fopen() is called. File pointer fp points to the file program.txt. If the file opening fails, fopen() returns a NULL. Function call exit() terminates the execution of the program. Data is read using the function fscanf() from the file. While reading from a file, it should be careful to use the same format specifications. After completing the read operation, the file has to be closed to flush out the associated buffers. The program reads numbers from program.txt and displays it in the output window.

**Example2:** Program to write student details into a file.

```

#include <stdio.h>

#include<stdlib.h>

void main()

{

    char name[50];

    int marks, i, num;

    printf("Enter number of students: ");

    scanf("%d", &num);

    FILE *fp;

    fp = fopen("C:\\studentdetails.txt", "w");

```

```

        if(fptr == NULL)
        {
            printf("Error!");
            exit(1);
        }
        for(i = 0; i < num; ++i)
        {
            printf("For student%d\nEnter name: ", i+1);
            scanf("%s", name);          //reads data from keyboard
            printf("Enter marks: ");
            scanf("%d", &marks);
            fprintf(fptr, "\nName:%s\nMarks=%d\n", name, marks); //write
data to file.

        }

        fclose(fptr);
    }
}

```

The program read data from the keyboard using scanf function . If the file opening fails, fopen() returns a NULL. Function call exit() terminates the execution of the program. Data is written into the file using fprintf function. While reading from a file, it should be careful to use the same format specifications. After completing read operation the file have to be closed to flush out the associated buffers. The program reads name and marks from the keyboard and store it in order into the file.

### B. putc() and getc() functions

The functions getc() and putc() can handle one character at a time. fputc() writes a character to the file pointed to by the file pointer. If a file is opened in write mode with file pointer fptr, then the statement

**putc(c,fptr);**

Writes the character contained in the character variable c to the associated file.

Function getc() is used to read a character from a file that has been opened in read mode.

**c=getc(fp);**

The above statement would read a character from the file with file pointer fp. The getc() function will return EOF (End of File), when end of file has been reached. For both the functions, the file pointer moves by one character position.

**Example:** Write a program to read strings from the keyboard and write it into a file

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
    FILE *fp;
    char c;
    fp=fopen("program.c","w");//open file write mode
    while((c=getchar())!=EOF)
        //read character from keyboard
        putc(c,fp); //write character by character to file
    fclose(fp);
    fp=fopen("program.c","r");    //reopen the file

    while((c=getc(fp))!=EOF)
        //read character from keyboard
        printf("%c",c);    //printing on screen
    fclose(fp);
}
```

The above program enters the input data via the keyboard and writes it into the file, character by character. When EOF is reached, the file 'program.c' is closed. The file is again opened for reading. Then the contents are read, character by character, and displayed on the screen. Read operation terminates when getc encounters the EOF character.

### C. getw and putw functions

The getw and putw are similar to the getc and putc functions and are used to read and write integer values. When we deal with only integer data, these functions can be used. The format of getw and putw are following

Syntax:        putw(integer, fptr);  
                  getw(fptr);

### 4.2.6 Random Access to Files

There are situations when accessing only a particular part of a file. This can be achieved by the following functions.

#### fseek()

fseek() is used to move a file pointer to a specific position within a file.

**Syntax : fseek(file\_ptr, offset, position);**

where file\_ptr is the pointer to the file concerned, 'offset' is the number of bytes to offset from 'position', and 'position' denotes the position from where the offset is added. 'position' has three values as shown in Table 4.2.1.

Table 4.2.1: Value of position in fseek()

Value	Purpose
0	Starting of the file
1	Current position
2	End of file

e.g.: fseek(FILE \*fp, 10, 1);

The statement moves from the current position of the file pointer by moving 10 positions. When the operation is successful fseek returns zero, otherwise returns a non-zero value.

**Example:** Program to illustrate the execution of fseek()

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

```
void main ()
```

```

{
    FILE *fp;
    fp = fopen("file1.txt","w+");
    // Content of file
    fputs("This is structural design methodology", fp);
    //seeking 8th position from beginning
    fseek( fp, 8, 0 );
    // writes the content from 7th position
    fputs(" C Programming Language", fp);
    fclose(fp);
}

```

The above program opens the file file1.txt in read and write mode. The content of the file seeks the 8th position from the beginning and replaces the actual content with "C Programming Language". The content of the file will change to "This is C Programming Language."

### **ftell()**

ftell takes a file pointer and return a number that corresponds to the current position.

**Syntax : num =ftell(file\_ptr);**

Variable num gives the relative offset of the current position. The function returns the position as type long int which is an offset from the beginning of the file.

### **rewind()**

Function rewind() takes a file pointer and resets the position to the beginning of the file.

**Syntax: rewind(file\_ptr);**

num=ftell(file\_ptr);

The statements would assign value 0 to num because the file position has been set to the start of the file. Whenever a file is for input-output operations, a rewind is done implicitly.

## Recap

- ◆ A file is a group of related data stored in secondary storage.
- ◆ C supports a set of in-built functions to perform basic file operations.
- ◆ All files should be declared as type FILE before use.
- ◆ A file pointer is a pointer to the data type FILE and contains all the information about the file.
- ◆ File operations and mode of operations are key concepts in file management.
- ◆ Commonly used in-built file handling functions are fscanf(),getc(),getw() for input operations. Its counterpart functions are fprintf(),fputc() and putw() for output operations.
- ◆ Input-output operations in files can be done on a character basis, line basis or record basis.
- ◆ These operations are done using a buffer which improves efficiency.
- ◆ Buffers have limitations in size.

## Objective Type Questions

1. Which keyword is used to store a file pointer?
2. Give an example of a formatted file output function.
3. For truncation which mode is specified?
4. Which file mode is used for updating?
5. Why is it necessary to close a file during the execution of the program?
6. What is the data type of FILE in C?
7. Which component is appended to the mode string for binary files?
8. What is the value of EOF in C?
9. State True or False. Data of a file is stored on a hard disk.
10. In which form are data in text files stored?

## Answers to Objective Type Questions

1. FILE
2. fprintf
3. W
4. a (append mode or write mode)
5. Ensure all the buffers and all links to the file are broken.
6. opaque data type (Hint: Implementation is hidden)
7. b
8. -1
9. True
10. ASCII code

## Assignments

1. Develop a program to write employee's details into a file and display it on screen.
2. Write a program to append content from the last position of the file using fseek().
3. Write a program to write student details into a file and read from it.
4. Write a program to count the characters, spaces and newlines in a file.
5. Explain various input/output functions used in file operation.
6. What is EOF. Explain with an example.
7. Write a C program to create a new file and write "Hello, World!" into it.
8. Implement a C program to read the contents of a text file named "input.txt" and display it on the screen. Make sure to handle errors appropriately.
9. Write a C program to append additional text to an existing file named "data.txt" using file handling functions.
10. Develop a C program to read integers from a file named "numbers.txt", calculate their sum, and display the result on the screen.
11. Create a C program to count the occurrences of a specific character (e.g., 'a') in a text file named "textfile.txt". Display the count on the screen.
12. Implement a C program to copy the contents of one file into another file.
13. Write a C program to read student details (name, roll number, and marks) from the user and write them into a file named "student\_details.txt".
14. Develop a C program to read the contents of a file named "paragraph.txt", count the number of words, and display the count on the screen. Remember to handle errors gracefully.

## Reference

1. Balagurusamy, E. Programming in C. Tata McGraw-Hill Education, 2008.
2. Forouzan, Behrouz A. "Computer Science, A Structured Programming Approach Using C, Brooks." Cole. California, 2001.
3. Kernighan, Brian W., and Dennis M. Ritchie. The C programming language. Prentice Hall, Englewood Cliffs, N.J., 1988.
4. Gottfried, Byron S. Schaum's Outline of Programming in Structured Basic. McGraw-Hill Professional, 1992.





# Command-line Arguments

## Learning Outcomes

After the successful completion of the unit, the learner will be able to:

- ◆ understand the concepts of argument passing to main()
- ◆ learn argc and argv parameters in C
- ◆ achieve skills to implement efficient programs using Command-line arguments.

## Prerequisites

Arguments or parameters are used to pass values to a function. It is very useful for giving different values to a function.

The functions differ based on the number and type of parameters.

**Parameters are of two types:**

- ◆ Actual parameter
- ◆ Formal parameter

Actual parameter is the parameter which we passed during a function call. Formal parameters are parameters which are used in function definitions.

## Key Concepts

argc, argv, FILE, fopen

## Discussion

### 4.3.1 Command-line Arguments

We learned about functions, arguments, and their functionalities in previous units. In user-defined functions, users can include arguments within the program or through the console window. Have you ever thought about the possibility of passing arguments through the command line? Is it possible to control a C program from outside the program? The answer to the above questions is command-line arguments.

It is possible to pass values to the C program from the command line when the program is invoked. These values or arguments are called command-line arguments. Command-line arguments are also a method to control C programs from outside.

We know that the main function begins the execution of the program and it can also take arguments like other functions. The arguments that pass on to main() at the command prompt are called command-line arguments. Generally, the main() function has two arguments- argc and argv.

- ◆ argc is the number of arguments passed. 'argc' refers to argument count. It stores the number of arguments including the name of the program. That is, if a value is passed to a program, the value of argc should be 2.
- ◆ argv is an array of pointers to strings and points to each argument passed to the program. 'argv' refers to an argument vector. argv[0] is the name of the program. From argv[1] to argv[argc-1] represents each command line argument.

If we want to execute a program to copy the contents of a file PGM1 to another one PGM2, we can use a command line like

```
> PROGRAM PGM1 PGM2
```

where PROGRAM is the filename of the executable code where it is stored. Here, argc is three and argv is the array of three pointers to strings which are given below:

```
argv[0] -> PROGRAM
```

```
argv[1] -> PRG1
```

```
argv[2] -> PGM2
```

To access the command line arguments, we must declare the main function and its parameters as follows,

```
main(int argc, char *argv[])  
{  
    .....  
}
```

During execution, the strings on the command-line are passed to main(). To understand the working of command-line arguments, let us discuss it through a C program.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int count;
    printf("Program name is %s\n", argv[0]);
    if( argc == 2 )
        printf("The argument is -%s\n", argv[1]);
    else if( argc > 2 )
    {
        printf("More than two arguments supplied.\n");
        for(count=0;count<argc;count++)
            printf("argv[%d] - %s" , count, argv[count]);
    }
    else //argc=1
        printf("No arguments passed,only program name.\n");
}
```

The above program (program name - test) checks arguments which are supplied from the command line and take steps based on that.

