Course Code: - CSM-6115
Course Name: - Programming with C

MASTER OF COMPUTER APPLICATIONS (MCA)

PROGRAMME DESIGN COMMITTEE

Prof. Masood Parveez
Vice Chancellor – Chairman
MTSOU, Tripura

Prof. Abdul Wadood Siddiqui
Dean Academics
MTSOU, Tripura

Prof. C.R.K. Murty
Professor of Distance Education
IGNOU, New Delhi

Prof. Mohd. Nafees Ahmad Ansari
Director of Distance Education
Aligarh Muslim University, Aligarh

Prof. P.V. Suresh
Professor of Computer Science
IGNOU, New Delhi

Prof. V.V. Subrahmanyam
Professor of Computer Science
IGNOU, New Delhi

Prof. S. Nagakishore Bhavanam
Professor of Computer Science
Mangalayatan University, Jabalpur

Prof. Jawed Wasim
Professor of Computer Science
Mangalayatan University, Aligarh

COURSE WRITERS

Dr. Md. Amir Khusru Akhtar
Associate Professor of Computer Science
MTSOU, Tripura
CSM-6111 Data Communication & Computer Networks

Dr. Ankur Kumar
Assistant Professor
MTSOU, Tripura
CSM-6112 Computer Organization & Architecture

Dr. Manish Saxena
Assistant Professor of Computer Science
MTSOU, Tripura
CSM-6113 Discrete Mathematics

Dr. Duvvuri B. K. Kamesh
Assistant Professor of Computer Science
MTSOU, Tripura
CSM-6114 Accountancy and Financial Management

Mr. Pankaj Kumar
Assistant Professor of Computer Science
Mangalayatan University, Aligarh
CSM-6151 Programming with 'C' & Lab

Ms. Vanshika Singh
Assistant Professor of English
MTSOU, Tripura
ENM-6101 Professional Communication

COURSE EDITORS

Prof. S. Nagakishore Bhavanam
Professor of Computer Science
Mangalayatan University, Jabalpur

Prof. Jawed Wasim
Professor of Computer Science
Mangalayatan University, Aligarh

Dr. Manoj Varshney
Associate Professor of Computer Science
MTSOU, Tripura

Dr. M. P. Mishra
Associate Professor of Computer Science
IGNOU, New Delhi

Dr. Akshay Kumar
Associate Professor of Computer Science
IGNOU, New Delhi

FORMAT EDITORS

Dr. Nitendra Singh
Associate Professor of English
MTSOU, Tripura

Ms. Angela Fatima Mirza
Assistant Professor of English
MTSOU, Tripura

Dr. Faizan
Assistant Professor of English
MTSOU, Tripura

Ms. Vanshika Singh
Assistant Professor of English
MTSOU, Tripura

MATERIAL PRODUCTION

1. Mr. Himanshu Saxena
2. Ms. Rainu Verma
3. Mr. Jeetendra Kumar
4. Mr. Khiresh Sharma
5. Mr. Ankur Kumar Sharma
6. Mr. Pankaj Kumar

CONTENT

Page No.

Block I: Algorithmic Process & Basic of 'C' Programming **5-115**

Unit–1: Algorithms, General Approaches & Analysis, Program and Programming Language, Fundamental Stages of Problem Solving, Features of Programming Language, Flow Charts.

Unit–2: Learning outcomes, Program and Programming Language, Introduction to C Language, Programming Format of C.

Unit–3: Creating a C Program, Compilation process in C Program, Link and Running C Program, Diagrammatic Illustration.

Block II: Operator and Expressions of 'C', Control Flow Mechanisms **116-214**

Unit–4: Building Blocks – Character set of C, C Tokens, Keywords, and Identifiers of the C. Fundamental elements of 'C' – Data Types in C, Variables.

Unit–5: Logical and Relational Operators in 'C', Expressions in 'C', and Type Conversions in Expressions.

Unit–6: Key Terminologies, Design Control Statements, Loop Control Statements, and Exit Function.

Unit–7: Declaring & Accessing Data Elements, Arrays Declaration, Initialization, and Passing Functions.

Block III: Strings, Tools for Modular Programming and Pointers **215-288**

Unit–8: Essential Techniques & Functions, Declaration and Initialization of Strings, Overview and Applications.

Unit–9: Functions, Prototypes, Calling a Function, Return Statement, Sets of Variables & Storage Classes, and Recursion.

Unit–10: Handle Variables and Parameters, Pointer and their Characteristics, Passing Pointers to Functions, and Pointers and Strings.

Block IV: Multiple Data Elements, Pre-processors Directives, and Files 289-365

Unit–11: Declaration of Structures, Accessing the Members of a Structure, Initializing, Function Arguments, and Pointers to Structures.

Unit–12: Defining Unions, Initializing Unions, and Accessing the Members of a Union.

Unit–13: Translation Phase, 'C' Pre-processor, Implement Constants, Reading from other files, and Conditional Selection of code and Pre-Processor Commands.

Unit–14: File Handling in C using file Pointers, Input and Output using file Pointers, Sequential Vs Random Access Files and Unbuffered I/O – The UNIX File Routines.

BLOCK I: ALGORITHMIC PROCESS & BASIC OF 'C' PROGRAMMING

UNIT 1 ALGORITHMIC PROCESS & BASICS OF C PROGRAMMING

Basic Structure

1.0 Introduction

1.1 Objectives

1.2 Learning outcomes

1.3 Program and Programming Language

1.3.1 Early Developments of Programming

1.3.2 Programming Elements

1.3.3 Design and Implementations

1.3.4 Uses and Properties

1.4 Introduction to C Language

1.4.1 History of C language

1.4.2 The Structure of C Program

1.4.3 Features of C language

1.4.4 C Program installation

1.4.5 Tools and Uses

1.5 Programming Format of C

1.5.1 Basic format and functions

1.5.2 A simple C programming

1.6 Creating a C Program

1.7 Compilation process in C Program

1.7.1 The Compiler

1.7.2 Semantic and Syntax Errors

1.8 Link and Running C Program

1.8.1 Running Program through Menu

1.8.2 Running Executable File

1.8.3 Linker Errors

1.8.4 Logical and Runtime Errors

1.9 Diagrammatic Illustration
1.10 Conclusion
1.11 Unit based Questions /Answers

1.0 INTRODUCTION

This unit introduces you to problem-solving ideas, particularly as they apply to computer programming. In this section, we introduce C, a standardized, industrial value programming language recognized for its power and portability as an implementation vehicle for these computer-based problem-solving methodologies. C's characteristics are designed to accurately match the capabilities of the targeted CPUs. It has a long history of usage in operating systems and device drivers, and protocol stacks, but its use in application software is declining. C is frequently used on computer architectures ranging from supercomputers to microcontrollers and embedded devices.

Language is a part of the method for two people is to be communicate with one another. To communicate, both parties must understand the language. Even if the two people do not speak the same language, a translator can help translate one language into another that the second person understands. A computer language, like a translator, allows a user and a machine to communicate. One version of the computer language is understood by the user, while the other is understood by the machine. A translator (or compiler) is necessary to convert from user form to computer form. A computer language, like any other, has a grammar known as syntax.

It is an imperative procedural language with a static type system

that supports structured programming, lexical variable scoping, and recursion. It was meant to be constructed in such a way that it provides low-level memory access and language features that translate effectively to machine instructions while requiring little runtime support. Despite its low-level capabilities, the language was meant to promote cross-platform development. With little changes to its source code, a standards-compliant C program developed with portability in mind may be generated for a broad range of computer platforms and operating systems.
This unit offers a concise introduction to programming, with an emphasis on understanding the concept of a computer.

A computer is an electronic device that receives, processes, and responds to instructions provided by users. It operates under the control of a computer program, which guides it in handling and manipulating data. A program consists of a set of instructions and data, often developed with the goal of building user-friendly applications, such as mobile apps. This unit is tailored for learners who have an interest in programming and aim to pursue a career in the field.

This unit will explain to you the fundamentals of the programming language C.

1.1 OBJECTIVES

After completing this unit, you will be able to:

- Gain comprehensive knowledge of the C programming language.
- Develop logical thinking skills essential for creating

programs and applications using C.

- Understand fundamental programming concepts, enabling an easier transition to other programming languages in the future.
- Build a strong foundation in the core principles of C programming.
- Apply the learned concepts through practical components designed to provide hands-on experience.

1.2 LEARNING OUTCOMES

Advantages of understanding the concepts from the unit are:

- Understanding a functional hierarchical code organization.
- Ability to define and manage data structures based on problem subject domain.
- Ability to work with textual information, characters and strings.
- Ability to work with arrays of complex objects.
- Understanding a concept of object thinking within the framework of functional model.
- Understanding a concept of functional hierarchical code organization.
- Understanding a defensive programming concept. Ability to handle possible errors during program execution.

1.3 PROGRAM AND PROGRAMMING LANGUAGE

A programming language is a formal language that provides instructions for computers to perform specific tasks. It is used for developing software, applications, and controlling computer

systems. Popular programming languages include Java, Python, C++, JavaScript, and C#. The chosen language depends on the platform, target audience, and desired output. Programming languages evolve and change over time, with new ones established and old ones improved to meet evolving demands. It provides a notation system for creating computer programs which is frequently defined by its syntax and semantics. They are usually specified using a formal language. A language usually has at least one application, like a compiler or interpreter, that enables programs written in the language to run. Programming language theory is a study of the structure, execution, evaluation, description, and classification of programming languages. The C programming language source code for a small computer program.

Programming is a talent that is growing in demand in the employment market. Anyone who works with technology should have at least a fundamental grasp of how software works. You can obtain a career coding, developing software, data architecture, or building user interfaces if you have a programming background.

While there are many of methods to categorize programming languages, they typically fall into five basic groups:

(i) **Procedural Programming Language (PPL):** Procedural language employs a series of statements or instructions to get the desired result. A procedure is a set of steps; therefore a program developed in one of these languages will have one or more procedures. Examples of procedural language includes: C, C++, Java, Pascal and BASIC.

(ii) **Functional Programming Languages (FPL):** Rather than focusing on statement execution, functional languages concentrate on the output of mathematical functions and

evaluations. Each function--a reusable code module--completes a defined job and produces a result. The outcome depends on the data you enter into the function. Among the most prominent functional programming languages are: Scala, Erlang, Haskell, Elixir and F#.

(iii) **Object-Oriented Programming Languages (OOP):** This language views a program as a collection of objects made up of data and program components known as attributes and methods. Objects can be reused both within and outside of a program. Because code is easy to reuse and scale, it is a common language type for complicated applications. Among the most common object-oriented languages are: PHP, C#, Ruby, Java and Python.

(iv) **Script-Oriented Languages (SOL):** Scripting languages are used by programmers to automate tedious operations, manage dynamic online content, and support processes in bigger applications. Among the most popular scripting languages are: PHP, Ruby, bash, Perl, Python and Node.js.

(v) **Logic Programming Languages (LPL):** A logic programming language communicates a collection of facts and rules to advise a computer on how to make decisions rather than telling it what to do. Logic languages include the following: Prolog, Absys, Datalog and Alma-O.

1.3.1 Early Developments of Programming

Early computers, like the Colossus, were programmed without a stored program by altering their circuitry or configuring physical

controllers. Later, programs could be written in machine language, where the programmer puts each command in a numeric format that the hardware can directly execute. These programs were read in decimal or binary form from various sources, such as punched cards, paper tape, magnetic tape, or switches on the computer's front panel. First-generation programming languages (1GL) were later coined to refer to machine languages. Second-generation programming languages (2GL) or assembly languages were created, which were tightly linked to the instruction set architecture of the individual machine. These languages made software more human-readable and reduced time-consuming and error-prone address computations. In the 1950s, the first high-level programming languages (3GL) were developed. Plankalkül, an early high-level programming language for computers, was created by Konrad Zuse between 1943 and 1945 for the German Z3. Short Code, suggested by John Mauchly in 1949, was one of the earliest high-level languages ever devised for electronic computers. Short Code statements, as opposed to machine code, represented mathematical formulas in a comprehensible format. However, every time the program was performed, it had to be translated into machine code, which made the process significantly slower than executing the comparable machine code. Alick Glennie created Autocode at the University of Manchester in the early 1950s. As a programming language, it employs a compiler to translate the language into machine code automatically. The original code and compiler were created in 1952 at the University of Manchester for the Mark 1 computer, and it is regarded as the first compiled high-level programming language. In 1954, R. A. Brooker created the "Mark 1 Autocode" for the Mark 1.

Brooker collaborated with the University of Manchester in the 1950s to design an auto code for the Ferranti Mercury. D. F.

Hartley of the University of Cambridge Mathematical Laboratory created the EDSAC 2 Autocode in 1961, a direct derivation from Mercury Autocode. Atlas Autocode was created for the University of Manchester Atlas 1 computer. In 1954, John Backus invented FORTRAN, the first widely used high-level general-purpose programming language with a functional implementation. It remains a popular high-performance computing language, with programs benchmarking and ranking supercomputers. Grace Hopper designed FLOW-MATIC, an early programming language, for the UNIVAC I between 1955 and 1959. Hopper and her team created a specification for an English programming language and constructed a prototype in 1955. The FLOW-MATIC compiler was made public early in 1958 and was almost complete in 1959. FLOW-MATIC had a significant impact on the architecture of COBOL because only it as well as its immediate descendent AIMACO were in service at that time.