If a single argument is supplied, it produces the following result.

```
>test test1
```

Program name is test

The argument is test1

When two arguments are supplied, the output will be like the following:

```
>test test1 test2
```

More than two arguments supplied

argv[0] - test

argv[1] - test1

argv[2] - test 2

When no argument is supplied, the output will be like following

>test

No arguments passed, only program name.

If no argument is supplied, argc will be one (program name itself). argv[1] points to the first command line argument and argv[n] points to last argument. argv[argc] is a Null pointer.

**Example :** Program to add two numbers using command-line arguments

```
#include <stdio.h>
#include<stdlib.h>
int main(int argc, char *argv[])
{
    int num1,num2,sum;
    num1 = atoi(argv[1]);
    num2= atoi(argv[2]);
    sum = num1+num2;
    printf("Sum of %d, %d is: %d\n",num1,num2,sum);
    return(0);
}
```

**Output**

> sum 4 7

Sum of 4, 7 is 11

Command line >sum 4 7, inputs three arguments - sum(program name), 4, and 7. The program finds the sum of the second and third arguments and displays it. atoi() is a library function that converts a string to an integer.

**Example :** Program to implement file copy programs using command-line arguments.

```
#include <stdio.h>

#include <stdlib.h>

void main ( int argc, char *argv[ ] )
{
    FILE *fsource, *ftarget ;
    char ch ;
    if ( argc != 3 )
    {
        puts ( "Error in number of arguments\n" ) ;
        exit ( 1 ) ;
    }
    fsource = fopen( argv[1], "r" ) ;
    if ( fsource == NULL )
    {
        puts ( "Error in source file handling\n" ) ;
        exit ( 2 ) ;
    }
    ftarget = fopen( argv[ 2 ], "w" ) ;
    if ( ftarget == NULL )
    {
        puts ( "Error in target file handling\n" ) ;
        fclose( fsource ) ;
        exit ( 3 ) ;
    }
    while ( 1 )
    {
        ch = fgetc( fsource ) ;
```

```
        if(ch == EOF)
            break ;
        else
            fputc (ch,ftarget) ;
    }
    fclose( fsource ) ;
    fclose( ftarget ) ;
}
```

### Output

```
> test test1.c test2.c
```

In the above program, argv[0] contains the base address of the test name test (program name), argv[1] contains the base address of test1.c and argv[2] contains the base address of 'test2.c'. The program opens the file test1.c in read mode, test2.c in write mode and copies the content from the first file to the second one.

## Recap

- ◆ Command line argument approach the parameter supplied through the command line.
- ◆ The strings supplied at the command line are stored in memory
- ◆ The address of the program name is stored in argv[0], the address of the second string is stored in argv[1] and so on
- ◆ argc is an argument counter and argv is an argument vector.
- ◆ main() can take arguments argc and argv and through this, the information can be passed on to the program.

## Objective Type Questions

1. What is the index of the last argument in the command-line argument?
2. State True or False. argv is an array of character pointers.
3. What will be the output?

```
int main(int argc, char *argv[])  
{  
    int result;  
    result= argv[1]+argv[2];  
    printf(“%d”,result);  
    return(0);  
}
```

4. What do argv[0] and argv[1] represent in command line arguments?
5. What is the second argument in the command line argument?

## Answers to Objective Type Questions

1. argc - 1(Hint: First argument is program name)
2. True
3. Error // string to number conversion needed.
4. filename and pointer to the first command line argument
5. A pointer to an array of character strings that contain the arguments

## Assignments

1. Write a program to find the largest among three numbers using command line arguments.
2. Write a program to concatenate strings entered using command line arguments.
3. Write a C program to calculate the factorial of a number provided as a command-line argument. Ensure error handling for invalid inputs.

4. Implement a C program that accepts two integers as command-line arguments and swaps their values. Display the swapped values on the screen.
5. Develop a C program to calculate the sum of all integers provided as command-line arguments. Handle scenarios where non-integer inputs are provided.
6. Create a C program that reads a text file specified as a command-line argument and counts the number of words, lines, and characters in it. Display the counts on the screen.
7. Write a C program to search for a specific word in a text file provided as a command-line argument. Display all occurrences of the word along with their line numbers.
8. Implement a C program to copy the contents of one text file to another text-file. Accept the file names as command-line arguments.
9. Develop a C program to perform basic arithmetic operations (addition, subtraction, multiplication, division) on two numbers provided as command-line arguments. Display the results.

## Reference

1. Balagurusamy, E. Programming in C. Tata McGraw-Hill Education, 2008.
2. Forouzan, Behrouz A. "Computer Science, A Structured Programming Approach Using C, Brooks." Cole. California, 2001.
3. Kernighan, Brian W., and Dennis M. Ritchie. The C programming language. Prentice Hall, Englewood Cliffs, N.J., 1988.
4. Gottfried, Byron S. Schaum's Outline of Programming in Structured Basic. McGraw-Hill Professional, 1992.





# Macros and Preprocessor Directives

## Learning Outcomes

After the successful completion of the unit, the learner will be able to:

- ◆ understand the concepts of preprocessor directives
- ◆ achieve skills to include preprocessor directives in a program efficiently
- ◆ understand the implementation of command-line arguments

## Prerequisites

Preprocessing is an essential step in any productive task. It prepares the data for analysis. Before we start actual processing, the data has to be preprocessed for clarity and better processing.

We are familiar with courier service. It allows you to send a parcel or letter from one location to another. Consider you booked for a product by giving your contact details and delivery address. The product you booked is the actual content and all others, like contact details, address, packing cover, etc, are aids to deliver the parcel correctly. When the product gets delivered, you will remove all unnecessary elements that we have mentioned above. It is an example of preprocessing an item.

Before the product was delivered, all the elements were necessary for proper management and delivery of the product. But after proper delivery, the elements have to be removed and the product is the ultimate component that you need. Likewise, preprocessing in C is exactly what its name implies. It is a program that processes the source code before compilation. In this unit, we are discussing how C program contents are preprocessed, which are the methods, and how it is implemented.

## Key Concepts

Macro, File inclusion, Conditional compilation

## Discussion

### 4.4.1 Concept of Preprocessor Directives

Preprocessor directives are one of the productive techniques implemented in the C language. The preprocessor is a program that processes the source program before it is passed to the compiler. As the name suggests, preprocessors preprocess the program before actual compilation, and preprocessor directives control the operation of preprocessors. Preprocessor directives are placed in the source program before the main function. The preprocessor examines the source code for any preprocessor directives before the compilation. If there is any, appropriate actions will be taken and the code is handed to the compiler.

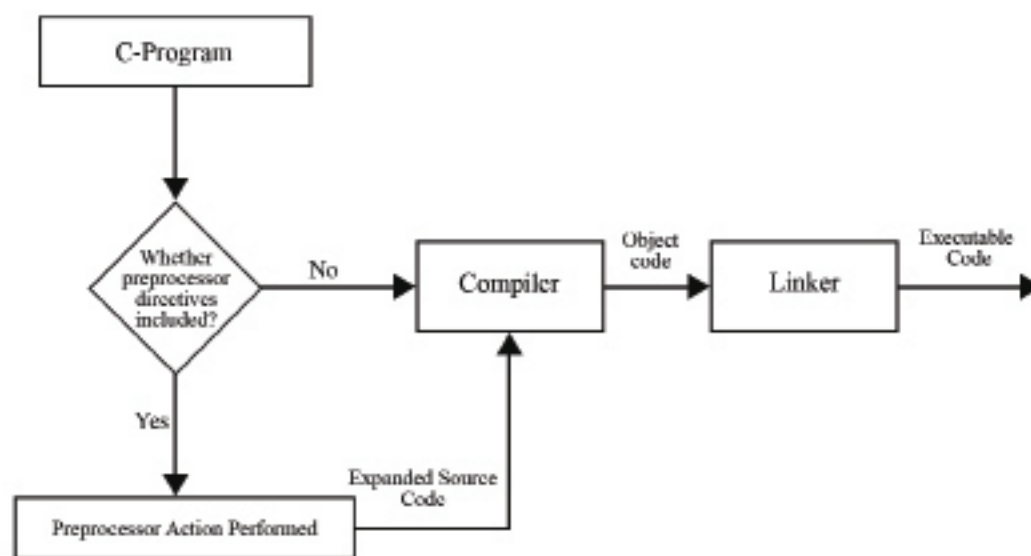


Fig 4.4.1 Build Process of C program

The combination of different stages of writing a C program is known as the build process (Fig 4.4.1). Before a C program is compiled, if necessary, it is passed through another program called a preprocessor. The preprocessor converts the source code to expanded source code.

Preprocessor directives follow some syntax rules that are different from normal syntax. It begins with the symbol # and does not require a semicolon at the end. For example, #include and #define are some directives we have already discussed. Remember, preprocessor directives can be placed anywhere in a program, but are generally placed at the beginning of a program. Let us look at some commonly used directives.