1.3.2 Programming Elements

Every programming language has certain basic building fragments for describing data and the operations or transformations that are done to it (such as adding two integers or selecting an item from a collection). These primitives are specified via syntactic and semantic principles which demonstrate their structure and meaning accordingly.

(i) **Syntax:** The surface form of a computer language can be identified as its syntax. Several programming languages are basically textual; they employ textual sequences of words, numbers, as well as punctuation, similar to written natural languages. Some programming languages, alternatively, tend to be graphical in nature, relying on visual interactions between symbols

to express a program.

The syntax of a language specifies the many symbol combinations that can be used to create a correctly structured program. Semantics handles the meaning assigned to a combination of symbols. Because most languages are textual, this article focuses on textual syntax. The syntax of a programming language is often established using a combination of standard expressions and grammatical structures.

(ii) **Semantics:** The word semantics (It is the study of meaning and reference. The word can be applied to various different areas, including theory, language study, and computer science.) relates to the meaning of languages rather than their structure (syntax).

(iii) **Type System:** A type system specifies how the programming language categorizes items and expressions into formats, how those kinds may be manipulated, and how they interact. A type system's purpose is to identify and typically enforce a particular level of accuracy in programs written in that language by recognizing certain erroneous operations. Any sort of decidable system includes a trade-off: although it rejects many wrong programs, it may also prevent certain accurate, but rare, ones. A variety of languages feature type loopholes, which are generally unchecked casts that can potentially used by the programmer to expressly enable a normally forbidden action between distinct types.

In most typed languages, the type system is primarily used to type check programs; however, a few languages, especially functional ones, infer types, reducing the need for the programmer to write

type annotations. Type theory is the formal structure and analysis of type systems.

1.3.3 Design and Implementations

Programming languages have traits with natural languages, such as having a syntactic form independent from its semantics and exhibiting language families of related languages branching one from another. However, because they are artificial creations, they differ fundamentally from languages that have evolved through usage. Because a programming language has a clear and limited definition, it can be thoroughly defined and examined in its entirety. Natural languages, on the other hand, have changeable meanings as determined by their users in different groups. While constructed languages are artificial languages that are built from the ground up with a specific goal in mind, they lack the exact and comprehensive semantic definition that a programming language possesses.

Several programming languages have been developed and created from scratch, modified to fit new requirements, and integrated with others. Many have ultimately gone out of favor. Although attempts have been made to create a single "global" programming language that suits all needs, none of them have been widely regarded as fulfilling this function. The range of situations in which languages are employed necessitates the development of varied programming languages:

- Programs can vary greatly in scale—from simple scripts written by individual enthusiasts to complex systems developed by large teams of programmers.
- Developers possess a wide range of expertise, from beginners who prioritize simplicity to advanced professionals capable of handling

intricate designs.

- Software must be optimized for performance, memory usage, and clarity, whether it's running on small microcontrollers or powerful supercomputers.
- Some programs remain unchanged for long periods, while others undergo frequent updates and modifications.
- Developers often have personal or professional preferences, influenced by their familiarity with certain problem-solving methods or programming languages.

One recurrent tendency in programming language evolution has been to offer a greater capacity to handle issues at a higher degree of abstraction. The early programming languages were inextricably linked to the computer's underlying hardware. As new programming languages have evolved, capabilities that allow programmers to express ideas that are further distant from straightforward translation to underlying hardware instructions have been added. Because programmers are less dependent on the computer's complexity, their programs may perform more computation with less effort from the programmer. As a result, they can write more functionality per time unit.

Natural-language programming aims to eliminate specialized language requirements, but its merits are debated. Dijkstra and Perlis condemn it, while Structured English and SQL use hybrid techniques. The designers and users of a language must construct a variety of artifacts that control and facilitate programming practice. The most important of these artifacts are the language definition and implementation.

1.3.4 Uses and Properties

Numerous different programming languages have been developed, mostly in the realm of computing. Individual software projects frequently employ five or more programming languages.

Programming languages demand a greater degree of precision and completeness compared to most human forms of communication. While natural languages tolerate ambiguity and minor errors—often still conveying the intended meaning—computers lack this flexibility. They strictly follow the instructions given and cannot infer what the programmer may have meant. Therefore, the programming language's syntax, the program itself, and its inputs must clearly define the expected behavior during execution, within the scope of the program's control. To express algorithmic ideas without needing full accuracy for execution, pseudo code is often used. This method blends everyday language with code-like constructs to help convey computational logic in a more understandable way. Programming languages, however, offer a structured means to define data and outline how it can be manipulated or processed systematically. Through the abstractions these languages provide, a programmer can translate complex concepts into precise, executable instructions.

Fundamentally, programming involves using a set of core building blocks to represent and solve problems. The process of writing code entails combining these primitives to create new software or adapting existing code to suit new requirements or changing conditions. This creative and logical activity forms the essence of software development. Programs can be executed automatically in batch mode or interactively with user input, often through an interpreter. In interactive scenarios, individual commands are

essentially small programs executed in sequence. Languages that allow this real-time, line-by-line execution without prior compilation are commonly known as scripting languages.

1.4 BASICS OF 'C' LANGUAGE

It is one of the most widely used programming languages on the planet, because the syntax is comparable, if you know C, you will have no trouble learning other popular programming languages such as Java, Python, C++, C#, and so on. C is extremely quick when compared to other programming languages such as Java and Python. C is a very flexible programming language that may be utilized in both applications and technology. It is a general-purpose programming language that was created in 1972 and is still widely used today. It is quite powerful; it has been used to create systems for operating systems, databases, applications, and so on.

1.4.1 History of C Language

The creation of C is inextricably linked to the development of the Unix operating system, which was first developed in assembly language on a PDP-7 by Dennis Ritchie and Ken Thompson, who included other ideas from colleagues. They eventually opted to move the operating system to the PDP-11. The first PDP-11 version of Unix was written in assembly language as well. Thompson wished to use a programming language to create utilities for the new platform. He first attempted to create a Fortran compiler but quickly abandoned the concept. Instead, he designed BCPL, a simplified version of the recently discovered systems programming language. Because the official definition of BCPL was not available at the time, Thompson adjusted the syntax to be less complex and more comparable to SMALGOL, a simplified ALGOL. Thompson dubbed the outcome B. B was described as

"BCPL semantics with an abundance of SMALGOL syntax" by him. B, like BCPL, featured a bootstrapping compiler to make porting to new machines easier. However, because B was too sluggish and couldn't make use of PDP-11 characteristics like byte addressability, few utilities were eventually created in it.

(i) **First C and new B launch:** In 1971, Ritchie enhanced B to use PDP-11, introducing New B (NB) as a character data type. Thompson used NB to design the Unix kernel, influencing its evolution. NB introduced arrays of int and char, pointers, arrays of all types, and function return types. Arrays within expressions were converted to pointers, and the language was renamed C. Version 2 Unix, commonly known as Research Unix, contained the C compiler and several tools written with it.

(ii) **Mechanisms and modifications of the Unix kernel:** The Unix kernel was largely re-implemented in C in Version 4 Unix, which was published in November 1973. The C language has gained several significant features by this point, such as struct types. The preprocessor was created in 1973, at the request of Alan Snyder, and in acknowledgment of the utility of the file-inclusion techniques provided in BCPL and PL/I. Its initial form simply contained files and basic string replacements: parameter less macro #include and #define. Soon after, it was expanded to include macros with arguments and conditional compilation, mainly by Mike Lesk and subsequently by John Reiser.

Unix was among the first operating system kernels written in a language other than assembly. In 1961, the Multics system (written in PL/I) and the Master Control Program (MCP) for the Burroughs B5000 (written in ALGOL) were examples. Ritchie and Stephen C. Johnson made additional improvements to the language in 1977

to improve the portability of the Unix operating system. Johnson's Portable C Compiler was the foundation for various C implementations on new platforms.

The commented-out int type specifiers might be deleted in K&R C, but are mandatory in subsequent standards. Function parameter type examines were not performed because K&R function declarations did not include any information about function arguments, although some compilers would issue a warning message if a local function was called with the incorrect number of arguments, or if multiple calls to an external function used different numbers or types of arguments. Separate tools, such as Unix's lint program, were created to ensure function uniformity across various source files. Several features were added to the language in the years after its release, backed by compilers from AT&T and other manufacturers. These were some examples:

- void functions
- functions that return struct or union kinds
- struct data type allocation
- enumerated types

The vast number of extensions and absence of agreement on a standard library, along with the popularity of the language and the fact that not even Unix compilers accurately implemented the K&R specification, made standardization necessary.

(iii) **ISO C and ANSI C:** In the late 1970s and 1980s, C was implemented for various mainframe computers, minicomputers, and microcomputers, including the IBM PC. In 1983, the American National Standards Institute (ANSI) formed a committee, X3J11, to establish a standard specification of C. The standard was ratified as ANSI X3.159-1989 "Programming Language C" in 1989, often

referred to as ANSI C, Standard C, or sometimes C89. In 1990, the ANSI C standard was adopted by the International Organization for Standardization (ISO) as ISO/IEC 9899:1990, also known as C90. The C standardization process aimed to produce a superset of K&R C, incorporating unofficial features and additional features. C89 is supported by current C compilers and most modern C code is based on it.

(iv) **C99:** The C standard was revised in the late 1990s, leading to the publication of ISO/IEC 9899:1999 in 1999, also known as "C99". C99 introduced new features such as inline functions, data types, variable-length arrays, flexible array members, floating point support, variadic macros, and one-line comments. Although backwards compatible with C90, C99 is stricter in some ways. GCC, Solaris Studio, and other C compilers now support many or all of C99's new features. Microsoft Visual C++ implements the C89 standard and C99 parts for compatibility with C++11. C99 also requires support for identifiers using Unicode and suggests support for raw Unicode names.

(v) **C11:** The C standard, formerly known as "C1X," was developed in 2007 and officially released as ISO/IEC 9899:2011 in 2011. It introduced new features like type generic macros, anonymous structures, Unicode support, atomic operations, multi-threading, and bounds-checked functions.

(vi) **C17 (C standard revision):** The latest standard for the C programming language is C17, which was published in June 2018 as ISO/IEC 9899:2018. It has no new language attributes, merely technical repairs and explanations of C11 flaws. 201710L is the standard macro _STDC_VERSION_.

(vii) **C23 (C standard revision):** C23 is the informal moniker for

the next main C language standard version (following C17). It is scheduled to be released in 2024.

(viii) **Embedded C:** It has always required nonstandard additions to the C language in order to handle exotic features like as fixed-point arithmetic, many different memory banks, and fundamental I/O operations. The C Standards Committee produced a technical report in 2008 that extended the C language to solve these difficulties by creating a uniform standard to which all implementations must comply. It has features not found in standard C, including as fixed-point arithmetic, named address spaces, and basic I/O hardware addressing.

1.4.2 The Structure of 'C' Program

A 'C' program's basic structure is separated into six components, making it easier to read, edit, document, and comprehend in a specific manner. In order to build and execute effectively, the C program must adhere to the guidelines outlined below. In a well-structured C program, debugging is easy.

(i) **Sections of the C Program:** A program's proper execution is the responsibility of six basic components. The following sections are mentioned:

a) **Documentation:** This part includes a program description, the program's name, and the program's inception date and time. It is supplied in the form of comments at the beginning of the program. Documentation can be represented in the following ways:

```
// description, name of the program, programmer name, date, time etc.
```

OR

```
/* description, name of the program, programmer name, date, time etc. */
```

Anything placed in comments will be viewed as program documentation and will not interfere with the supplied code. Essentially, it provides the reader with an overview of the software.

b) **Preprocessor Section:** All of the program's header files will be defined in the program's preprocessing section. Header files allow us to incorporate better code from others into our own. Before the compilation process, a copy of these various files is placed into our software. **Example:**

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

c) **Definition:** Preprocessors are programs that process our source code prior to compilation. There are several processes involved in the creation and implementation of the program. Preprocessor instructions begin with the symbol '#'. The #define preprocessor is used to declare a constant that will be used throughout the program. When the compiler encounters this term, it replaces it with the actual piece of declared code.

Example:

```
#define long long ll
```

d) Global Declaration: Global variables, function declarations, and static variables are all included in the global declaration section. Variables and functions specified in this scope can be utilized throughout the application.

Example:

```
int num = 18;
```

e) **Main() Function:** A main function must be present in every C program. This section contains the program's main() function. Declaration and implementation are carried out within the curly braces in the main program. The return type of the main() function can be either int or void. The compiler is informed by void() main that the program will not produce any value. The main() function instructs the compiler that the code will return a numerical value.

Example:

```
void main()
```

OR

```
int main()
```

f) **Sub Programs:** In this area of the program, user-defined functions

are invoked. When they are called from the main() function or from outside the main() function, control of the program is transferred to the called function. These are specified in accordance with the programmer's needs.

Example:

```
int sum(int x, int y)
{
return x+y;
}
```

(ii) **Structure of the C Program with Example:**

Example: Below C program to find the sum of 2 numbers:

```
// Documentation
/**
* file: sum.c
* description: program to find sum.
*/

// Link
#include <stdio.h>

// Definition
#define X 20

// Global Declaration
int sum(int y);

// Main() Function
int main(void)
{
int y = 55;
printf("Sum: %d", sum(y));
return 0;
}

// Subprogram
int sum(int y)
```

```
{
return y + X;
}
```

Output:

Sum: 75

Explanation of the aforementioned Program:

The aforesaid software is explained in detail below. With a description of the program's meaning and application.

Sections	Description
/ * file: sum.c * author: you * description: program to find sum. */**	It is the comment section and is part of the description section of the code.
#include<stdio.h>	Header file which is used for standard input-output. This is the preprocessor section.
#define X 20	This is the definition section. It allows the use of constant X in the code.
int sum(int y)	This is the Global declaration section includes the function declaration that can be used anywhere in the program.
int main()	main() is the first function that is executed in the C program.
{…}	These curly braces mark the beginning and end of the main function.
printf("Sum: %d", sum(y));	printf() function is used to print the sum on the screen.
return 0;	We have used int as the return type so we have to return 0 which states that the given program is free from the error and it can be exited successfully.

Sections	Description
int sum (int y) **{** **return y + X;** **}**	This is the subprogram section. It includes the user-defined functions that are called in the main () function.

Compilation and the execution of a C program involves the following steps are Program Creation, Compilation of the program, Execution of the program and output of the Program.

The following are some of the elements we acquired about the structure of the C Program in this article:

- A C program's basic structure is separated into six components, making it easier to read, edit, document, and comprehend in a specific manner.
- In a well-structured C program, debugging is easy.
- A C program is divided into six sections: Documentation, Preprocessor Section, Definition, Global Declaration, Main() Function, and Sub Programs.
- The following stages are taken during the compilation and execution of a C program:

i). Development of a Program

ii). Putting together the program

iii). The initiative is being carried out.

iv). Result of the program

(iii) **Syntax of C Program:**

The C standard specifies a formal grammar for C. Line ends are not normally relevant in C; nevertheless, line boundaries are during the preprocessing step. Comments can appear within the delimiters /* and /, or (since C99) after // until the end of the line. Comments delimited by / and */ do not nest, and if they occur inside string or

character literals, they are not interpreted as comment delimiters. The declarations and defined functions are found in C source files. Declarations and statements are included in function definitions. Declarations either declare new types with keywords like struct, union, and enum, or assign types to and maybe reserve storage for new variables by writing a type followed by the variable name. Built-in types are specified via keywords that include char and int. Braces (and, sometimes known as "curly brackets") are used to limit the scope of declarations and to serve as a single statement for structuring controls.

C, being an imperative language, specifies actions using statements. The most frequent statement is an argument statement, which consists of an argument to be evaluated preceded by a semicolon; as a result of the assessment, functions and variables may be called and new values set. C includes many control-flow statements designated by reserved keywords to change the regular sequential execution of statements. If... [otherwise] conditional execution and do... while, while, and for sequential execution (looping) provide structured programming. The for statement has distinct initialization, evaluation, and reinitialization expressions, which can be omitted in any order. Within the loop, break and continue can be utilized. Break is used to exit the innermost contained loop statement, while continue skips to its reinitialisation. There is also a non-structured goto statement that goes directly to the function's assigned label. The value of an integer expression is used to pick a case to be performed by switch. Unlike numerous other languages, the the control process will continue to the next case until interrupted by a break.

Expressions in C can employ a variety of integrated operators and call function, without no restriction on the sequence of

evaluations. All side effects, including variable storage, occur before the next "sequence point," which includes the conclusion of each expression statement as well as the entrance to and return from each function call. Sequence points can also appear during the assessment of expressions that contain specific operators (&&, ||,?:, and the comma operator). This allows the compiler to optimize object code at a high level, but it needs greater effort from programmers to get accurate results. Because of the impact on existing software, the C standard was unable to remedy many of these flaws.

1.4.3 Features of ‘C’ Language

C is an imperative programming language. Dennis Ritchie was the first to create it in 1972. It was created primarily as an operational programming language for creating an operating system.

C language's core characteristics are low-level memory access, a minimal set of keywords, and an elegant style; these characteristics make C language excellent for system programming such as computer operating systems or compiler development.

What are the Most Significant C Language Characteristics?

Here are a few of the most essential C language features:

- Procedural Language Structure: Languages like C follow a procedural programming approach, where instructions are executed in a defined sequence. In C, programs are typically structured into multiple functions, each responsible for a specific task. This step-by-step execution can make it easier for beginners to follow the program’s flow. While many new programmers might assume this is the only programming style, there are multiple paradigms in the software world, with object-oriented programming (OOP) being one of the most prominent alternatives.

- Speed and Efficiency: Although modern programming languages like Python and Java offer more built-in features, they often compromise on execution speed due to additional layers of abstraction. C, as a middle-level language, allows direct interaction with hardware components, making it exceptionally efficient. Its performance benefits from being statically typed, which ensures variable types are known at compile-time, reducing overhead during runtime. This efficiency and speed are key reasons why C is often recommended as a starting point for learning programming.
- Modular Design and Reusability: C promotes the concept of modularity by allowing code to be divided into reusable libraries. These libraries enable developers to manage complex codebases more efficiently by reusing common functionalities. The standard C library is a powerful toolset that provides pre-built solutions for frequently encountered problems, thereby enhancing development productivity and maintaining cleaner code architecture.
- Versatile and Extensible Language: C is recognized as a general-purpose programming language with applications spanning from operating systems (like Windows, Linux, and iOS) to database engines (such as MySQL and Oracle). Its robust set of built-in operators and extensive libraries makes it suitable for a wide range of tasks. As a middle-level language, it bridges the gap between low-level hardware manipulation and high-level application development. C programs are also highly portable across systems, and their extensible nature means new functionalities can easily be added to existing codebases.

1.4.4 'C' Program Installation

- Installing the C Programming Language Across Different Operating Systems Setting up a development environment for C programming varies slightly depending on the operating system in use. Below is a general guideline for installing C on popular platforms such as Windows and macOS.
- For Windows Users: To begin programming in C on a Windows system, you'll first need an Integrated Development Environment (IDE) like Visual Studio Code, Code::Blocks, or Dev-C++. After selecting and installing your preferred IDE, the next step is to install a C compiler. Common options include GCC via MinGW or Clang, both of which enable you to compile and execute C programs directly from your development environment.
- For macOS Users: On macOS, one of the most convenient ways to start programming in C is by installing Xcode. Xcode is a complete development suite that includes a built-in C compiler and tools for writing, compiling, and debugging code. If you prefer not to use Xcode, you can opt for standalone compilers such as GCC or Clang, which can be installed using Homebrew, a package manager for macOS.

- **Linux:**

  Most Linux distributions come with GCC (GNU Compiler Collection) preinstalled. You can use a terminal or command-line interface to compile and run C programs.

  You can also install an IDE such as Code::Blocks or Dev-C++ if you prefer a graphical development environment.

  NOTE: The exact procedures to install C may differ based on the operating system edition and the tools you use. Make

careful to follow the manufacturer's or community's directions.

Here's how to install the C programming language on Windows, macOS, and Linux:

Windows:

- Download and install Visual Studio Code from the official website: https://code.visualstudio.com/download
- Download and install MinGW (GCC) from the official website: https://osdn.net/projects/mingw/releases/
- Open Visual Studio Code and go to Extensions, then search for "C/C++" and install the Microsoft C/C++ extension.
- Open the Command Palette (Ctrl + Shift + P) and select "C/C++: Edit configurations (UI)". Add the following configuration in the "tasks.json" file:

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "type": "cppbuild",
      "label": "C/C++: g++.exe build active file",
      "command": "C:\\MinGW\\bin\\g++.exe",
      "args": [
        "-g",
        "${file}",
        "-o",
        "${fileDirname}\\${fileBasenameNoExtension}.exe"
      ],
      "options": {
        "cwd": "C:\\MinGW\\bin"
      },
```

```
        "problemMatcher": [
          "$gcc"
        ],
        "group": {
          "kind": "build",
          "isDefault": true
        }
      }
    ]
}
```

- Write your first C program, save it with a “.c” extension, and press Ctrl + Shift + B to build and run it.

macOS:

- Launch Terminal and perform the following command to install Xcode Command Line Tools:

```
xcode-select –install
```

- Install GCC by running the following command:

```
brew install gcc
```

- Write your first C program in a text editor, save it with a “.c” extension, and compile it using GCC by running the following command:

```
gcc -o myProgram myProgram.c
```

- Run your program by typing the following command:

```
./myProgram
```

Linux:

- Most Linux distributions come with GCC preinstalled. You can check if GCC is installed by running the following command in a terminal:

```
gcc --version
```

- If GCC is not installed, you can install it using the package manager of your distribution. For example, on Ubuntu, you can run the following command:

```
sudo apt-get install build-essential
```

- Write your first C program in a text editor, save it with a ".c" extension, and compile it using GCC by running the following command:

```
gcc -o myProgram myProgram.c
```

- Run your program by typing the following command:

```
./myProgram
```

Note: These examples are intended to be given you as an idea of how to install C and compile a small application. The precise methods may differ based on the applications and operating system versions you are running.

1.4.5 Tools and Uses

(i) **Tools or IDE for 'C' program**

Though we've covered the relevance and demand for the C language, in this unit we'll look in depth at a critical precondition necessary for conducting programming in the C language, namely, a C IDE (Integrated Development Environment). In general, IDEs are designed to make things simpler for developers and boost their productivity by including tools such as a code editor, debugging support, a compiler, auto code completion, and many more. A C IDE offers you with a full collection of tools for the developing applications in C languages. There are various C IDEs accessible for both experienced developers and beginner programmers to use to program without difficulty, and you may choose any one of them based on your needs.

Meanwhile, to make your job easier, we've produced a list of the best C IDEs:

a) **Visual Studio:** First and foremost, here is an enlightening a Integrated and Development Environment (IDE) created by the

IT behemoth, Microsoft. Microsoft's Visual Studio as provides you a with several impressive capabilities like automated code completion, code redesigning, syntax highlighting, assistance with various languages, and many more. Aside from C/C++ and C#, Visual Studio supports a variety of additional languages via plugins, including JavaScript, TypeScript, XML, and others, as well as Python, Ruby, and others. Its features are as follows:

- Compatible with: Windows, macOS, and Linux
- Code completion using IntelliSense
- Built-in Git Integration
- Easy Azure Development
- Integrated Debugger and VCS support

b) **CLion:** CLion is another popular cross-platform C/C++ Integrated Development Environment (IDE) for programmers that is integrated with the CMake build system and supports macOS, Linux, and Windows. It was created by JetBrains and includes a smart C/C++ code editor for improved code help, safe refactoring and rapid documentation, the ability to test individual units of source code, effective code and project management, and so on. In addition to C/C++, CLion supports numerous more languages via plugins, including Kotlin, Python, Swift, and others. Its features are as follows:

- Integrated debugger
- On-the-fly code analysis
- Supports Embedded Development
- Supports CVS (Concurrent Versions System) & TFS (Team Foundation Server)
- Compatible with: Windows, macOS, and Linux

c) **Eclipse:** Eclipse is a well-known brand in the area of

Integrated Development Environments (IDEs). Although it is best recognized for its outstanding support for Java, Eclipse has also proven to be a valuable IDE for C and C++. It has several essential features for C programming, such as code auto-completion, code refactoring, visual debugging tools, remote system explorer, and many more. Furthermore, you may enhance the functionality of Eclipse IDE by incorporating numerous additional plugins according on your needs. Its features are as follows:

- Open-source & Rich Community
- Compatible with: Windows, macOS, and Linux
- Easier Project Creation
- Supports Static Code Analysis
- Easy Debugging

d) **Code::Blocks:** Going down the list, we have Code::Blocks, an open-source C IDE written in C and built with the wxWidgets GUI toolkit. Code::Blocks has all of the essential features needed for C and C++ programming, such as syntax highlighting, a tabbed interface, code completion, code coverage, simple navigation, debugging support, and so on. Furthermore, it allows you to include full breakpoint conditions, which means you may stop the code execution if the condition is true. The Code::Blocks IDE's source code and may make changes based on your choices for a C Integrated Development Environment. Its features are as follows:

- Compatible with: Windows, macOS, and Linux
- Supports multiple compilers – GCC, Clang, and Visual C++
- Extensible via plugins

- Full Breakpoints Support
- Open-source & Rich Community

e) **CodeLite:** CodeLite is a different open-source C and IDE (Integrated Development Environment) that many developers prefer. It improves compiler compatibility by including built-in assistance with GCC, Clang, and Visual C++, and it is also compatible with additional languages outside C/C++, such as PHP, JavaScript (Node.js), and others. CodeLite provides you with a plethora of useful tools, like code restructuring, organizing projects, code browsing, syntax highlighting, test automation, and a lot more. CodeLite also includes a number of other features like clickable errors, clang-based completion of code for C projects, and so on. CodeLite also offers a Rapid Application Development application for creating wxWidgets-based apps. Its features are as follows:

- Compatible with: Windows, macOS, and Linux
- Project Management
- Interactive Debugger
- Valgrind Support
- Supports Static Code Analysis

f) **NetBeans:** NetBeans, created by the Apache Software Foundation - Oracle Corporation, is additionally a popular IDE among C/C++ developers. This open-source and free Integrated Development Environment (IDE) allows you to develop C and C++ programs using dynamic and static libraries. NetBeans provides several C/C++ development enhancements such as code restructuring, bracket matching, automated indentation, unit evaluation, and many more. Furthermore, it provides excellent support for a wide range of

compilers, including Oracle Solaris Studio, GNU, CLang/LLVM, Cygwin, MinGW, and others. NetBeans additionally provides features like as quicker file navigation, source inspection, packaging, and so on. NetBeans, like Eclipse, has improved drag and drop functionality, which is why it is highly recommended for students and beginner-level C/C++ developers. Its features are as follows:

- Free and Open Source
- Compatible with: Windows, macOS, Linux, and Solaris
- Qt Toolkit Support
- Supports Remote Development
- Efficient Project Management

Therefore, these are the most recommended IDEs for C developers, together with their individual features and benefits. However, before selecting any of the IDEs, you must first determine your requirements, since this is really important! For example, if you require a if you are a beginner-level programmer looking for greater drag-and-drop functionality, you may use NetBeans or Eclipse; and so on.

(ii) **Uses of 'C' program**

a) Justification for usage in system programming:

C is a programming language that is commonly used in the implementation of operating systems and embedded system applications. This is due to numerous factors:

- The code generated after compilation processes does not need many system features and can be invoked in a straightforward manner from some boot code - it is easy to

execute.

- The C language statements and operators typically map properly on a sequence of commands for the target processor, resulting in a low run-time interest on system resources - it is fast to execute.
- The C language, with its extensive collection of operators, can take use of many of the capabilities of target CPUs. Where a certain CPU contains more exotic instructions, a language version may be built with perhaps inherent functions to take advantage of those instructions - it can leverage almost all of the target CPU's characteristics.
- The language allows for the simple overlay of structures over blocks of data that are binary, enabling the data to be understood, traversed, and updated - it can create data structures and even file systems.
- The language has a rich collection of operators for integer computation and logic, as well as perhaps varying sizes of floating-point values - it can process suitably organized data effectively.
- C is a little language, with only a few statements and few features that create a lot of target code - it's understandable.
- C offers immediate control over the allocation of memory and deallocation, which provides fair efficiency and predictable time to memory-handling tasks while eliminating the need to worry about periodic stop-the-world garbage disposal events - it has steady performance.
- C allows for the usage and implementation of several memory allocation strategies, such as the standard malloc and free; a complicated mechanism for various domains; or a version for an OS kernel which could suit DMA, be used in interrupt handlers, or incorporate with the virtual memory system.

- Because platform hardware can be accessible via pointers and type punning, system-specific functions (e.g. Control/Status Registers, I/O registers) may be set and utilized with C code - it interacts well with the platform on which it runs.
- Based on the linker and environment, C code may also call assembly language libraries and be called from assembly language - it collaborates well with other lower-level programs.
- C, as well as its calling conventions and linker structures, are frequently used in combination with other high-level languages, with calls to and from C supported - it interoperates well with other high-level programs.
- C has a developed and diverse ecosystem that includes frameworks, libraries, and open source compilers, debuggers, and tools, it is the de facto standard. It is likely in that the drivers already exist in C, or that a similar CPU architecture exists as a back-end of a C compiler, therefore there is less motivation to use another language.

b) Previously used for web development:

C was formerly used for web development, with the Common Gateway Interface (CGI) acting as a "gateway" to transfer data between the web application, the server, and the browser. With its speed, security, and near-universal availability, C may have been selected over interpreted languages. It is a no longer usual practice to design websites with C, and there are several alternative web development tools available.

c) Some additional languages are written in C as well:

Because of C's widespread availability and efficiency, compilers, libraries, and interpreters for other programming languages are frequently written in C. Python, Perl, Ruby, and

PHP, for example, have reference implementations written in C.

d) Allows use with computationally demanding libraries:

Although the layer of abstractions from hardware is thin and the overhead is minimal, C allows programmers to develop efficient implementation of algorithms and data structures, which is a key criteria for computationally intensive systems. The GNU Multi Precision Arithmetic Library, the GNU Scientific Library, mathematically, and MATLAB, for example, are written entirely or partially in C. Many languages enable invoking C library functions; for example, the Python-based architecture NumPy makes use of C for high-performance and hardware interaction.

e) C is used as an intermediate language:

C sometimes can be utilized as an intermediate language by other language implementations. This method can be employed for scalability or convenience; utilizing C as an intermediary language eliminates the need for machine-specific code generators. C has certain characteristics that aid with the compilation of produced code, such as line-number preprocessor directives and optional unnecessary commas at the end of initializer lists. However, some of C's inadequacies have encouraged the development of additional C-based languages, such as C--, that are expressly designed for usage as intermediate languages. Furthermore, the modern main compilers GCC and LLVM both provide an intermediate format that is not C, and both allow front ends for numerous languages, including C.

f) ***End-user programs:***

C is also commonly used to create end-user apps. Such programs, however, can also be created in more recent higher-level languages.

1.5 PROGRAMMING FORMAT OF 'C'

In C, the format specifier informs the compiler about the kind of data to be written or scanned during input and output operations. They usually begin with a % symbol and are utilized in formatted strings in functions such as printf(), scanf(), sprintf(), and so on. The C programming language has a number of format specifiers associated with various data types, like %d for int, %c for char, and so on. This article will go over some of the most often used formatting specifiers and how to utilize them.

The table below lists the most widely used format specifiers in C:

Format Specifier	Description
%c	For character type
%d	For signed integer type
%e	For scientific notation of floats
%f	For floats type
%g	For float type with the current precision
%i	Unsigned integer
%Id or %li	Long
%If	Double
%Lf	Long double
%lu	Unsigned int or Unsigned long
%lli	Long long
%llu	Unsigned long long
%o	Octal representation
%p	Pointer
%s	String
%u	Unsigned int
%x	Hexadecimal representation
%n	Print nothing
%%	Print % character

1.5.1 Basic format and functions

1. Character Format Specifier – %c in C:

In C, the format specifier for the char data type is %c. It may be used in C for both formatted input and formatted output.

Syntax:

```
scanf("%d...", ...);
printf("%d...", ...);
```

Example:

```
// C Program to illustrate the %c format
specifier.
#include <stdio.h>
int main()
{
        char c;
        // using %c for
        character input
        scanf("Enter some
        character: %c", &c);
// using %c for character
        output
        printf("The entered
        character: %c", &c);
        return 0;
}
Input:
Enter some character: A
Output:
The entered character: A
```

2. Integer Format Specifier (signed) – %d in C:

The signed integer format specifier%d can be used in the scanf() and print() methods, as well as other functions that employ formatted text for int data type input and output.

Syntax:

```
scanf("%d...", ...);
printf("%i...", ...);
```

Example:

```
// C Program to demonstrate the use of %d
        and %i
#include <stdio.h>
// Driver code
int main()
{
        int x;
        // taking integer input
```

```
        scanf("Enter the two
        integers: %d", &x);

        // printing integer
        output
        printf("Printed using
        %%d: %d\n", x);
        printf("Printed using
        %%i: %3i\n", x);
        return 0;
}
```

Input:
Enter the integer: 45
Output:
Printed using %d: 45
Printed using %i: 45

3. Unsigned Integer Format Specifier – %u in C:

The format specifier for the unsigned integer data type is %u. When we provide the %u a negative integer value, it transforms it to its first complement.

Syntax:

```
printf("%u...", ...);
scanf("%u...", ...);
```

Example: Write a C program below shows how to the use %u in C.

```
// C Program to illustrate the how to use %u
#include <stdio.h>
// driver code
int main()
{
        unsigned int var;
        scanf("Enter an integer: %u", &var);
        printf("Entered Unsigned Integer:
        %u", var);

// trying to print negative value using %u
        printf("Printing -10 using %%u:
        %u\n", -10);
        return 0;
}
```

Input:
Enter an integer: 25
Output:
Entered unsigned integer: 25
Printing -10 using %u: 2494692768

4. Floating-point format specifier – %f in C:

The %f is floating-point format specifier in C that can be used inside a formatted string for float data input and output. In addition to %f, we may use the format specifiers %e or %E to display the floating-point value in exponential form.

Syntax:

```
printf("%f...", ...);
scanf("%e...", ...);
printf("%E...", ...);
```

Example:

```
// Here C program demonstrates to use of %f, %e and %E
#include <stdio.h>
// driver code
int main()
{
float a = 12.67;
printf("Using %%f: %f\n", a);
printf("Using %%e: %e\n", a);
printf("Using %%E, %E", a);
return 0;
}
```

Output:

```
Using %f: 12.670000
Using %e: 1.267000e+01
Using %E, 1.267000E+01
```

5. Unsigned Octal number for integer – %o in C:

In the C program, we may use the%o format specifier to print or accept input for the unsigned octal integer number.

Syntax:

```
printf("%o...", ...);
scanf("%o...", ...);
```

Example:

```
#include <stdio.h>
int main()
{
int a = 67;
printf("%o\n", a);
return 0;
}
```

Output

```
103
```

6. Unsigned Hexadecimal for integer – %x in C:

For hexadecimal integers, the format specifier %x is used the prepared text. The alphabets in the hexadecimal numerals will be as a lowercase in this situation. Instead of %X, we use %X for uppercase alphabet digits.

Syntax:

```
printf("%x...", ...);
scanf("%X...", ...);
```

Example:

```
// Write a C Program to be demonstrate the use of %x and %X
#include <stdio.h>
int main()
{
int a = 15454;
printf("%x\n", a);
printf("%X", a);
return 0;
}
```

Output

```
3c5e
3C5E
```

7. String Format Specifier – %s in C:

In C, the %s symbol is used to print strings or to accept strings as input.

Syntax:

```
printf("%s...", ...);
scanf("%s...", ...);
```

Example:

```
// C program to illustrate
   the use of %s in C
#include <stdio.h>
int main()
{
char a[] = "Hi Raghav";
printf("%s\n", a);
return 0;
}
```

Example:

The operation of %s with scanf() differs slightly from that of printf(). Let's look at this with the aid of the C program below.

```
// C Program to illustrate the working of %s
   with scanf()
#include <stdio.h>
```

```
int main()
{
char str[50];
// taking string as input
scanf("Enter the String: %s", str);
printf("Entered String: %s", str);
return 0;
}
```

Example:

Input

Enter the string: Hi Raghav

Output

Hi

As we can see, the string is only searched until it encounters whitespace. In C, we may avoid this by using scan sets.

8. Address Format Specifier – %p in C:

The C programming language also has a format specifier for printing addresses/pointers. In C, we may use %p to display addresses and pointers.

Syntax

```
printf("%p...", ...);
```

Example:

```
#include <stdio.h>
int main()
{
 int a = 10;
 printf("The Memory Address of a:
       %p\n",(void*)&a);
 return 0;
}
```

Output

The Memory Address of a: 0x7ffe9645b3fc

Input and Output Formatting:

The C programming language has certain facilities for formatting input and output. They are often placed between the % sign and the format specifier symbol. Here are a few examples:

A negative (-) symbol indicates left alignment.

A number after % defines the minimum field width to be printed; if the characters are fewer than the width, the leftover space is filled with space; if it is more, it is displayed as is without truncation.

A period (.) sign denotes the separation between field width and accuracy.

Precision specifies the number of digits in an integer, the number of characters in a string, and the number of digits after the decimal point in a floating value.

Example of I/O Formatting:

```
// C Program to demonstrate the formatting
       methods.
#include <stdio.h>
int main()
{
char str[] = "beginner";
printf("%20s\n", str);
printf("%-20s\n", str);
printf("%20.5s\n", str);
printf("%-20.5s\n", str);
return 0;
}
```

Output:

```
beginner
beginner
     begin
begin
```

5.1.2 A simple C programming

C Program To Print Your Own Name:

In this case, we have two approaches for printing the name:

- Using printf()
- Using scanf()

Input:

Enter Name: Raghav

Output:

Name = Raghav

Example1:

In this example, we use the printf() function to print the user name.

```
// C program to demonstrate printing of
// our own name using printf()
#include <stdio.h>
int main()
{
// print name
   printf("Name : Raghav");
   return 0;
}
```

Output:

Name = Raghav

Example2:

In this example, we use scanf() to accept the user's name and then print it.