#### 4.4.1.1 Macros

Macros are pieces of code and have a specific name. Macro substitution is a process of replacing an identifier in a program with a predefined string composed of one or

more tokens. When the compiler encounters the identifier or macro name, the compiler replaces the name with the actual code.

‘#define’ is the directive that defines a macro. This statement is usually known as macro definition and is defined as follows:

Syntax : #define identifier value

e.g.: #define NUM 20

#define PI 3.1415

#define CAPITAL “TVM”

The statement replaces every occurrence of the identifier NUM in the source code by 20. It is a convention to write macros in capital letters to easily identify as symbolic constants. The following code demonstrates macro definition, substitution, and execution.

```
#include<stdio.h>
#define COUNT 5          //macro definition
void main()
{
    int i;
    for(i=0;i<COUNT;i++)
        printf("%d\n",i);
}
```

#### Output

0  
1  
2  
3  
4

In the above program, the variable count is defined before main(). That statement is called the macro or macro definition. The variable count is called the macro template and 5 is the corresponding macro expansion. In the preprocessing stage, the preprocessor substitutes every occurrence of count with 5.

Macros that we discussed above are called object-like macros. The identifier is replaced by a value. The following are some more examples of a macro being written.

#define directive can be used to define operators

ex: #define OR ||

#define directive can be used to replace a condition

```
#define con(a>20 OR a<50)
```

```
.....
```

```
if(con)
```

```
{
```

```
}
```

#define directive can be used to replace an entire C statement.

```
#define print printf("The result is unknown")
```

```
....
```

```
if(con)
```

```
print
```

Another type of macros is function-like macros and it looks like a function call. A macro with arguments is known as a macro call. When a macro is called, the preprocessor substitutes the string by replacing the formal parameters with actual ones. Consider the following code segment to understand the working of function-like macro.

```
#include<stdio.h>

//macro definition
#define max(num1,num2) ((num1>num2)?num1:num2)

void main()
{
    int a,b,m;
    printf("\nEnter two numbers ");
    scanf("%d %d",&a,&b);
    m=max(a,b); //control goes to macro definition
    printf("Maximum value is %d",m);
}
```

In the above program, whenever the preprocessor finds the function call `max(num1, num2)`, in main method and it expands to `((num1>num2)?num1:num2)`. When a macro call occurs, the preprocessor replaces the macro template with its macro expansion.

## Output

Enter two numbers 5 2

Maximum value is 5

Other special kind of macros are pre-defined macros. C compiler predefines some preprocessor macros. Let us discuss predefined macros with examples,

`_DATE_` represents current date

`_TIME_` represents current time

`_FILE_` represents current file name

```
printf("%s", _DATE_);
```

```
printf("%s", _TIME_);
```

```
printf("%s", _FILE_);
```

A macro can be undefined, using `#undef`.

## Syntax: `#undef identifier`

e.g.: `#undef NUM`

## Macros Vs Functions

Macros and functions look similar, but they are different in many aspects. Table 4.4.1 shows the differences between macros and functions.

Table 4.4.1 Macros vs Functions

Macros	Functions
The preprocessor replaces the macro template with the macro definition	Control passes to the function along with arguments, and after execution of function it returns the control to the position where it was called from
Runs faster	Slower compared to macros
Increases the program size	Smaller and more compact programs
Memory utilization is high	Less memory space requirements

### 4.4.2 File Inclusion

File inclusion directive causes an external file to be included as part of a program. It directs the compiler to include a file in the source program. Two types of files can be included – Header files and User-defined files.

Header Files – Different functions are defined in different files. For example, `printf()` and `scanf()` are included in the header file `stdio.h`. Similarly, `sqrt()` is defined in `math.h`. To include these functions in a program, corresponding header files should be added in the program.

**Syntax :** `#include(filename.h>`

e.g.: `#include<stdio.h>`

In this case, the specified file is searched only in the standard directories.

User-defined files – Larger programs can be divided into smaller ones and can be added to the program.

Syntax: `#include "filename"`

e.g.: `#include"prgm1.c"`

The preprocessor inserts the entire contents of the file specified into the source code of the program. The above statement would look for the file `program.c` in the current directory and the specified list of directories, and insert it.

### 4.4.3 Conditional Compilation Directives

As the name suggests, conditional compilation compiles a specific portion of a program or it can skip the compilation of a specific portion of a program, depending on the condition specified. It is also known as compiler control directives because it directs the compiler to skip certain parts of source code when they are not needed.

`#if`, `#else`, `#ifdef`, `#elif` and `#endif` are examples for conditional compilation preprocessing directives.

**1. `#ifdef` preprocessor directive** ensures a macro is defined in the header. If it is defined, the code between `#ifdef` and `#endif` executes in the program. The C processor also supports a more general form of `#if` directive.

**Syntax:**

```
#ifdef expression
{
    statements;
}
#endif
```

Consider the following example,

```

#include<stdio.h>
#define COUNT 0

void main()
{
    printf("Count is %d",COUNT);

    #ifdef COUNT
    {
        #undef COUNT

        int COUNT;

        printf("\nEnter value for count");
        scanf("%d",&COUNT);

        printf("Count is %d",COUNT);
    }
    #endif
}

```

This ensures that even if COUNT is defined in the program, its definition is removed by #ifdef.

### Output

Count is 0

Enter value for count 23

Count is 23

## 2. #elif, #else, #endif

#ifdef, #elif, #else, and #endif are similar to the usual if-else control instruction in C.

### Syntax:

#ifdef expression

{

```

        statements;
    }
    #elif expression
    {
        statements;
    }
    #else
    {
        statements;
    }
    #endif

```

### 3. #ifndef

#ifndef denotes if not defined. If it is included in the program, all the lines between the #ifndef and corresponding #endif directive are active in the program. Consider the following code

```

#include "DEFINE.H"

#ifndef NUM
#define NUM 1
#endif

```

Suppose DEFINE.H is the header file that defines NUM macro. The directive #ifndef NUM searches the definition of NUM in the header file and if not defined, then all the lines between the #ifndef and #endif directive are left active and define a value 1 to NUM. If NUM has been already defined in the header file, the #ifndef condition becomes false and #define statement is ignored.



## Recap

- ◆ Macro is a predefined code that replaces an identifier
- ◆ When a `#define` directive is noted, the preprocessor replaces the macro template with a macro definition.
- ◆ Preprocessor converts the source code to expanded source code and it passes to the compiler.
- ◆ In object-like macro, simple string replacement is used to define constants.
- ◆ In a function-like macro, the preprocessor substitutes the string by replacing formal parameters with actual parameters.
- ◆ To restrict a macro to a particular part of the program `#undef` is used.

## Objective Type Questions

1. If a number of instructions are repeated in the main program, how can you reduce the length of the program?
2. Preprocessor in C works on which file?
3. Which keyword is used to define macro?
4. What does `stdio` in `stdio.h` stands for?
5. How do you separate multiline macros?
6. What will be the output?

```
#include<stdio.h>

#define max 10

void main()
{
    #ifdef (max)
    Printf("Welcome");
    }
```

7. What will be the output?

```
#include<stdio.h>

void main()
```

```

{
    #ifndef count
    printf("Welcome");
    #endif
    printf("Enjoy");
}

```

8. Which symbol identifies a preprocessor directive in C?
9. State True or False. Macros increase program speed compared to functions

## Answers to Objective Type Questions

1. Using macros (Hint: macro replaces the repeating code)
2. .c file (Hint: Source file)
3. define
4. Standard input-output
5. using \ operator
6. Error (Hint: #endif missing)
7. Welcome Enjoy (Hint: count not defined, therefore both statements executed)
8. # symbol
9. True

## Assignments

1. Discuss the use of areas of all preprocessor directives and give code segments for each.
2. Discuss the role of preprocessor directives in C programming. How do they control the behavior of the preprocessor?
3. Write a C program to demonstrate the use of object-like macros. Define a macro for the value of pi and use it to calculate the area of a circle.
4. Implement a C program that utilizes function-like macros to perform arithmetic operations (addition, subtraction, multiplication, division) on two numbers entered by the user.

5. Create a C program that includes a user-defined header file and demonstrates the usage of macros defined within that header file. Provide a brief explanation of the header file's purpose.
6. Develop a C program that illustrates conditional compilation using `#ifdef` and `#ifndef` directives.
7. Include a macro definition and conditionally execute code based on whether the macro is defined or not.
8. Write a C program to read an integer from the user and use a macro to determine if it is even or odd. Display an appropriate message based on the result.
9. Implement a C program that includes multiple header files using `#include` directives. Ensure that each header file contains unique macros and demonstrate their usage in the main program

## Reference

1. Balagurusamy, E. Programming in C. Tata McGraw-Hill Education, 2008.
2. Forouzan, Behrouz A. "Computer Science, A Structured Programming Approach Using C, Brooks." Cole. California, 2001.
3. Kernighan, Brian W., and Dennis M. Ritchie. The C programming language. Prentice Hall, Englewood Cliffs, N.J., 1988.
4. Gottfried, Byron S. Schaum's Outline of Programming in Structured Basic. McGraw-Hill Professional, 1992.

# PROBLEM SOLVING AND PROGRAMMING IN C LAB MANUAL



## 1. Introduction

This lab manual has been designed to accompany your learning journey in the field of computer programming using C language which aligns with the academic needs of SGOU BCA learners. Whether you're taking your first steps into the world of coding or looking to strengthen your programming skills, this manual is customized to cater to the requirements of your academic curriculum.

## 2. Target Audience

Geared towards BCA learners, this manual is an essential companion for your coursework. Whether you are a beginner with minimal programming experience or a more advanced learner, the content is structured to meet the diverse needs of BCA learners.