```
// C program to demonstrate printing of
// our own name using scanf()
#include <stdio.h>
int main()
{
char name[20];
printf("Enter name: ");
// user input will be taken here
scanf("%s", name);
printf("Your name is %s.", name);
return 0;
}
```

Output:

Enter Name: Raghav
Name = Raghav

1.6 CREATING A C PROGRAM:

Creating and Editing a C Program Across Operating Systems

C programs can be written and executed on various platforms such as DOS and UNIX, and are typically saved with the .c extension. On DOS-based systems, source code can be entered using any basic text editor like EDIT. For example, to open and edit a file named testprog.c, the command used would be:

C:\> edit test1prog.c

For those working with Turbo C, the environment includes its own built-in editor. To launch the Turbo C editor, navigate to the executable using its full path. For instance:

C:\> turboc\bin\tc

This command runs the Turbo C IDE, allowing users to write and save their programs within its interface. Files saved this way are automatically given a .c extension.

On UNIX systems, C source files are also saved with the .c

extension to indicate they are C programs. Programmers often use editors such as vi, emacs, or xedit to create and modify code. For example, to edit testprog.c using vi, the following command is used:

$ vi testprog.c

These editors allow programmers to make changes to the code at any time during development.

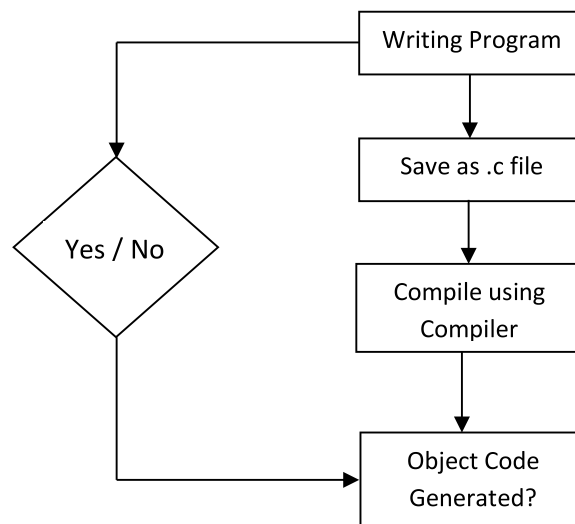
1.7 COMPILATION PROCESS IN C PROGRAM

Once a program is written, it should be saved with a .c extension, indicating that it is written in the C programming language. While C is considered a high-level language, computers do not directly understand it. Therefore, the next essential step is to convert the human-readable source code into machine-readable object code. This conversion is handled by a specialized tool known as a compiler. Each programming language typically has its own compiler designed to interpret and translate its syntax into machine code. The compiler analyzes the source code for any syntax errors and, if none are found, it proceeds to generate the object code.

However, if the compiler detects any errors in the code, it will not produce the object file. These errors must be resolved before the program can be successfully compiled. The process ensures that only syntactically correct code is translated into executable instructions that the computer can understand and run.

A flowchart can help visualize this process—starting from writing and saving the program, followed by compilation, and ending

either in successful object code generation or an error report. This structured approach highlights the fundamental steps involved in creating and preparing a C program for execution.



This flowchart represents the sequential steps of writing a program, saving it with the ".c" extension, using a compiler to compile the code into object code, and then checking whether the as a object code was successfully generated or not.

1.7.1 The Compiler

On a UNIX-based system, if your program file is named testprog.c, you can compile it using a simple command in the terminal:

cc testprog.c

This command compiles the source code and, if no syntax errors are present, generates an executable file by default named a.out. If you prefer a custom name for the output file, you can use the -o option:

cc testprog.c -o testprog

This command compiles the source file and creates an executable

named testprog instead of the default a.out.

On the other hand, if you're using Turbo C in a DOS environment, the compiler options are available directly through the graphical menu. Once you compile a program with correct syntax, it generates an object file with a .obj extension, for example, testprog.obj. If there are syntax errors, they will be displayed on the screen, and the object file will not be created.

1.7.2 Semantic and Syntax Errors

Each programming language has its own defined set of grammatical rules, and any code written in that language must adhere to those rules. For instance, consider the English sentence: “Raghav, is playing, with a ball.” This sentence is grammatically incorrect due to the improper placement of commas. Similarly, in the C programming language, the code must follow a specific syntax.

When a C program is compiled, the compiler checks whether the syntax of the code is valid. If the program contains any syntax errors, the compiler will highlight them, usually displaying the corresponding line numbers to help the programmer identify and fix the issues.

In addition to syntax errors, a C program may also contain semantic errors. These are logical inconsistencies or meaningless statements that the compiler may interpret as warnings rather than errors. Although such programs can still be compiled, it's advisable to correct these warnings to avoid potential issues during execution.

For example, if you declare a variable but never use it, the compiler may display a warning such as “code has no effect.” Even though it doesn’t stop compilation, unused variables consume memory unnecessarily and can clutter the code, potentially leading to confusion or inefficiencies in larger programs.

1.8 LINK AND RUNNING C PROGRAM

Once the compilation process is complete, the next crucial step in the program execution cycle is linking. During compilation, the code is translated into an object file with the .obj extension. However, this file isn't yet ready to run because it may include function calls from C's standard libraries—functions that the user hasn't defined but are referenced in the code through header files.

C programming provides a rich set of standard libraries containing predefined functions for various common operations. When your program references one of these functions, the compiler notes its presence but doesn’t include its actual implementation in the object file. The linker’s job is to connect these function references to their actual definitions found in the standard library.

This linking process is essential for creating a complete, executable program. The linker merges the user-written code with the compiled versions of the library functions, ensuring that all external references are resolved properly.

In summary, the linker is a specialized tool that assembles all the required pieces—both from the user’s code and the standard libraries—into a single executable file, typically with a .exe

extension. This final file is what users can run to see the output of their programs.

1.8.1 Running a Program Using Menu Options

When working with TurboC in a DOS environment, a graphical menu interface appears once the TurboC executable is launched. This menu offers various options for compiling and running C programs:

- **Link**: Performs the linking process after compilation.
- **Make**: Compiles the program and links it in a single step.
- **Run**: Executes the compiled program.

Each of these options leads to the creation of an executable file. To view the output of the program, you need to switch to the user screen window.

1.8.2 Executing an Executable File

Once an .exe file is successfully generated, it can be run directly.
In the **UNIX environment**, a powerful utility named make helps compile complex programs efficiently using a configuration file known as a **make file**. For simple programs stored in a single file, running the following command is sufficient:
make test prog

This command compiles testprog.c, links it with the standard library (e.g., for using functions like printf), and creates an executable file named testprog.

In the **DOS environment**, the .exe file created after successful compilation and linking can be executed directly from the

command prompt. For instance, if your program file is test.c, and it compiles successfully to test.exe, you can run it by typing:
c>test

1.8.3 Linker Errors

Sometimes, a program compiles without any syntax errors, but the executable file isn't created. This can be due to **linker errors**. These errors occur when the compiler recognizes a function declared in a header file but cannot locate its actual implementation in the standard library.

Such mismatches—where the declaration exists but the object code is missing—result in linker errors. These issues must be resolved for the executable to be generated.

1.8.4 Logical and Runtime Errors

After successful compilation and linking, running the program may result in one of the following situations:

1. The program runs correctly and displays the expected output.
2. The program runs, but the output is incorrect.
3. The program fails during execution and stops abruptly.

First Case: Indicates that the program is functioning properly.

Second Case: Points to a **logical error**—an error in the logic of the program that leads to wrong results even though the code compiles and links without issue. These errors are challenging to spot and fix. **Debugging**—the process of examining the program line by line—is essential to identify and correct them. TurboC provides a tracer tool that assists in debugging.

Example: Consider a program meant to calculate the average of three numbers:

```
/* Program to compute average of three numbers
* #include<stdio.h>
  main( )
  {
  int a,b,c,sum,avg;
  a=10; b=5; c=20;
  sum = a+b+c; avg = sum / 3;
  printf("The average is %d\n", avg);
  }
```

OUTPUT:

The average is 8.

Although the correct average is 8.33, the result shows 8. This discrepancy is because avg is an integer variable, and integer division truncates the decimal part. This is a **logical error** because the program logic fails to handle decimal results.

- **Third Case**: When the program crashes during execution, it's a **runtime error**. These errors aren't detected during compilation or linking but occur when the program runs. They often stop the execution and generate an error message on the screen.

Example: Write a program to divide a sum of two numbers by their difference

```
/* Program to divide a sum of two numbers by their difference*/
#include <stdio.h>
main()
{
```

```
int a,b; float c;
a=10; b=10;
c = (a+b) / (a-b);
printf("The value of the result is %f\n",c);
}
```

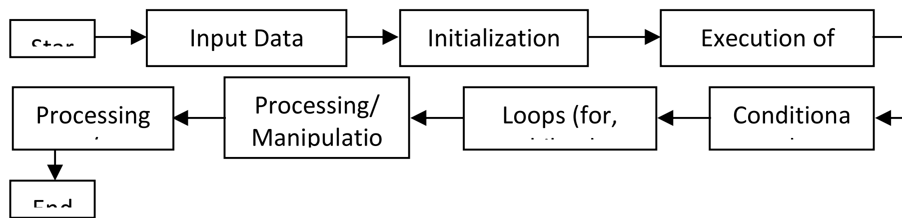
The program mentioned earlier will compile and link without any issues. When executed, it will successfully run up to the first printf statement, and the corresponding message will be displayed on the screen.

However, as soon as the program attempts to execute the next line that involves a division by zero, it encounters a **runtime error**. Specifically, the system throws an error message like **"Divide by zero"**, and the program terminates immediately.

Errors like these, which are not detected during the compilation or linking phases but occur during execution, are referred to as **runtime errors**.

1.9 DIAGRAMMATIC ILLUSTRATION

The diagrammatic depiction of the program execution process is shown in the image below.



A flowchart representing the execution process of a, C program is generally composed of a series of stages. Here's a simple illustration:

In end this flowchart demonstrates the general flow of a, C program. It starts with data input (if necessary), followed by initializing variables, executing the main code or algorithm, applying conditional statements or loops as needed, calling functions (if used), processing data, displaying results, and finally, ending the program. This flow can vary depending on the specific logic and structure of, the C program being executed.

1.10 CONCLUSION

In this unit, you have gained an understanding of what a program and a programming language are. You explored how programming languages are broadly categorized into high-level and low-level types. You also learned how to define the C programming language and identify its key features. The historical development of C was discussed, including its unique position as a middle-level

language that combines aspects of both high-level and low-level languages.

The advantages of using a high-level language over a low-level one were highlighted. You practiced converting algorithms and flowcharts into C programs and explored the steps to write and save a C program in both UNIX and DOS environments.

Additionally, you learned how to compile and execute C programs on these platforms. The unit also introduced you to different kinds of errors encountered during programming—such as syntax, semantic, logical, linker, and runtime errors—and the strategies to correct them. You are now capable of writing basic C programs using arithmetic operations and the printf() function.

With this foundational knowledge in place, you're now prepared to explore more advanced concepts in C programming in the upcoming units.

UNIT 2 BUILDING BLOCKS OF C

Function as building blocks

2.0 Introduction

2.1 Objectives

2.2 Character set of C

2.3 C Tokens

2.4 C Programming Keywords

2.5 Identifiers of the C

2.6 Conclusion

2.7 Unit-based Questions /Answers

2.0 INTRODUCTION

This unit introduces you to a character set in C programming refers to the set of characters, including letters, digits, symbols, and control characters, that can be used to write programs. Initially, C used the ASCII (American Standard Code for Information Interchange) character set, which includes 128 characters mapped to specific numeric values. Extended character sets like UTF-8 and Unicode accommodate a broader range of characters. Understanding character sets in C programming is fundamental as it governs how characters are represented, stored, and manipulated in programs.

In this unit introduces and illustrate that C is a general-purpose programming language extensively used in games and web development, machine learning, and data mining applications. Generally, people think that high-level languages like Python, Java, and JavaScript have surpassed C in popularity and use in recent years. Still, C

Language applications are frequently utilized all around the globe. The understanding of programming is inadequate without the integration of the C language. Therefore, it tends to dominate the field of programming.

What you configure in programming is more important than what you know. With the technological world constantly changing, problem-solving is the only talent that allows you to manage advancements while also evolving. Begin with C, the language from which most modern programming languages are developed, to hone your fundamental programming skills and problem-solving talents. Despite being introduced 50 years ago, C is extensively used in almost every sector and is recognized as the finest language for beginners. This begs the question of what C is and why it is still so widely used.

The C programming language is a procedural language. It was designed by Dennis Ritchie as a system programming language for building operating systems. C language' slow-level memory access, minimal keyword set, and clear style make it ideal for system programming, such as operating system or compiler development. C soon established itself as a strong and dependable programming language, with some of the most well-known names remaining associated with it today. C is the programming language used to develop Microsoft Windows, Apple's OS X, and Symbian. Google's Chromium, MySQL, Oracle, and the bulk of Adobe's products all employ the C programming language. It is also vital in our daily lives, since most smart gadgets rely on it.

This unit will explain to you the functions as building blocks of the programming language C.

2.1 OBJECTIVES

After completing this unit, you will be able to:

- Designed to provide complete knowledge of C language applications.
- Help to create programs and applications in C.
- Help to understand the basic to advanced concepts related to Objective – C Programming languages.
- This unit includes a component that is intended to provide the learner with hands-on experience with the ideas.

2.2 CHARACTER SET OF C

Understanding character sets in C programming is fundamental as it governs how characters are represented, stored, and manipulated in programs. Here's an in-depth look at character sets in C:

Character Set in C Programming:

1. Basics of Character Representation:

i) **ASCII (American Standard Code for Information Interchange):** ASCII is one of the earliest and most widely used character encoding schemes. It represents characters using 7 bits (extended ASCII uses 8 bits) and includes control characters, uppercase and lowercase letters, digits, and special symbols.

Character	ASCII	Detail
!	33	Exclamation point or Exclamation mark
“	34	Inverted commas, quote marks or quotations
#	35	Hash, number, pound, octothorpe

$	36	Doller sign or generic currency
%	37	Percent
&	38	Ampersand or and
‘	39	Single quote or an apostrophe
(	40	Open or left parenthesis
)	41	Right or close parenthesis
*	42	Asterisk, often known as a star
+	43	Plus
,	44	Comma
-	45	Dash, hyphen or minus sign
.	46	Comma, dot or full stop
/	47	Forward slash, whack slash
:	58	Colon
;	59	Semicolon
<	60	Angle brackets for less than
=	61	Equal sign
>	62	Angle brackets for greater than
?	63	Inquiry mark
@	64	Asperand, at, or the at symbol
[	91	Enable brackets
\	92	Backslash
]	93	Open bracket
^	94	Circumflex or Caret
_	95	Underscore
‘	96	Open quotation, backquote
{	123	Open brace, curly bracket
}	125	Close brace, curly bracket
~	126	Tilde

ii). **Unicode:** Unicode is a standard for consistent encoding and representation of text across different languages and platforms. It uses variable bit lengths (UTF-8, UTF-16, UTF-32) to accommodate a vast array of characters from different languages and symbol sets.

2. Character Constants and Escape Sequences:

i) **Character Constants:** In C, characters can be represented using single quotes (''). For example, 'A', 'a', '1', or '$'. These constants are directly mapped to their ASCII or Unicode values.

ii). Escape Sequences: C also supports escape sequences (e.g., '\n', '\t', '\x') to represent special characters or non-printable characters within strings, enabling easy manipulation of characters in code.

Character	ASCII	Detail
<space>	32	Space
\t	9	Horizontal tab
\n	10	Newline
\v	11	Vertical tab
\f	12	Feed
\r	13	Carriage Return

3. ASCII and Extended ASCII:

i). ASCII Range: Standard ASCII includes characters in the range 0 to 127. Extended ASCII (using 8 bits) expands this range to include additional characters, symbols, and special characters.

ii). Platform-Specific Variations: Different platforms might have their own extended ASCII variations, leading to potential compatibility issues when moving code between systems.

4. Unicode and Multibyte Characters:

i). Unicode Encoding Schemes: UTF-8, UTF-16, and UTF-32 are encoding schemes used to represent Unicode characters. UTF-8 is widely used due to its compatibility with ASCII and variable-length encoding.

ii). Multibyte Characters: Characters in Unicode may span multiple bytes, especially in UTF-16 and UTF-32, leading to complexities in handling and manipulating multibyte characters in C programs.

5. Locale and Character Set Functions:

i). Locale-Specific Functions: C provides functions like `setlocale()` and `wctomb()` to handle locale-specific character sets and encoding conversions.

ii). Character Classification Functions: Functions like `isalpha()`, `isdigit()`, and `islower()` allow programmers to perform character-based operations based on character classes.

6. Wide Characters and Internationalization:

i). Wide Character Representation: C supports wide characters (`wchar_t`) to handle characters beyond the ASCII range, facilitating internationalization and localization efforts.

ii) L10n and I18n: Wide character support allows for localization (L10n) and internationalization (I18n) of software, enabling the display of text in different languages and character sets.

7. Challenges and Considerations:

i). Compatibility and Portability: Dealing with different character sets and encoding schemes can pose challenges, especially when writing code that needs to be portable across various systems and locales.

ii). Handling Multibyte Characters: Manipulating and processing multibyte characters requires careful handling to avoid unintended behavior or errors in character manipulation and string processing functions.

8. Best Practices and Recommendations:

i). Use Standard Functions: Utilize standard C library functions like `isalpha()`, `tolower()`, and `toupper()` for character manipulation to ensure portability and compatibility.

ii). Avoid Hard-Coding Values: Rely on character constants and escape sequences instead of hard-coding specific ASCII or Unicode values for improved readability and maintainability.

2.3 C TOKENS

Understanding tokens is fundamental in C programming as they form the building blocks of C code. Tokens represent the smallest individual units of a C program, aiding in syntax analysis and code interpretation. Here's an in-depth exploration of C tokens:

1. Definition and Types of Tokens:

i). Definition: Tokens in C programming are the smallest individual units constituting a C program. They include keywords, identifiers, constants, strings, operators, and special symbols.

ii). Types of Tokens: C tokens are categorized into keywords (e.g., `int`, `if`, `while`), identifiers (user-defined names), constants (numeric or character literals), strings (sequences of characters enclosed in double quotes), operators (arithmetic, relational, logical), and special symbols (punctuation characters).

2. Keywords and Identifiers:

i). Keywords: Keywords are reserved words with predefined

meanings in C. Examples include `if`, `else`, `for`, `int`, and `void`. They cannot be used as identifiers.

ii). Identifiers: Identifiers are user-defined names used to represent variables, functions, or other entities in a C program. They consist of letters, digits, and underscores, beginning with a letter or underscore.

3. Constants:

i). Numeric Constants: Numeric constants represent fixed numerical values, such as integers (`123`, `-45`), floating-point numbers (`3.14`, `0.75`), and scientific notation (`2.5e3`).

ii). Character Constants: Character constants are single characters enclosed in single quotes (`'A'`, `'5'`, `'%'`), representing their ASCII or Unicode values.

4. Strings and Escape Sequences:

i). String Constants: Strings are sequences of characters enclosed in double quotes (`"Hello, World!"`). They can contain alphanumeric characters, symbols, and escape sequences (`\n`, `\t`, `\"`) for special characters.

ii). Escape Sequences: Escape sequences are special combinations of characters that represent non-printable or special characters within strings.

5. Operators:

i). Arithmetic Operators: Operators such as `+`, `-`, `*`, `/`, `%` perform arithmetic operations on numeric values.

Operator	Purpose

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder after integer division

ii). Relational and Logical Operators: `==`, `!=`, `>`, `<`, `&&`, `||` perform comparisons and logical operations.

Operator	Meaning
==	equality
!=	not equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

iii). Assignment and Increment/Decrement Operators: `=`, `+=`, `-=`, `++`, `--` modify variable values.

6. Special Symbols:

Punctuation Characters: Special symbols include punctuation characters like parentheses `()`, braces `{}`, brackets `[]`, commas `,`, semicolons `;`, and colons `:` used to structure C code and define blocks, expressions, and statements.

7. Tokenization and Lexical Analysis:

i). Lexical Analysis: The process of breaking down a C program into tokens is called lexical analysis. Tokenization involves identifying tokens and categorizing them based on their types.

ii). Tokenization Tools: Lexical analyzers and tokenizers are used to scan source code, recognize tokens, and pass them to the parser for further syntactic analysis.

8. Role in Syntax Analysis:

Syntax Parsing: Tokens serve as the input for the parser during syntax analysis. The parser uses tokens to analyze the structure and syntax of the program based on predefined grammar rules.

2.4 C PROGRAMMING KEYWORDS

Understanding keywords is crucial in C programming as they are reserved words with predefined meanings and specific functionalities. Here's an in-depth exploration of keywords in C:

1. Definition and Importance:

i). Definition: Keywords are reserved words in C with predefined meanings and functionalities. They serve specific purposes within the language and cannot be used as identifiers or variable names.

ii). Importance: Keywords play a crucial role in defining the syntax, structure, and functionality of C programs. They facilitate control flow, data manipulation, and define the basic elements of the language.

2. Types of Keywords:

i). Primary Keywords: Primary keywords are fundamental to C programming and include words like `int`, `char`, `float`, `if`, `else`, `while`, `for`, `switch`, and `return`.

ii). Additional Keywords: C has a set of additional keywords introduced in various C standards (e.g., C99, C11), such as `_Bool`, `_Complex`, `_Imaginary`, and others, providing additional functionalities or data types.

3. Commonly Used Keywords:

i). Data Type Keywords: `int`, `char`, `float`, `double`, `void` define data types used for variable declaration and manipulation.

ii). Control Flow Keywords: `if`, `else`, `switch`, `case`, `default`, `while`, `for`, `do`, `break`, `continue` control the flow of execution within the program.

iii). Function Keywords: `return` specifies the value returned by a function to the calling code.

character	character constant consist of a single character, single digit, or a single special symbol enclosed within a pair of single inverted commas. i.e. ‘A’,’%’.
integer	An integer constant refers to a sequence of digits. There are. Three types of integers: decimal, octal, and hexadecimal. In octal notation, write (0) immediately before the octal representation. For example: 0.76, -076. In hexadecimal notation, the constant is preceded by 0x. Example: 0x3E, -0x3E. No commas or blanks are allowed in integer constants.
real	Real constants consist of three parts: Sign (+ or 0) , Number portion (base), exponent portion i.e., +.72, +72 , +7.6E+2, 24.3e-5
string	A string constant is a sequence of one or more characters enclosed within a pair of double quotes (" "). If a single character is enclosed within a pair of double quotes, it will also be interpreted as a string constant. Examples: "Welcome to Microtek \ n” , “a” , “123”
logical	A logical constant can have either a true value or a false value. In 'C all the non-zero values are treated as true value while 0 is treated as false.

iv). Storage Class Keywords: `auto`, `extern`, `static`, `register` determine the storage duration and scope of variables.

4. Reserved Status and Restrictions:

i). Reserved Status: Keywords are reserved by the language and cannot be used as identifiers, function names, or variable names within the code.

ii). Case Sensitivity: Keywords are case-sensitive in C. For instance, `int` is a keyword, but `Int`, `INT`, or `iNt` are not recognized as keywords and can be used as identifiers.

5. Evolving Keyword Set:

i). Standardization and Updates: C standards (e.g., C89, C99, C11) introduce new keywords or modify the behavior of existing keywords to enhance the language's capabilities.

ii). Backward Compatibility: C standards strive to maintain backward compatibility, ensuring that code written in older versions of C remains valid in newer versions.

6. Vendor-Specific Extensions:

Vendor-Specific Keywords: Some C compilers introduce vendor-specific keywords or extensions to enhance functionality or optimize code for specific platforms. These keywords might not be standard across different compilers.

7. Best Practices and Usage:

i). Avoiding Keyword Conflicts: Developers should avoid using keywords as variable names or identifiers to prevent conflicts and maintain code readability.

ii). Consistent Use: Adhering to consistent naming conventions and avoiding ambiguous identifiers or names that resemble keywords ensures code clarity and avoids potential errors.

8. Role in Program Structure:

Defining Structure: Keywords play a vital role in defining the structure of C programs, delineating functions, control flow, variable types, and other essential elements of the language.

Absolutely, let's delve deeper into the significance and

categorization of keywords in C programming:

Keywords in C Programming: Core Building Blocks

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

1. Significance of Keywords:

i). Precise Functionality: Keywords serve specific purposes and have well-defined functionalities within the language. They determine how variables are declared, how control structures function, and how data types are handled.