## 3. Objective

The primary objective of this C Programming Lab Manual is to equip Bachelor of Computer Applications (BCA) learners at Sreenarayanaguru Open University with a robust foundation in C programming. The manual aims to achieve the following:

1. **Comprehensive Understanding:** Provide learners with a comprehensive understanding of the fundamental concepts, syntax, and principles of the C programming language.
2. **Hands-On Experience:** Facilitate hands-on learning through a series of structured lab exercises, enabling learners to apply theoretical knowledge to practical coding scenarios.
3. **Real-World Relevance:** Illustrate the real-world relevance of C programming by integrating examples and exercises that reflect applications in software development and problem-solving.
4. **Progressive Skill Development:** Foster a gradual progression of skills, starting from basic programming constructs and advancing to more complex topics, ensuring learners develop a well-rounded skill set in C programming.
5. **Preparation for Future Courses:** Lay a solid foundation for learners who may pursue more advanced programming courses, ensuring they are well-prepared for the challenges of subsequent coursework and professional endeavors.

By the end of this programming journey, BCA learners should feel confident in their ability to write, debug, and understand C code, setting the stage for success in both academic and practical applications within the field of computer science.



## 4 Instructions to Academic Counsellors

Each lab may consist of a conducting the experiment, post-test, and writing a report. The potential goals of labs are to enhance and deepen understanding of programming concepts, gain experience in working collaboratively in a team setting, develop problem-solving skills using C programming concepts. The academic counsellor should ensure that the lab learning outcomes are achieved.

Below are some guidelines and suggestions counsellor should consider:

- ◆ Lab time is limited, and most labs will take the entire time. Therefore, be punctual.
- ◆ Convey the learners about the importance of reading the lab manual and SLM in advance.
- ◆ At the beginning and end of the lab the counsellors should capture their biometric attendance with location and time using the designated app
- ◆ The counsellor should upload the attendance of the learners after two-third of total lab duration
- ◆ Lab sessions will be conducted during weekends.
- ◆ Duration of a lab session may vary between 3 to 6 hours.
- ◆ The counsellor should ensure the availability of systems and required software/tools prior to each lab session.
- ◆ Learners may work individually or in groups (maximum 3 members).
- ◆ The counsellor should tell the learners exactly what is expected of them in the lab.
- ◆ The counsellor should monitor student progress throughout the lab period.

## 5 Instructions to learners

- ◆ Learners should be regular and come prepared for the lab practice.
- ◆ In case learners miss a class, it is their responsibility to complete the missed experiment(s).
- ◆ learners should bring the lab manual
- ◆ They should implement the given program individually or in groups (maximum of 3).
- ◆ Learners should keep at least three copies of the stipulated format with them.

## 6 Hardware Requirements

Desktop/ Laptop computer

- ◆ Processor: 1GHz or faster with 2 or more cores on a compatible 64 bit processor
- ◆ RAM: 4GB or larger
- ◆ Storage: 64 GB or larger storage device

## 7 Software Requirements

- ◆ Linux Operating System with GCC / TURBO C in WINDOWS OS

## 8 About lab work

Steps involved in program development:- To develop the program in a high-level language and translate it into machine level language following steps has to be practiced.

### Step 1. Identify or analyze the problem

The first step is to understand the problem or requirement that the program is intended to solve. This involves gathering information from stakeholders and defining the scope of the project.

### Step 2. Designing

Once the requirements are clear, the next step is to design the program's architecture. This includes creating a high-level design that outlines the structure of the program and how different components will interact with each other, including writing algorithms and drawing flowcharts.

**Writing Algorithms:** An algorithm is a step-by-step procedure or set of instructions designed to solve a specific problem or accomplish a particular task. Writing algorithms involves breaking down the problem into smaller, manageable steps and describing the logical flow of operations to achieve the desired outcome. Algorithms can be written in natural language or using pseudocode, which is a simplified programming-like language.

**Algorithm:-** It is a method of representing the step by step process for solving a problem. Each step is called an instruction. Characteristics of algorithm are:

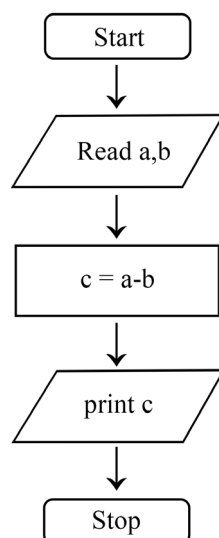
1. **Finiteness:-** It terminates with a finite number of steps.
2. **Definiteness:-** Each step of the algorithm is exactly defined.
3. **Effectiveness:-** All the operations used in the algorithm can be performed exactly in a fixed duration of time.

4. Input:- An algorithm must have an input before the execution of a program.
5. Output:- An algorithm has one or more outputs after the execution of the program.

Example of the algorithm to find the difference of two numbers:

- ◆ Step1: Start
- ◆ Step2: READ a, b
- ◆ Step3: Subtract a and b and store in variable c
- ◆ Step4: Print c
- ◆ Step5: STOP

**Drawing Flowcharts:** A flowchart is a graphical representation of an algorithm or process, showing the sequence of steps and decision points involved. Flowcharts use different shapes to represent different types of actions or decisions, such as rectangles for processes, diamonds for decisions, and arrows to indicate the flow of control. Drawing flowcharts helps visualize the logical flow of the program and understand how different components interact with each other.



**Step 3. Writing the program(Coding):** Coding, also known as programming, is the process of writing instructions for a computer to execute. These instructions, written in a programming language, tell the computer what tasks to perform and how to perform them. Coding is what allows developers to create software, applications, websites, and other digital tools that we use in our everyday lives.



#### **Step 4. Linking the program with the required library modules**

Linking a program with required library modules is an essential step in software development, especially when working with programming languages that support modularization and external libraries. By effectively linking a program with the required library modules, developers can leverage existing code and libraries to add functionality to their applications, speeding up development and improving code maintainability.

#### **Step 5. Compiling the program**

Compiling a program is the process of translating the source code written in a high-level programming language into machine-readable code, typically in the form of object code or executable files.

#### **Step 6. Executing the program**

Executing a program involves running the compiled executable file on a computer or device to perform the tasks it was designed for. Executing a program involves loading it into memory, running its main logic, handling errors, producing output, and interacting with users or other software components as needed to accomplish its tasks.

```
#include "KMotionDef.h"
```

```
int main()
```

```
{
```

```
    ch0->Amp = 250;
```

```
    ch0->output_mode=MICROSTEP_MODE;
```

```
    ch0->Vel=70.0f;
```

```
    ch0->Accel=500.0f;
```

```
    ch0->Jerk =2000f;
```

```
    ch0->Lead=0.0f;
```

```
    EnableAxisDest(1,0);
```

```
    ch1->Amp = 250;
```

```
    ch1->output_mode=MICROSTEP_MODE;
```

```
    ch1->Vel=70.0f;
```

```
    ch1->Accel=500.0f;
```

```
    ch1->Jerk =2000f;
```

```
    ch1->Lead=0.0f;
```

```
    EnableAxisDest(1,0);
```

```
    DefineCoordSystem(0,1,-1,-1);
```

```
    return 0;
```

```
}
```

# BLOCK 5

# PART A





## Familiarization of Input and Output Statements

### 5.1.1 Introduction

Input statements enable programs to receive data from users or external sources, while output statements allow programs to communicate results, messages, or information back to the user. This lab session is designed to provide you with hands-on experience in working with input and output statements.

### 5.1.2 Aim

- ◆ To study the use of Control Strings in C (\a, \t, \n,...etc.)
- ◆ To familiarize the format specifiers in C
- ◆ To prompt and input basic details of a person (Name, age, sex)
- ◆ To print various message outputs using various format specifiers

### 5.1.3 Sample Exercises

1. Write a C program to demonstrate the use of escape sequences, including \a (alert), \t (tab), and \n (newline). Print a message that includes these escape sequences.
2. Write a C program to display "Hello, World!" on the screen.
3. Create a program that takes two numbers as input and outputs their sum using format Strings.
4. Develop a C program that requests the learner's name as input and displays the entered name on the screen. Ensure the program is capable of handling complete names, even those containing spaces
5. Write a C program that prompts the user to input their birthday (day, month, and year). The program should then calculate and output the number of days remaining until their next birthday.

## 5.1.4 Problem

Capture bio-data of a person and display the details using appropriate formatting. Write a C program that prompts the user to enter their personal information, including their name, age, and favorite programming language. The program should then display this information with proper formatting using escape characters. Use the concepts of control strings, fundamental data types and basic input-output statements.

### Exercise 1: Use of Escape Sequences

Write a C program to demonstrate the use of escape sequences, including \a (alert), \t (tab), and \n (newline). Print a message that includes these escape sequences.

Test Case 1:

- ◆ Input: None
- ◆ Expected Output:

This is an example of alert:

This is an example of tab:   Tabbed Text

This is an example of newline:

New Line

### Exercise 2: Display "Hello, World!"

Write a C program to display "Hello, World!" on the screen using format Specifiers.

Test Case 2:

- ◆ Input: None
- ◆ Expected Output:

Hello, World!

### Exercise 3: Input Two Numbers and Display their Sum

Create a program that takes two numbers as input and outputs their sum using format Specifiers.

Test Case 3:

- ◆ Input:  
Enter the first number: 10  
Enter the second number: 20
- ◆ Expected Output:

The sum is: 30



### Exercise 4: Input Learner's Name and Display

Develop a C program that requests the learner's name as input and displays the entered name on the screen. Ensure the program is capable of handling complete names, even those containing spaces.

Test Case 4:

- ◆ Input:  
Enter your name: John Doe
- ◆ Expected Output:

Hello, John Doe!

### Exercise 5: Calculate Days Until Next Birthday

Write a C program that prompts the user to input their birthday (day, month, and year). The program should then calculate and output the number of days remaining until their next birthday.