ii). Syntax Definition: Keywords form the foundation of C's syntax, enabling the creation of robust and structured programs by providing essential components for defining operations and control flow.

2. Categorization of Keywords:

i). Data Type Keywords: Keywords like `int`, `char`, `float`, `double`, and `void` specify the type of data a variable can hold, defining its size, storage, and operations.

ii). Control Flow Keywords: These keywords govern the execution flow within a program, including conditional statements (`if`, `else`), looping (`while`, `for`, `do`), and branching (`switch`, `case`).

iii). Storage Class Keywords: Keywords such as `auto`, `static`, `extern`, and `register` define the storage duration, scope, and visibility of variables.

iv). Function Keywords: `return` specifies the value returned by a function to the calling code.

3. Evolving Nature of Keywords:

i). Standard Evolution: Newer C standards introduce additional keywords or modify the behavior of existing keywords to improve language capabilities, enhance expressiveness, or introduce new functionalities.

ii). C11 Additions: C11 introduced keywords like `_Bool` for boolean types, `_Noreturn` to indicate that a function does not return, and `_Thread_local` for thread-local storage duration.

4. Special Uses of Keywords:

**i). Sizeof` Operator:** While not a keyword, `sizeof` is a special operator used to determine the size of a data type or variable in bytes.

**ii). Typedef` Keyword:** `typedef` is used to create new data type names, improving code readability and abstraction.

5. Avoiding Keyword Misuse:

i). Identifier Naming Conventions: Following consistent naming conventions helps prevent accidentally using keywords as

identifiers or variable names, reducing potential conflicts and errors.

ii). Compiler Warnings: Modern compilers often provide warnings when keywords are used inappropriately, alerting developers to potential issues.

6. Platform and Compiler Variations:

i). Platform-Dependent Keywords: Some keywords might behave differently or have platform-specific implementations based on the C compiler or the target platform.

ii). Standard Compliance: Different compilers may implement different versions of the C standard, impacting the availability or behavior of certain keywords.

7. Extensibility and Custom Keywords:

Extensibility via Macros: In certain cases, developers use preprocessor macros or naming conventions to emulate keyword-like behavior, creating custom functionalities or defining macros to aid in programming tasks.

8. Interplay with Syntax and Program Logic:

i). Syntax Integrity: Keywords provide a framework for ensuring syntax integrity, guiding the compiler in parsing and understanding the structure and flow of C programs.

ii). Logical Flow Control: The strategic use of control flow keywords helps in designing logical program structures, enabling complex decision-making and looping functionalities.

2.5 IDENTIFIERS OF THE C

C identifiers represent the name in the C program, for example, variables, functions, arrays, structures, unions, labels, etc. An identifier can be composed of letters such as uppercase, lowercase letters, underscore, and digits, but the starting letter should be either an alphabet or an underscore. If the identifier is not used in the external linkage, then it is called as an internal identifier. If the identifier is used in the external linkage, then it is called as an external identifier.

We can say that an identifier is a collection of alphanumeric characters that begins either with an alphabetical character or an underscore, which are used to represent various programming elements such as variables, functions, arrays, structures, unions, labels, etc. There are 52 alphabetical characters (uppercase and lowercase), underscore character, and ten numerical digits (0-9) that represent the identifiers. There is a total of 63 alphanumerical characters that represent the identifiers.

Rules for constructing C identifiers

- The first character of an identifier should be either an alphabet or an underscore, and then it can be followed by any of the characters, digits, or underscores.
- It should not begin with any numerical digit.
- In identifiers, both uppercase and lowercase letters are distinct. Therefore, we can say that identifiers are case sensitive.
- Commas or blank spaces cannot be specified within an identifier.
- Keywords cannot be represented as an identifier.
- The length of the identifiers should not be more than 31 characters.
- Identifiers should be written in such a way that it is meaningful, short, and easy to read.

Example of Identifiers:

Total, sum, average, _m_, sum_1, etc.

Example of Invalid Identifiers:

2sum (Starts with a numerical digit)

Types of identifiers

- Internal identifier
- External identifier

Internal Identifier

If the identifier is not used in the external linkage, then it is known as an internal identifier. The internal identifiers can be local variables.

External Identifier

If the identifier is used in the external linkage, then it is known as an external identifier. The external identifiers can be functioning names, global variables.

Differences between Keyword and Identifier

Keyword	Identifier
Keyword is a pre-defined word.	The identifier is a user-defined word.
It must be written in a lowercase letter.	It can be written in both lowercase and uppercase letter.
Its meaning is pre-defined in the c compiler.	Its meaning is not defined in the c compiler.
It is a combination of alphabetical characters.	It is a combination of alphanumeric characters.
It does not contain the underscore character.	It can contain the underscore character.

2.6 CONCLUSION

Character sets in C programming are crucial for representing, storing, and processing characters within programs. Understanding character encoding schemes, escape sequences, handling wide characters, and addressing localization challenges are essential for writing robust and portable C programs that effectively handle diverse character sets and language requirements. ASCII, the fundamental character encoding standard, provided a base for representing text characters in early computing. Unicode, a comprehensive standard, allows for the representation of a wide range of characters from different languages and scripts. Multibyte characters, particularly in UTF-8, enable the representation of characters beyond the ASCII range, facilitating multilingual support and text processing in modern computing environments. Locale functions allow C programs to adapt to users' language and cultural preferences, affecting formatting and text handling.

Character set functions enable proper manipulation and classification of characters based on locale-specific rules, which is crucial for multilingual text processing. Internationalization is the foundational step in making software adaptable to diverse languages and cultures, enabling global reach and usability.

UNIT 3 APPLICATION AND INFORMATION

Structure

3.0 Introduction

3.1 Objectives

3.2 Applications of C

- 3.2.1 Operating Systems
- 3.2.2 Graphical User Interface
- 3.2.3 Embedded Systems
- 3.2.4 Google
- 3.2.5 Design of a Compiler
- 3.2.6 Mozilla Firefox and Thunderbird
- 3.2.7 Gaming and Animations
- 3.2.8 MySQL

3.3 New Programming Language Platforms

3.4 Translators of high-level languages into machine language

3.5 C Programming Projects

- 3.5.1 Basic C Projects
- 3.5.2 Intermediate C Projects with Source Code
- 3.5.3 Advanced C Projects with Source Code

3.6 Conclusion

3.7 Unit based Questions /Answers

3.0 INTRODUCTION

In this unit introduces illustrate that C is a general-purpose programming language extensively used in games and web development, machine learning, and data mining applications. Generally, people think that high-level languages like Python,

Java, and JavaScript have surpassed C in popularity and use in recent years. Still, C Language applications are frequently utilized all around the globe. The understanding of programming is inadequate without the integration of the C language. Therefore, it tends to dominate the field of programming.

What you can figure out in programming is more important than what you know. With the technological world constantly changing, problem-solving is the only talent that allows you to manage advancements while also evolving. Begin with C, the language from which most modern programming languages are developed, to hone your fundamental programming skills and problem-solving talents. Despite being introduced 50 years ago, C is extensively used in almost every sector and is recognized as the finest language for beginners. This begs the question of what C is and why it is still so widely used.

The C programming language is a procedural language. It was designed by Dennis Ritchie as a system programming language for building operating systems. C language's low-level memory access, minimal keyword set, and clear style make it ideal for system programming, such as operating system or compiler development. C soon established itself as a strong and dependable programming language, with some of the most well-known names remaining associated with it today. C is the programming language used to develop Microsoft Windows, Apple's OS X, and Symbian. Google's Chromium, MySQL, Oracle, and the bulk of Adobe's products all employ the C programming language. It is also vital in our daily lives, since most smart gadgets rely on it.

This unit will explain to you the applications and information of language C.

3.1 OBJECTIVES

After completing this unit, you will be able to:

- Designed to provide complete knowledge of C language applications.
- Help to create programs and applications in C.
- Help to understand the basic to advanced concepts related to Objective-C Programming languages.
- This unit includes a component that is intended to provide the learner with hands-on experience with the ideas.

3.2 APPLICATIONS OF C LANGUAGE

The development of system software and desktop applications is mostly accomplished via the use of C programming. The following are some examples of C programming applications.

3.2.1 Operating Systems

A high-level programming language built in the C programming language was used to construct the first operating system, which was UNIX. Later on, the C programming language was used to write Microsoft Windows and several Android apps.

the influence of C in operating systems (OS) is immense. Here's an exploration of how the C programming language is integral to the development and functionality of operating systems:

The Role of C in Operating Systems:

C programming language holds a pivotal role in the creation, maintenance, and evolution of operating systems across diverse computing environments. Its characteristics of efficiency, portability, and close-to-hardware capabilities make it the language of choice for OS development. The following are key areas where C is extensively used within operating systems:

1. **Kernel Development:**

a) **Low-Level System Interaction:** C's ability to interact directly with hardware and manage system resources efficiently makes it the language of choice for kernel development. It allows developers to write code that deals with memory management, process scheduling, interrupt handling, and device drivers.

b) **Portability:** C's portability enables developers to write OS kernels that can be easily adapted to different hardware architectures.

2. **Device Drivers:**

Hardware Interaction: C is crucial in writing device drivers, which are essential for enabling communication between hardware devices and the operating system. Its ability to access and control hardware directly is vital for efficient device operations.

3. **System Utilities:**

a) **Command-Line Tools:** Many system utilities, such as command-line interfaces and system administration tools, are

written in C due to its efficiency and ability to manage system resources effectively.

b) **File Systems:** C is commonly used in developing file system utilities and functionalities like file I/O operations, directory management, and file permissions.

4. Portability and Efficiency:

a). **Platform Independence:** C's portability allows operating systems written in C to be compiled and run on different hardware platforms with minimal modifications, making it a preferred choice for cross-platform development.

b). **Efficient Resource Utilization:** The language's efficiency in managing memory and system resources contributes significantly to the overall performance of the operating system.

5. System Libraries and APIs:

Standard C Library: Operating systems often provide a standard C library that offers a set of functions and utilities for application developers. These libraries are typically written in C and provide essential functionalities like memory allocation, input/output operations, and string manipulation.

6. OS Maintenance and Evolution:

Ease of Maintenance: C's structured and modular nature simplifies the maintenance and enhancement of existing operating systems. New features and improvements can be efficiently added to the codebase without compromising the system's stability.

7. **Operating System Research and Development:**

Experimentation and Prototyping: C is commonly used in academia and research institutions for experimenting with new OS concepts, prototyping new features, and understanding the intricacies of OS internals due to its clarity and simplicity.

3.2.2 GUI (Graphical User Interface)

Since the beginning of time, Adobe Photoshop has been one of the most widely used picture editors. It was created entirely with the aid of the C programming language. Furthermore, C was used to develop Adobe Illustrator and Adobe Premiere.

The C programming language, known for its efficiency and versatility, also plays a crucial role in the development of Graphical User Interfaces (GUIs). Here's an exploration of how C is applied in this domain:

Applications of C in Graphical User Interfaces:

Graphical User Interfaces (GUIs) have become an integral part of modern software applications, providing users with visually interactive and user-friendly experiences. While languages like Java and Python are popular for GUI development, C has its own applications, particularly in scenarios where performance and system-level interactions are critical.

a). Window Management:

Windowing Systems: C is often used to develop window management systems that handle the creation, manipulation, and display of windows on the screen. These systems are fundamental components of GUIs, allowing users to interact with multiple applications simultaneously.

b). Widget Toolkits:

Creation of GUI Elements: C is employed in developing widget toolkits, which are libraries of graphical components (widgets) such as buttons, textboxes, and sliders. These toolkits provide the building blocks for constructing GUIs and are often written in C for efficiency and performance.

c). Custom Controls and Graphics:

Efficient Rendering: C's low-level capabilities make it suitable for efficient rendering of graphics and custom controls. It allows developers to have precise control over the graphical elements and optimize performance for resource-intensive applications like video editing software or games.

d). Cross-Platform GUI Development:

Portability: C, with its portability, enables the creation of cross-platform GUI applications. The same C codebase can be compiled for different operating systems, reducing development efforts and ensuring a consistent user experience across platforms.

e). Embedded Systems GUIs:

Resource Constraints:

In embedded systems where resources are limited, C is often preferred for GUI development due to its efficient use of system resources. Customized graphical interfaces on devices like medical equipment, industrial control systems, and IoT devices can be efficiently implemented using C.

f). Application Interfaces:

Integration with System APIs:

C is used to create GUI applications that seamlessly integrate with system-level APIs. This is crucial for applications that require direct interaction with the underlying operating system for tasks such as file management, process control, and system configuration.

g). Performance-Critical Applications:

Graphics-Intensive Software:

In applications demanding high performance, such as graphic design software or 3D modeling tools, C is chosen for its ability to optimize code and efficiently handle complex calculations and graphics rendering.

h). Game Development:

Game User Interfaces: C is frequently used in the development of game user interfaces, where responsiveness and efficiency are critical. Game menus, heads-up displays (HUDs), and interactive

elements are often implemented in C to ensure a smooth gaming experience.

i). Integration with Hardware:

Device Interaction: C's capability to interact closely with hardware is beneficial for GUI applications that involve communication with peripherals and external devices. This is particularly relevant in industrial control systems and scientific instruments.

3.2.3 Embedded Systems

Because it is directly related to the machine hardware, C programming is often regarded as the best choice for scripting programs and drivers for embedded systems, among other things.

Embedded systems, found in a wide array of devices from consumer electronics to industrial machinery, rely heavily on the C programming language due to its efficiency, close-to-hardware capabilities, and portability. Here's an exploration of how C is applied in embedded systems:

a). Applications of C in Embedded Systems:

Embedded systems, characterized by their specialized functionalities and limited resources, often rely on the efficiency and control offered by the C programming language. These systems, deeply integrated into various devices and machinery, benefit from C in several key areas:

i). Device Control and Drivers:

Hardware Interaction: C's ability to directly access and control hardware resources makes it ideal for writing device drivers and interfacing with sensors, actuators, and other peripherals commonly found in embedded systems.

ii). Real-time Control: Embedded systems often require real-time control of hardware components, and C's ability to manage timing and low-level operations efficiently is crucial in such scenarios.

b). System Boot-up and Initialization:

Bootstrap Code: C is used in writing boot code that initializes the embedded system's hardware components during the startup process. This includes configuring memory, setting up interrupts, and initializing essential system components.

c). Firmware Development:

i). Efficient Code Execution: C's efficiency in utilizing system resources allows developers to create firmware that operates within the limited memory and processing power available in embedded systems.