Test Case 5:

- ◆ Input:  
Enter your birthday (day month year): 15 11 2000
- ◆ Expected Output (assuming current date is 1st January 2023):

Days until your next birthday: 319

### Exercise 6 : Capture Bio-data of a Person and Display Details

Write a C program that prompts the user to enter their personal information, including their name, age, and favorite programming language. The program should then display this information with proper formatting using escape characters. Use the concepts of control strings, fundamental data types, and basic input-output statements.

Test Case 1: Normal Input

- ◆ Input:  
Enter your name: Alice  
Enter your age: 25  
Enter your favorite programming language: C
- ◆ Expected Output:

-----  
Bio-data  
-----

Name: Alice Age: 25

Favorite Programming Language: C  
-----

### Test Case 2: Empty Name

- ◆ Input:  
Enter your name: Enter your age: 30  
Enter your favorite programming language: Python
- ◆ Expected Output:

#### Bio-data

Name: (No Name Provided) Age: 30

Favorite Programming Language: Python

### Test Case 3: Special Characters in Name

- ◆ Input:  
Enter your name: John@Doe Enter your age: 22  
Enter your favorite programming language: Java
- ◆ Expected Output:

#### Bio-data

Name: John@Doe Age: 22

Favorite Programming Language: Java

## APPENDIX

Input Statements	Output Statements
1. scanf(): scanf(format, &variable);	1. printf(): printf(format, variables);
2. fgets(): fgets(buffer, size, stdin);	2. puts(): puts("This is a string.");
3. getchar(): char ch = getchar();	3. putchar(): putchar('A');
4. getch() (Non-Standard, in some environments): char ch = getch();	4. fputs(): fputs("Hello, world!", stdout);
	5. fprintf(): fprintf(stdout, "Value: %d\n", value);
	6. putc(): putc('A', stdout);



## Variables and Datatypes : Familiarizing basic conversions

### 5.2.1 Introduction

In this lab experiment, we will explore the concept of variables and datatypes in the C programming language. Variables are containers that store data, and datatypes define the type of data a variable can hold.

### 5.2.2 Aim

- ◆ To familiarize and practice data conversions in C programming
- ◆ To prompt and input basic details in one unit and convert it to another unit of desired choice

### 5.2.3 Problem

Create C programs to perform data conversions and unit conversions. Include the following unit conversions:

1. Temperature Conversion
2. Distance Conversion
3. Time Conversion (year to days)
4. Time Conversion (seconds to hour) etc.

### 5.2.4 Exercises : Variables and Datatypes

#### Exercise 1: Variable Declaration and Initialization

Write a C program that declares and initializes variables of different datatypes (int, float, char). Print the values of these variables.

## Exercise 2: Arithmetic Operations

Write a C program that performs basic arithmetic operations (addition, subtraction, multiplication, division) using variables. Print the results.

## Exercise 3: Integer to Float Conversion

Convert an integer to a floating-point number and display the result.

Test Case 1:

- ◆ Input: 10
- ◆ Expected Output: 10.0

Test Case 2:

- ◆ Input: -5
- ◆ Expected Output: -5.0

## Exercise 4: Float to Integer Conversion

Convert a floating-point number to an integer and display the result.

Test Case 3:

- ◆ Input: 7.8
- ◆ Expected Output: 7

Test Case 4:

- ◆ Input: -3.5
- ◆ Expected Output: -3

## Exercise 5: Character to Integer Conversion

Convert a character to its corresponding integer value and display the result.

Test Case 5:

- ◆ Input: 'A'
- ◆ Expected Output: 65

Test Case 6:

- ◆ Input: '9'
- ◆ Expected Output: 57



## Exercise 6: Integer to Character Conversion

Convert an integer to its corresponding character and display the result.

Test Case 7:

- ◆ Input: 97
- ◆ Expected Output: 'a'

Test Case 8:

- ◆ Input: 74
- ◆ Expected Output: 'J'

## Exercise 7: String to Integer Conversion

Convert a string containing an integer to its numeric value and display the result.

Test Case 9:

- ◆ Input: "123"
- ◆ Expected Output: 123

Test Case 10:

- ◆ Input: "-45"
- ◆ Expected Output: 45

## Exercise 8: Unit Conversions - Celsius to Fahrenheit

Problem: Write a C program to convert temperature from Celsius to Fahrenheit. The formula for conversion is  $F = \frac{9}{5}C + 32$ , where F is the temperature in Fahrenheit and C is the temperature in Celsius.

Instructions:

1. Declare variables for Celsius and Fahrenheit.
2. Read the temperature in Celsius from the user use float or double.
3. Perform the conversion using the given formula.
4. Print the result.

Test Cases :Exercise 8

Test Case 1:

- ◆ Input: 25 (Celsius)
- ◆ Expected Output: 77.000000 (Fahrenheit)

Test Case 2:

- ◆ Input: 0 (Celsius)
- ◆ Expected Output: 32.000000 (Fahrenheit)

Test Case 3:

- ◆ Input: -10 (Celsius)
- ◆ Expected Output: 14.000000 (Fahrenheit)

## Exercise 9 : Unit Conversions

Create C programs to perform the unit conversion tasks outlined below:

### Temperature Conversion 1:

*Prompt the user to enter the current temperature in Celsius. Display the converted temperature.*

Enter the current temperature in Celsius: [user input]

Do you want to convert it to Fahrenheit (F) or Kelvin (K)? : [user input]

Converted temperature: [result]

### Test Case 1: Temperature Conversion

*Input:*

Enter the current temperature in Celsius: 25

Do you want to convert it to Fahrenheit (F) or Kelvin (K)? F

*Expected Output:*

Converted temperature: 77°F

### Temperature Conversion 2 :

*Ask the user to enter a temperature in degrees Celsius. Inquire whether they would like to convert it to Fahrenheit. Display the converted temperature.*

Enter the temperature in degrees Celsius: [user input]

Do you want to convert it to Fahrenheit (F)? [user input] Converted temperature: [result]

### Test Case 2: Temperature Conversion

*Input:*

Enter the temperature in degrees Celsius: -10

Do you want to convert it to Fahrenheit (F)? yes

*Expected Output:*

**Converted temperature: 14°F**

**Distance Conversion 1 :**

*Ask the user to input a distance in kilometers. Inquire whether they would like to convert it to miles or meters. Output the converted distance.*

Enter the distance in kilometers: [user input]

Converted distance: [result]

**Test Case 1: Distance Conversion**

*Input:*

Enter the distance in kilometers: 10

*Expected Output:*

**Converted distance: 6.2137 miles**

**Weight Conversion 1:**

*Prompt the user to enter their weight in kilograms. Ask if they prefer the weight in pounds or grams. Display the converted weight.*

Enter your weight in kilograms: [user input]

Converted weight: [result]

**Test Case 1: Weight Conversion**

*Input:*

Enter your weight in kilograms: 70

*Expected Output:*

**Converted weight: 154.3236 pounds**

**Length Conversion 1 :**

*Prompt the user to enter a length in meters. Ask if they would like to convert it to feet. Display the converted length.*

Enter the length in meters: [user input]

Converted length [result]

**Test Case 1 : Length Conversion**

*Input:*

Enter the length in meters: 5

Expected Output: **Converted length: 16.4042 feet**

## Exercise 10: Time Conversion 1

*Inquire about the duration in hours. Ask if the user wants to convert it to minutes or seconds. Output the converted time.*

Enter the duration in hours: [user input]

In minutes: [result]

### Test Case 1 : Time Conversion 1

*Input:*

Enter the duration in hours: 24

*Expected Output:*

**In minutes: 1440 minutes**

## Exercise 11: Time Conversion 2

*Prompt the user to input a duration in years. Ask if they want to convert it to days. Display the converted duration.*

Enter the duration in years: [user input]

Converted duration: [result]

### Test Case 2: Time Conversion 2

*Input:*

Enter the duration in years: 5

*Expected Output:*

**Converted duration: 1825 days**

## Exercise 12: Time Conversion 3

*Ask the user to input a time in seconds. Inquire whether they would like to convert it to hours. Display the converted time*

Enter the time in seconds: [user input]

In hours: [result]

### Test Case 3 : Time Conversion 3

*Input:*

Enter the time in seconds: 3600

*Expected Output:*

In hours: 1 hour



## Operators and Expressions : Age Calculator

### 5.3.1 Aim

- ◆ To familiarize learners with arithmetic operators and expressions in the C programming
- ◆ To familiarize conditional statements in C

### 5.3.2 Problem

1. Write a C program that demonstrates the use of arithmetic operators (+, -, \*, /, etc). Take two integer inputs from the user and perform various arithmetic operations on them. Print the results.
2. Write a C program that uses relational operators and logical operators to compare two numbers.
3. Write a C program that uses bitwise operators to manipulate bits of an integer. Take an integer input from the user, perform bitwise operations, and print the results.
4. Create an age calculator that displays the age of a person in years, months, and days. Use the concepts of arithmetic operators

### 5.3.3 Arithmetic Operators

Objective: Write a C program that demonstrates the use of arithmetic operators (+, -, \*, /, %). Take two integer inputs from the user and perform various arithmetic operations on them. Print the results.

**Test Case :**

*Input:*

Enter the first number: 15

Enter the second number: 7

*Expected Output:*

Sum: 22

Difference: 8

Product: 105

Quotient: 2.142857

Mode: 1

## 5.3.4 Relational and Logical Operators

Objective: Write a C program that uses relational operators and logical operators to compare two numbers.

**Test Case :**

*Input:*

Enter the first number: 10 Enter the second number: 20

*Expected Output:*

10 < 20 is true

10 > 20 is false

10 <= 20 is true

10 >= 20 is false

10 == 20 is false

10 != 20 is true

(10 > 0) && (20 < 30) is true

(10 > 0) || (20 < 5) is true

!(10 == 20) is true

## 5.3.5 Bitwise Operators

Objective: Write a C program that uses bitwise operators to manipulate bits of an integer. Take an integer input from the user, perform bitwise operations, and print the results.

**Test Case :**

*Input:*

Enter an integer: 12

*Expected Output:*

Bitwise AND with 3: 0

Bitwise OR with 5: 13

Bitwise XOR with 9: 5

Bitwise NOT: -13

Left shift by 2: 48

Right shift by 1: 6

## 5.3.6 Age Calculator

Objective: Create an age calculator that displays the age of a person in years, months, and days. Use the concepts of arithmetic operators

**Test Case :**

*Input:*

Enter the birth year: 1990 Enter the birth month: 5 Enter the birth day: 15

*Expected Output:*

Age: 32 years, 6 months, 15 days





## Control Structures: Familiarization of various Looping Statements

### 5.4.1 Introduction

Loops in C are used to execute a block of code repeatedly. The three main types of loops in C are for, while, and do-while loops.

### 5.4.2 Aim

To understand and become familiar with different looping statements in the C programming language, including for, while, and do-while loops. Also understanding the use of break and continue statements.

### 5.4.3 Problem

Enter any 10 numbers:

1. Find and display even numbers
2. Display Armstrong numbers
3. Prime numbers
4. Numbers included in the Fibonacci series.

### 5.4.4 Exercise Questions: Looping Statements

#### Exercise 1: For Loop

Write a C program using the for loop to find positive even numbers below a given number or from a list of given numbers.

### Sample Test Case :

*Input:*

*Given Number: 15*

*Output:*

Positive even numbers below 15: 2, 4, 6, 8, 10, 12, 14

## Exercise 2: While Loop

Write a C program using the while loop for finding prime numbers below a threshold number or from a list of given numbers.

### Sample Test Case :

*Input:*

*Threshold Number: 30*

*Output:*

Prime numbers below 30: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29

## Exercise 3: Do-While Loop

Write a C program to check whether the entered number is an Armstrong or not using do-while loop

### Sample Test Case :

*Input:*

*enter a number: 153*

*Output:*

153 is an Armstrong number

## Exercise 4: Using break Statement

Instructions:

1. Write a program that uses a for loop to iterate from 1 to n.
2. Inside the loop, prompt the user to enter a number at each iteration.
3. If the number is prime, terminate the loop using the break statement.
4. Display the sum of all numbers entered by the user till then.

### Sample Test Case :

*Input:*

*n = 8, User enters: 1, 4, 5 (prime numbers)*

*Output:*

Sum of numbers entered: 10

## Exercise 5: Using continue Statement

Instructions:

1. Write a program that uses a while loop to iterate from 1 to 5.
2. Inside the loop, prompt the user to enter a number at each iteration.
3. If the user enters an even number, skip the remaining code in the loop using the continue statement.
4. Display a message indicating whether the entered number is odd or even.

### Sample Test Case :

*Input:*

*User enters: 2, 3, 5, 8*

*Output:*

Odd number: 3, Odd number: 5

Skipped even number 2 and 8

## Exercise 6: Combining break and continue

Instructions:

1. Write a program that uses a do-while loop to repeatedly prompt the user to enter a positive integer.
2. If the user enters a negative number, use break to terminate the loop.
3. If the user enters an even number, use continue to skip the rest of the loop and prompt for the next input.
4. Display the sum of all positive, odd numbers entered by the user.

### Sample Test Case :

*Input:*

*User enters: 3, -1, 4, 7, -2*

*Output:*

Sum of positive, odd numbers: 10

## Exercise 7: Menu-Driven Loop

Instructions:

Apply break and continue in a menu-driven loop.

1. Create a menu with options for addition, subtraction, multiplication, division and exit.
2. Implement a switch statement inside a loop to perform the selected operation.
3. If the user selects the "exit" option, use break to terminate the loop.
4. Display the result of each operation and continue the loop until the user chooses to exit.

**Sample Test Case :**

*Input:*

*Enter two numbers: 5 3*

*Enter your choice 1. Addition 2. Subtraction 3. Multiplication 4. Division  
5. Exit : 3*

*Output:*

Multiplication of 5 and 3 is 15

## Exercise 8: Loop with Multiple Conditions

Instructions:

Use break and continue in a loop with multiple conditions.

1. Write a program that uses a for loop to iterate from 1 to 20.
2. Inside the loop, check if the current number is divisible by 3 or 5.
3. If true, use continue to skip the current iteration; otherwise, print the number.
4. If the number is greater than 20, use break to terminate the loop.
5. Display the numbers that satisfy the conditions.

**Sample Test Case :**

*Input:*

*No specific input required*

*Output:*

Numbers divisible by 3 or 5: 3, 5, 6, 9, 10, 12, 15, 18, 20





## Arrays Pointers and Recursion

### 5.5.1 Aim

The aim of this exercise is to familiarize learners with fundamental concepts of C programming and to explore the use of arrays, recursion, and pointers in C programming.

1. Explore basic array operations such as declaration, initialization, and accessing elements. Implement algorithms that involve arrays, such as searching and sorting.
2. Grasp the fundamental principles of recursion and its advantages. Implement recursive functions to solve problems like factorial calculation, Fibonacci series, and Tower of Hanoi.
3. Master the concept of pointers and their role in C programming. Implement pointer arithmetic and understand the relationship between pointers and arrays.

### 5.5.2 Problem

Design a C program exercises to introduce the concepts of recursion, arrays, pointers, and string manipulation. Users can progressively build their skills gradually and work on practical programming tasks.

### 5.5.3 Exercises for Familiarizing Arrays, Pointers, and String Manipulation

#### 5.5.3.1 Arrays

Exercise 1: Find the Largest Element

Write a C program to find the largest element in an array.

Exercise 2: Binary Search

Implement a binary search algorithm for a sorted array.

### 5.5.3.2 Pointers

Exercise 3: Swap Using Pointers

Create a program to swap two numbers using pointers.

Exercise 4: String Concatenation

Implement a function to concatenate two strings using pointers.

Exercise 5: Dynamic Memory Allocation

Explore dynamic memory allocation by creating a program that dynamically allocates memory for an array.

### 5.5.3.3 String Manipulation

Exercise 6: String Concatenation

Write a C program to concatenate two strings without using the standard library functions.

Exercise 7: String Reversal

Write a C program to reverse a given string.

### Test Cases

#### *Arrays*

#### **Test Case 1: Largest Element**

Test the program to find the largest element with the following array: `int array[ ] = 5, 2, 8, 12, 3;`

**Expected Output:** 12

#### **Test Case 2: Binary Search**

Verify the binary search algorithm with the array: `int array[] = 1, 2, 3, 4, 5, 6, 7, 8, 9;`

`int key = 6;`

**Expected Output:** Element 6 found at index 5

### 5.5.3.4 Pointers

#### **Test Case 1: Swap Using Pointers**

Test the pointer-based swapping program with `a = 3` and `b = 7`.

**Expected Output:** After swapping,

a = 7 and b = 3.

#### **Test Case 2: String Concatenation**

Verify the string concatenation program with the strings: char str1[] = "Hello, "; char str2[] = "world!";

**Expected Output:** "Hello, world!"

#### **Test Case 3: Dynamic Memory Allocation**

Validate the dynamic memory allocation program by allocating memory for an array of size 5.

**Expected Output:** Successfully allocate and deallocate memory for the array.

#### **Test Case 4: String Concatenation-1**

*Input:* "Hello", "World"

**Expected Output:** "HelloWorld"

#### **Test Case 5: String Concatenation-2**

*Input:* "Programming", " is fun"

**Expected Output:** "Programming is fun"

#### **Test Case 6: String Reversal-1**

*Input:* "abcdef"

**Expected Output:** "fedcba"

#### **Test Case 7: String Reversal-2**

*Input:* "hello"

**Expected Output:** "olleh"



## Structures and Unions

### 5.6.1 Introduction

Structures and Unions are powerful tools for organizing and manipulating data. They allow programmers to create custom data types that can hold multiple pieces of information in a single variable.

### 5.6.2 Aim

The aim of this experiment is to introduce learners to the concepts of structures and unions in C programming. Structures and unions allow the grouping of variables under a single name, enabling the creation of more complex data types.

### 5.6.3 Problem

Designing a program to manage information about employees in a company. Each employee has different attributes such as name, employee ID, and salary. Utilize structures and unions to organize and manipulate this data efficiently.

### 5.6.4 Exercises for Familiarizing Structures and Unions

#### Exercise 1: Employee Information Structure

Define a structure named Employee to store information about an employee, including the employee\_name, employee\_ID, and salary.

#### Exercise 2: Employee Union

Define a union named EmployeeDetail to represent additional details about an employee. This union should include fields for employee skills, certifications, and department.



### Exercise 3: Input and Output Functions

Write functions to input and display information for an employee using the defined structure and union.

#### Test Cases

##### Test Case 1: Structure Initialization Input:

Enter employee details: Name: John Doe

Employee ID: 12345

Salary: 50000

*Expected Output:*

##### Employee Information:

Name: John Doe

Employee ID: 12345

Salary: \$50000

##### Test Case 2: Union Initialization Input:

Enter additional employee details: Skills: C, Python

Certifications: Certified Developer Department: Software Development

*Expected Output:*

##### Additional Employee Details:

Skills: C, Python

Certifications: Certified Developer Department: Software Development

##### Test Case 3: Complete Employee Information Input:

Enter complete employee details: Name: Alice Smith

Employee ID: 67890

Salary: 60000 Skills: Java, SQL

Certifications: Certified Tester Department: Quality Assurance

*Expected Output:*

##### Employee Information:

Name: Alice Smith

Employee ID: 67890 Salary: \$60000

Additional Employee Details:

Skills: Java, SQL

Certifications: Certified Tester Department: Quality Assurance

## 5.6.5 Understanding Structures

Question 1: Define a structure named Student to store student details. Include members roll number, name, subject 1 marks, and subject 2 marks. Initialize a variable of this structure type with the details of a student and display its information.

### Test Case:

*Input:* Initialize a Student variable with roll number = 101, name = "John Doe", subject 1 marks = 85, and subject 2 marks = 90.

*Output:*

Student Details:

Roll Number: 101

Name: John Doe

Subject 1 Marks: 85

Subject 2 Marks: 90

Question 2: Create a structure named Employee to store employee details. Include members' emp id, name, position, and salary. Initialize an array of this structure type with details of employees and display the information of each employee.

### Test Case:

*Input:* Initialize an array of Employee structures with details of three employees: (1, "Alice", "Manager", 60000), (2, "Bob", "Developer", 50000), (3, "Charlie", "Analyst", 45000).

*Output:*

Employee Details:

Employee ID: 1, Name: Alice, Position: Manager, Salary: 60000

Employee ID: 2, Name: Bob, Position: Developer, Salary: 50000

Employee ID: 3, Name: Charlie, Position: Analyst, Salary: 45000



## 5.6.6 Nested Structures

Question: Create a structure Address to store address details with members street, city, and zip code. Define another structure Person to store personal information with members name, age, and address (of type Address). Initialize a variable of type Person and display its details.

### Test Case:

*Input:* Initialize a Person variable with name = "Eve", age = 25, and address with street = "123 Main St", city = "Cityville", and zip code = "12345".

*Output:*

Person Details:

Name: Eve

Age: 25

Address: 123 Main St, City: Cityville, Zip Code: 12345

## 5.6.7 Union with Enumeration

Question: Define an enumeration Color with values RED, GREEN, and BLUE. Create a union named Paint to represent painted items with members color (of type Color) and quantity. Initialize a variable of this union and display the color and quantity.

### Test Case:

*Input:* Initialize a Paint union variable with color = GREEN and quantity = 10.

*Output:*

Paint Details: Color: GREEN Quantity: 10

```
#include "KMotionDef.h"
```

```
int main()
```

```
{
```

```
    ch0->Amp = 250;
```

```
    ch0->output_mode=MICROSTEP_MODE;
```

```
    ch0->Vel=70.0f;
```

```
    ch0->Accel=500.0f;
```

```
    ch0->Jerk =2000f;
```

```
    ch0->Lead=0.0f;
```

```
    EnableAxisDest(0,0);
```

```
    ch1->Amp = 250;
```

```
    ch1->output_mode=MICROSTEP_MODE;
```

```
    ch1->Vel=70.0f;
```

```
    ch1->Accel=500.0f;
```

```
    ch1->Jerk =2000f;
```

```
    ch1->Lead=0.0f;
```

```
    EnableAxisDest(1,0);
```

```
    DefineCoordSystem(0,1,-1,-1);
```

```
    return 0;
```

```
}
```

# BLOCK 6

## PART B





## Functions : Recursion, Call by Value and Call by Reference, File Management

### Introduction

In this lab experiment, we will explore the concept of functions-recursion, call by value, call by reference and file management.

### 6.1 Aim

The objective of this experiment is to understand and practice modular programming in C using functions. This experiment will also introduce the basics of file management, focusing on reading and writing data to files.

### 6.2 Problem

Consider the problem of the age calculator. Implement the solution using functions with and without arguments. Also implement the concepts of Recursion, Call by value and Call by reference and File management in following exercises.

### 6.3 Exercises on Functions : With and Without Arguments

#### 6.3.1 Implementing Functions Without Arguments

Create a C program for the age calculator without using function arguments.

##### Requirements:

1. Implement a function to get the birth year from the user.
2. Implement a function to get the current year from the user.
3. Implement a function to calculate and display the age using the obtained years.
4. In the main function, call the functions in the appropriate sequence.

**Test Case 1: Calculate age in years.**

*Input:*

Birth Year: 1990

Current Year: 2023

*Expected Output:*

**Your age is: 33 years**

**Test Case 2: Verify the calculation with different years.**

*Input:*

Birth Year: 1985

Current Year: 2023

*Expected Output:*

**Your age is: 38 years**

## 6.3.2 Implementing Functions With Arguments

Create a C program for the age calculator using function arguments.

**Requirements:**

1. Implement a function that takes birth year and current year as arguments.
2. Inside the function, calculate and display the age.
3. In the main function, get the birth year and current year from the user, and then call the function with these values.

**Test Case 3: Calculate age in years and months.**

*Input:*

Birth Year: 1995

Current Year: 2023

Birth Month: 6

Current Month: 9

*Expected Output:*

**Your age is: 28 years and 3 months**

**Test Case 4: Verify the calculation with different months.**

*Input:*

Birth Year: 1988



Current Year: 2023

Birth Month: 3

Current Month: 11

**Expected Output:**

Your age is: 35 years and 8 months

### 6.3.3 Enhancing the Age Calculator With Additional Information

Extend the age calculator program with more functionalities.

**Requirements:**

1. Modify the functions to not only calculate the age in years but also in months and days.
2. Implement separate functions for getting the birth month and current month.
3. Implement separate functions for getting the birth day and current day.
4. Display the calculated age in years, months, and days.
5. Update the current date, based on that the output need to be printed.

**Test Case 5: Calculate age in years, months, and days.**

*Input:*

Birth Year: 1992

Current Year: 2023

Birth Month: 9

Current Month: 1

Birth Day: 15

Current Day: 7

*Expected Output:*

Your age is: 30 years, 3 months, and 23 days

**Test Case 6: Verify the calculation with different days.**

*Input:*

Birth Year: 1980

Current Year: 2023

Birth Month: 2

Current Month: 6

Birth Day: 28

Current Day: 15

*Expected Output:*

Your age is: 43 years, 3 months, and 17 days

## 6.3.4 Modularizing the Code

Modularize the age calculator program by separating each function into a dedicated file.

### Requirements:

1. Create separate header files (.h) for each function.
2. Implement the functions in corresponding source files (.c).
3. Use the modularized functions in the main program to calculate and display the age.

## 6.3.5 Error Handling and Validation

Implement error handling and validation for user inputs.

### Requirements:

1. Ensure that the user inputs are valid integers and handle non-numeric inputs appropriately.
2. Check for logical errors, such as the current year being before the birth year, and display meaningful error messages.

## 6.4 Exercises on Functions : Recursion, Call by Value and Call by Reference, and File Management

### 6.4.1 Recursion

#### Exercise 1: Recursion

Write a C program that demonstrates the use of recursion. Create a recursive function to calculate the factorial of a given number.

#### Test Case 1: Factorial Calculation

*Input:* User inputs 5.

*Expected Output:*

The program displays the factorial of 5, i.e., 120.

#### Test Case 2: Factorial Calculation

*Input:* User inputs 0.





*Expected Output:*

The program displays the factorial of 0, i.e., 1.

## 6.4.2 Fibonacci Series

Implement a recursive solution to find the nth term of the Fibonacci series.

### Test Case 3: Fibonacci Series Calculation

*Input:* Verify the recursive Fibonacci series function for n = 8.

*Expected Output:*

The program displays the 8th term of the Fibonacci series, i.e., 21.

## 6.4.3 Tower of Hanoi

Solve the Tower of Hanoi problem using recursion.

### Test Case 4: Tower of Hanoi

*Input:* Validate the Tower of Hanoi solution with n = 3 disks.

*Expected Output:*

Follow the steps to move all disks to the destination.

## 6.5 Exercise Questions: Functions and File Management

### 6.5.1 Call by Value and Call by Reference

Write a C program that demonstrates the concepts of call by value and call by reference in functions. Create functions that swap two numbers using both methods.

#### Test Case 1: Call by Value Swap

*Input:* User inputs in two variables, a=10 and b=7.

*Expected Output:*

The program displays the swapped numbers using call by value.

The value of a and b inside the function are a = 7, b = 10

The value of a and b inside the main function are a = 10, b = 7

#### Test Case 2: Call by Reference Swap

*Input:* User inputs in two variables, a=10 and b=7.

*Expected Output:*

The program displays the swapped numbers using call by reference.

The value of a and b inside the function are a = 7, b = 10

The value of a and b inside the main function are a = 7, b = 10

## 6.5.2 File Reading and Display

Write a C program that includes a function to read the contents of a text file named "input.txt" and display them on the console. The function should take the filename as a parameter. Additionally, in the main program, prompt the user to enter the name of the file to read and call the function to display its contents.

### Test Case 1: File Reading and Display

*Input:* Enter the filename: input.txt

*Expected Output:*

Contents of input.txt displayed on the console.

## 6.5.3 File Writing with Functions

Create a C program that utilizes functions to perform file operations. Write a function that accepts a filename and a string as parameters and appends the string to the end of the file. In the main program, prompt the user to enter a filename and a string. Call the function to append the string to the specified file. Make sure to handle cases where the file does not exist, and create the file if needed.

### Test Case 1: File Writing with Functions

*Input:* Enter the filename: output.txt

Enter the string to append: Hello, World!

*Expected Output:*

String Hello, World! appended to output.txt.

## 6.5.4 File Copy Operation

Write a C program that includes a function to copy the contents of one file to another. The function should take two filenames as parameters: the source file to read from and the destination file to write to. In the main program, prompt the user to enter the names of the source and destination files and call the function to copy the contents.

### Test Case 1 : File Copy Operation

*Input:* Enter the source filename: source.txt

Enter the destination filename: destination.txt

*Expected Output:*

Contents of source.txt copied to destination.txt.

## 6.5.5 File Operation

Extend the previous program to include file management. Implement functions that read data from a file, process it, and write the result back to another file.

### Test Case 1: File Operation - Sum Calculation

*Input:* Data stored in "input.txt" - 15 and 20.

*Expected Output:*

The program reads the data, calculates the sum (35), and writes the result to "output.txt".

### Test Case 2: File Operation - Product Calculation

*Input:* Data stored in "input.txt" - 8 and 12.

*Expected Output:*

The program reads the data, calculates the product (96), and writes the result to "output.txt".

# MODEL QUESTION PAPER SETS





MODEL QUESTION PAPER- SET- I

**SREENARAYANAGURU OPEN UNIVERSITY**

QP CODE: .....

Reg. No : .....

Name : .....

**End Semester Examination 2024**  
**BACHELOR OF COMPUTER APPLICATIONS**  
**B21CA02DC**  
**PROBLEM SOLVING AND PROGRAMMING IN C**

**Time: 3 Hours**

**Max Marks: 70**

**Section A**

**Answer any ten questions. Each carries one mark**

1. Define an algorithm.
2. What is a compiler?
3. What are tokens in C?
4. What will be the value of 'a' in the statement `int a = 10 + 4.867; ?`
5. What is the purpose of a loader?
6. Which header file includes `scanf()` and `printf()` functions?
7. Give an example of an exit-controlled loop.
8. What is an array?
9. What is a pointer?
10. What will happen if an exit condition is absent in a recursive function?
11. What is the default return type of function definition?
12. What is the output of the following C program?

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

```
int main() {
```

```
int a = 20;
```

```
printf("Good ");
```

```
return 1;
```

```
printf("morning");
```



```
return 1;
}
```

13. What are the formal parameters of a function?
14. What do you mean by the lifespan of a variable?
15. What is a file pointer?

**(10 x 1 = 10 Marks)**

## **Section B**

**Answer any five questions. Each carries two marks**

16. What are the two types of problem-solving approaches?
17. Draw a flowchart for swapping two numbers
18. List the rules for creating identifiers in C programs?
19. What is the output of the given program?

```
#include<stdio.h>

int main(int argc, char const *argv[])
{
    int num1 = 46;
    int num2 = 4;
    float num3 = 3;
    printf("\nResult1 = %d", (num1 / num2) );
    printf("\nResult2 = %f ", (num1 / num3) );
    printf("\nResult3 = %f ", ((float)num1 / num2) );
    return 0;
}
```

20. Write a program in C to read 10 numbers.
21. Consider the following program. Suppose the address of variables i, j, and k be 5000, 6000, and 7000 respectively. Then what will be the output of the program?

```
# include <stdio.h>

int main( )
{
    int i = 3, *j, **k ;
    j = &i ;
```



```

k = &j ;
printf ( "Value of j = %u\n ", j ) ;
printf ( "Value of k = %u\n ", k ) ;
printf ( "Value of i = %d\n ", i ) ;
return 0 ;
}

```

22. What is the output of the following code?

```

int main()
{
    int a = 430;
    char *b = (char *)&a;
    *++b = 2;
    printf("%d ",a);
    return 0;
}

```

23. What is modular programming?

24. List any two advantages of recursion.

25. Write a program to find the largest number in an array by using a function?

**(5 x 2 = 10 Marks)**

## Section C

**Answer any five questions. Each carries four marks**

26. Compare machine language, assembly language and high-level language.
27. Briefly explain various symbols and their function in a flowchart.
28. Briefly explain the precedence and associativity of operators in C with an example.
29. Write a program in C to check whether a given string is a palindrome or not.
30. Write a C program to read 'n' strings and find the string lengths using pointers.
31. Illustrate with an example, the usage of the following string functions in C.
  - a.     strlen( )

- b. strcpy( )
- c. strcat( )
- d. strcmp( )

- 32. Compare built-in functions with user-defined functions.
- 33. What are the different types of recursion?
- 34. List the differences between structure and union.
- 35. How can we create a new file in C? Illustrate with an example.

**(4 x 5 = 20 Marks)**

### **Section D**

**Answer any two questions. Each carries fifteen marks**

- 36. Describe the basic structure of a C program with an example.
- 37. Explain different loops in C? Describe each with syntax and examples.
- 38. What are functions? How do we declare, define and call a function in C? Illustrate the usage of functions in C with an example.
- 39. What are storage classes? What are the different storage classes available in C? Illustrate the working of different storage classes in C with examples.

**(2 x 15 = 30 Marks)**





MODEL QUESTION PAPER- SET- II

**SREENARAYANAGURU OPEN UNIVERSITY**

QP CODE: .....

Reg. No : .....

Name : .....

**End Semester Examination 2024**  
**BACHELOR OF COMPUTER APPLICATIONS**  
**B21CA02DC**  
**PROBLEM SOLVING AND PROGRAMMING IN C**

**Time: 3 Hours**

**Max Marks: 70**

**Section A**

**Answer any ten questions. Each carries one mark**

1. Define a flowchart.
2. What will happen when you remove the semicolon from the end of a statement?
3. Write the syntax of the conditional operator.
4. What is the purpose of the linker?
5. Which command is used to exit a loop in C?
6. Give an example of an entry-controlled loop.
7. What are strings?
8. What is the maximum number of arguments that can be passed in a single function?
9. Which keyword is used to send output obtained in a function to a called function?
10. What are the actual parameters of a function?
11. What is an enumerated data type in C?
12. What do `argv[0]` and `argv[1]` represent in command line arguments?
13. What is the use of an `#undef` preprocessor directive?
14. List any four built-in functions in C.
15. What do you mean by the lifespan of a variable?

**(10 x 1 = 10 Marks)**



## Section B

**Answer any five questions. Each carries two marks**

16. What are the various steps involved in problem-solving?

17. What is the output of the given code snippet?

```
int main()
{
    int a=0;
    a = 10 + 5 * 2 * 8 / 2 + 4;
    printf("%d", a);
    return 0;
}
```

18. Illustrate the usage of getchar() and gets() functions with an example.

19. Write a program in C to check whether a given number is even or odd.

20. What is recursion?

21. List any two disadvantages of recursion.

22. What is a preprocessor directive?

23. What is the purpose of the #ifdef preprocessor directive?

24. What is the purpose of ftell() function?

25. Write the syntax for function declaration, definition and function call with parameters.

**(5 x 2 = 10 Marks)**

## Section C

**Answer any five questions. Each carries four marks**

26. Write an algorithm and draw a flowchart to print all the prime numbers between 1 and 100.

27. What are C tokens? Explain various tokens in C with examples.

28. Write a C program to read 'n' characters from the user and display the characters using getchar() and putchar() functions.

29. Explain any three decision-making statements in C with syntax.

30. Give an example of nested functions in C.

31. List the differences between recursion and iteration.



32. List the various file opening modes in C and their purpose.
33. Illustrate the usage of `putc()` and `getc()` functions in C with an example.
34. Write a program to find factorial of a number using functions
35. How can we create a new file in C? Illustrate with an example.

**(4 x 5 = 20 Marks)**

### **Section D**

**Answer any two questions. Each carries fifteen marks**

36. Explain various operators in C and their uses with examples.
37. What is dynamic memory allocation in C. Explain the usage of the following functions with an example?
  - a. `malloc()`
  - b. `calloc()`
  - c. `free()`
  - d. `realloc()`
38. What is a structure? Illustrate the usage of structure in C with an example.  
(7 marks)
39. What are command line arguments? Illustrate the working of command line arguments in C with examples.

**(2 x 15 = 30 Marks)**

## സർവ്വകലാശാലാഗീതം

വിദ്യായാൽ സ്വതന്ത്രരാകണം  
വിശ്വപൗരരായി മാറണം  
ഗ്രഹപ്രസാദമായ് വിളങ്ങണം  
ഗുരുപ്രകാശമേ നയിക്കണേ

കുതിരുട്ടിൽ നിന്നു ഞങ്ങളെ  
സൂര്യവീഥിയിൽ തെളിക്കണം  
സ്നേഹദീപ്തിയായ് വിളങ്ങണം  
നീതിവൈജയന്തി പാറണം

ശാസ്ത്രവ്യാപ്തിയെന്നുമേകണം  
ജാതിഭേദമാകെ മാറണം  
ബോധരശ്മിയിൽ തിളങ്ങുവാൻ  
ജ്ഞാനകേന്ദ്രമേ ജ്വലിക്കണേ

കുരിപ്പുഴ ശ്രീകുമാർ

# SREENARAYANAGURU OPEN UNIVERSITY

## Regional Centres

### Kozhikode

Govt. Arts and Science College  
Meenchantha, Kozhikode,  
Kerala, Pin: 673002  
Ph: 04952920228  
email: rckdirector@sgou.ac.in

### Thalassery

Govt. Brennen College  
Dharmadam, Thalassery,  
Kannur, Pin: 670106  
Ph: 04902990494  
email: rctdirector@sgou.ac.in

### Tripunithura

Govt. College  
Tripunithura, Ernakulam,  
Kerala, Pin: 682301  
Ph: 04842927436  
email: rcedirector@sgou.ac.in

### Pattambi

Sree Neelakanta Govt. Sanskrit College  
Pattambi, Palakkad,  
Kerala, Pin: 679303  
Ph: 04662912009  
email: rcpdirector@sgou.ac.in

# PROBLEM SOLVING AND PROGRAMMING IN C

COURSE CODE: B21CA02DC



YouTube



Sreenarayanaguru Open University

Kollam, Kerala Pin- 691601, email: [info@sgou.ac.in](mailto:info@sgou.ac.in), [www.sgou.ac.in](http://www.sgou.ac.in) Ph: +91 474 2966841

ISBN 978-81-970547-0-9



9 788197 054709