ii). Portability: C's portability enables firmware written in C to be easily adapted to different hardware architectures, allowing for broader use across various devices.

d). Real-time and Control Systems:

i). Real-time Operations: C is employed in developing control systems that require precise timing and responsiveness, such as in

automotive systems (engine control units), robotics, and industrial automation.

ii).Predictable Performance: The deterministic behavior of C allows developers to predict and control system responses, critical in safety-critical applications.

e). Internet of Things (IoT) Devices:

i). Resource Optimization: C's ability to manage resources efficiently is valuable in IoT devices where power consumption and memory utilization need to be optimized for extended battery life and cost-effectiveness.

ii). Sensor Data Processing: C is used to process sensor data and control communication protocols in IoT devices, enabling them to interact with other devices and the internet.

f). Communication Protocols:

i). Low-level Networking: C is utilized in implementing communication protocols such as Bluetooth, Wi-Fi, and Ethernet, allowing embedded systems to connect and communicate with other devices or networks.

ii).Peripheral Communication: C facilitates interaction with various communication interfaces like SPI, I2C, and UART, enabling data exchange with external devices or modules.

g). Industrial Automation and Control Systems:

i). Reliability and Stability: C's ability to produce code that is

stable and reliable is critical in industrial automation systems where precision and consistent performance are paramount.

ii).Customization: C allows for the development of customized control systems tailored to specific industrial applications, providing flexibility and adaptability.

h). Automotive and Aerospace Systems:

Safety-Critical Applications: C is used in safety-critical systems in automotive and aerospace industries due to its ability to produce predictable and reliable code, essential in ensuring the safety and functionality of these systems.

3.2.4 Google

You can also use the C/C++ programming language to create the Google Chrome web browser and the Google File System. Furthermore, the Google Open Source community includes many projects that are maintained with the aid of the C/C++ programming language.

When it comes to Google, while the primary languages used in their vast infrastructure might not be directly C due to the complexity and scale, C has had a significant influence on the technologies and projects developed by Google. Here's a look at some areas where C has played a role in Google's ecosystem:

Applications of C in Google's Technology Landscape:

Google, renowned for its innovative technologies and services, relies on a diverse array of programming languages and tools to power its platforms and services. While languages like Java,

Python, and Go are prevalent within Google's infrastructure, C has influenced various aspects:

a) **Systems Programming and Performance-Critical Components:**

i). Low-Level Infrastructure: While not the primary language in Google's services, C is utilized in critical systems programming, particularly in performance-critical components of infrastructure, such as parts of the Google File System (GFS) or certain elements of the networking stack.

ii). Optimized Code: In cases where efficiency and performance are paramount, C's ability to produce optimized code is invaluable, influencing specific parts of Google's core infrastructure.

b) **Open Source Projects:**

Contributions to Open Source: Google has contributed to various open source projects written in C. For instance, they've supported and contributed to projects like the Linux kernel, enhancing its functionality and performance, which indirectly benefits Google's infrastructure.

c). **Embedded and Hardware Projects:**

IoT and Hardware Development: While not always directly associated with Google's primary services, C has applications in Google's endeavors related to embedded systems, IoT, and hardware development. Projects like Android Things or hardware-specific optimizations might involve C programming for firmware and low-level hardware interactions.

d). Experimentation and Prototyping:

Research and Development: C might be utilized in Google's research and development efforts, especially in experimental projects exploring new technologies, algorithms, or prototypes where performance at a lower level is essential.

e). Legacy Systems and Optimization:

i). Legacy Codebases:* In certain legacy systems that have been part of Google's infrastructure for a long time, there might still be components written in C, especially where rewriting or migrating the code might not be immediately feasible or beneficial.

ii).Performance Optimization: C might be employed in optimizing certain critical algorithms or functionalities within Google's services to ensure they operate at their peak performance levels.

3.2.5 Design of a Compiler

You can widely use the C programming language to develop compilers, one of its most popular applications. Many other languages' compilers were created with the connection between C and low-level languages in mind, making it easier for the machine to grasp what was being written. Many prominent compilers, such as Clang C, Bloodshed Dev-C, Apple C, and MINGW, were developed with the C programming language.

The development of a compiler involves intricate processes that necessitate efficient handling of syntax, semantics, and code generation. C, known for its system-level access and efficiency, is

frequently used in the creation of compilers. Here's an exploration of how C is applied in compiler design:

Applications of C in Compiler Design:

Compilers, vital in translating high-level programming languages into machine-readable code, require a meticulous design process. C, with its system-level capabilities and efficiency, is often employed in various aspects of compiler development:

1. Lexical Analysis (Lexers):

Tokenization: C's capability to handle strings and characters efficiently is crucial in building lexers. Lexical analyzers written in C break down source code into tokens, identifying keywords, identifiers, literals, and symbols.

2. Syntax Analysis (Parsers):

Parsing Algorithms: C is utilized in implementing parsers that enforce the grammatical structure of programming languages. Tools like Bison or Yacc generate C code for parsers, converting context-free grammars into code structures for syntactic analysis.

3. Semantic Analysis:

i). Type Checking: C's ability to manipulate memory and data structures is beneficial in implementing type systems and performing type checking during the semantic analysis phase of compilation.

ii). Error Detection: C aids in implementing checks for semantic

errors, ensuring the correctness of the code being compiled.

4. Intermediate Code Generation:

Code Representation: C is used to generate intermediate code representations of source programs. This involves constructing abstract syntax trees (ASTs) or intermediate code representations that act as a bridge between the source and target code.

5. Optimization Phase:

Efficient Algorithms: C's efficiency in managing memory and processing power is crucial in implementing optimization algorithms. Compilers written in C perform various optimizations like dead code elimination, loop optimizations, and constant folding to enhance program efficiency.

6. Code Generation:

Target Machine Independence: C allows for the generation of target-independent code. Compiler backends written in C produce machine code or assembly language specific to the target architecture while abstracting the complexities of hardware interactions.

7. Integration with System Libraries:

Utilization of Standard C Library: Compiler implementations often rely on the standard C library to perform various tasks, such as memory allocation, input/output operations, and string manipulation.

8. Error Handling and Reporting:

Diagnostic Messages: C is used to implement error handling mechanisms, generating informative diagnostic messages during compilation, aiding developers in debugging their code.

9. Portability and Maintenance:

Modular Design: C's structured nature allows for modular and maintainable compiler codebases. This facilitates easier enhancements, bug fixes, and porting the compiler to different platforms.

3.2.6 Mozilla Firefox and Thunderbird

Because Mozilla Firefox and Thunderbird were free and open-source email client projects, they were included here. As a result, they were developed in the C/C++ programming language.

Mozilla Firefox and Thunderbird, as flagship products of the Mozilla Foundation, heavily rely on various programming languages, including C, for their development. While these applications predominantly use a mix of languages for different components, C plays a significant role in their core functionalities and system-level interactions:

Applications of C in Mozilla Firefox and Thunderbird:

Mozilla Firefox, a widely used web browser, and Thunderbird, an email client, are developed by the Mozilla Foundation. While these applications use multiple languages, C is instrumental in several key areas:

1. Core Engine Development:

Firefox's Gecko Engine: C is used extensively in the development of Gecko, the rendering engine that powers Firefox. Gecko handles the display and interpretation of web content, requiring efficient handling of HTML, CSS, and JavaScript, which C helps facilitate at a low-level.

2. Performance-Critical Components:

i). Optimization: C is crucial in optimizing critical components of Firefox and Thunderbird for performance, ensuring smooth and responsive user experiences while rendering web pages or managing email data.

ii). System Resource Management: C's ability to manage system resources efficiently is valuable in handling memory and processor usage within these applications.

3. Platform-Specific Implementations:

Cross-Platform Compatibility: While Firefox and Thunderbird are designed to work across different operating systems, C aids in creating platform-specific implementations for Windows, macOS, and Linux, allowing for consistent functionality across diverse environments.

4. Browser and Email Client Interactions:

System Integration: C is used in interfacing with system libraries and APIs, allowing Firefox and Thunderbird to interact with the underlying operating system for functionalities like file I/O,

networking, and user interface interactions.

5. Extension and Add-on Development:

SDKs and APIs: Mozilla provides SDKs and APIs for developers to create extensions and add-ons for Firefox and Thunderbird. C may be involved in the core components of these SDKs, enabling developers to extend the functionalities of these applications.

6. Media Handling and Processing:

Audio/Video Support: C plays a role in handling audio and video codecs, ensuring compatibility and efficient playback of multimedia content within these applications.

7. Security and Memory Management:

Memory Security: C's low-level capabilities are essential in managing memory securely, contributing to the overall security and stability of Firefox and Thunderbird against vulnerabilities like buffer overflows.

8. Browser Engine Architecture:

Gecko Components: Gecko, being a core component of Firefox, relies on C for its architecture, allowing efficient handling of web content and providing the backbone for browser functionalities.

3.2.7 Gaming and Animation

Because the C programming language is based on a compiler and is thus far quicker than Python or Java, it has gained popularity in

the game industry. Some of the most basic games, such as the Dino game, Tic-Tac-Toe, and the Snake game, are written in C programming languages. In addition, doom3, a first-person shooter horror game developed by id Software in 2004 for Microsoft Windows and written in C, is one of the most powerful graphics games ever created.

C programming language holds a significant position in the realm of gaming and animations due to its performance, system-level access, and ability to interact closely with hardware. Here's an exploration of how C is applied in the gaming and animation industries:

Applications of C in Gaming and Animations:

C serves as a cornerstone in gaming and animations, facilitating the creation of immersive experiences through its efficiency, direct hardware interaction, and suitability for performance-critical tasks:

1. Game Engines and Development:

i). Core Game Logic: C is extensively used in game engines to handle the core game logic, ensuring smooth gameplay by efficiently managing game states, physics, and AI computations.

ii). Graphics Programming: C, coupled with graphics libraries like OpenGL and DirectX, powers rendering engines, enabling the creation of visually stunning graphics and effects in games and animations.

2. Real-Time Rendering and Performance:

i). Graphics Optimization: C's low-level capabilities are crucial in optimizing graphics rendering pipelines, ensuring real-time performance in displaying complex scenes, textures, and animations.

ii). Efficient Memory Management: C's control over memory allocation and management helps in optimizing resource usage, which is crucial in graphics-intensive applications.

3. Animation and Simulation:

Animation Frameworks: C is employed in animation frameworks and tools for creating lifelike character animations and scene movements, providing the backbone for animation software.

Physics Simulation: C aids in implementing physics engines used for simulating realistic interactions between objects, characters, and environments in games and animations.

4. Cross-Platform Development:

Platform Independence: C's portability allows game developers to write code that can be compiled across multiple platforms, facilitating cross-platform game releases for various operating systems and devices.

5. Game AI and Scripting:

i). AI Implementation: C is used to develop artificial intelligence algorithms for game characters, providing them with behaviors,

decision-making abilities, and interactive responses.

ii). Scripting Engines: In game development, C is utilized to create scripting engines that allow game designers to implement dynamic behaviors and game mechanics without recompiling the entire game codebase.

6. Embedded Systems and Consoles:

i). Console Game Development: C is prevalent in developing games for gaming consoles due to its performance optimization and ability to harness the hardware capabilities of consoles like PlayStation, Xbox, and Nintendo.

ii). IoT and Embedded Gaming: C's efficiency is leveraged in developing games for embedded systems and IoT devices, catering to gaming experiences on a diverse range of devices with limited resources.

7. Tools and Middleware Development:

Game Development Tools: C is used to create development tools, middleware, and APIs that assist game developers in optimizing performance, debugging, and creating game assets.

8. Custom Hardware Interactions:

Specialized Peripherals: In applications involving specialized gaming peripherals or hardware-specific interactions, C enables direct communication with hardware, allowing developers to create tailored experiences.

3.2.8 MySQL

MySQL is another open-source project that is used in relational database management systems (RDBMS). It was developed in the C/C++ programming language.

MySQL, an open-source relational database management system, extensively uses C and C++ in its development for various critical components. Here's an exploration of how C is applied in MySQL:

Applications of C in MySQL:

MySQL, a popular RDBMS known for its reliability, performance, and scalability, leverages the C programming language in several key areas:

1. Core Database Engine:

C Codebase: The core of MySQL's database engine is primarily written in C. This includes fundamental functionalities like query parsing, query optimization, data manipulation, indexing, and transaction handling.

2. Performance Optimization:

i). Efficient Algorithms: C allows developers to implement high-performance algorithms for data storage, retrieval, and processing, ensuring the database engine operates swiftly even with large datasets.

ii). Memory Management: C's control over memory allocation and management is crucial in optimizing MySQL's memory usage,

leading to better performance and reduced overhead.

3. System Interaction and Portability:

System-Level Interactions: C enables MySQL to interact closely with the underlying operating system, facilitating efficient I/O operations, process management, and system calls.

Platform Independence: While MySQL supports various operating systems, C allows for the development of a codebase that can be compiled and run across different platforms with minimal modifications.

4. Custom Extensions and Plugins:

Plugin Architecture: MySQL's plugin architecture, enabling the development of custom extensions, storage engines, and functionalities, is often implemented in C to ensure compatibility and performance.

5. Database Drivers and Connectors:

Native Drivers: C is used in developing native drivers and connectors (e.g., C API) that enable various programming languages and applications to interact seamlessly with the MySQL database.

6. Memory Management and Optimization:

Resource Utilization: C's efficiency in managing resources aids in optimizing memory allocation and utilization within the database system, contributing to improved performance and stability.

7. Query Optimization and Execution:

Query Processing: C's capabilities are harnessed in optimizing and executing SQL queries efficiently, ensuring that complex queries are processed swiftly and accurately.

8. Security and Stability:

Codebase Robustness: C's structured nature assists in creating a robust and secure codebase for MySQL, enhancing the database's stability and resistance to vulnerabilities.

9. Open-Source Contributions:

Community Development: MySQL being an open-source project welcomes contributions from developers worldwide, many of whom contribute in C to enhance and extend its functionalities.

3.3 NEW PROGRAMMING LANGUAGE PLATFORMS

It is not only C that gave rise to C++. This programming language incorporates all the features of C while also incorporating the concept of object-oriented programming. Still, it has also given rise to many other programming languages widely used in today's world, such as MATLAB and Mathematica. It makes it possible for applications to run more quickly on a computer.

New programming language platforms are constantly emerging, aiming to address modern challenges, enhance developer productivity, and cater to evolving computing paradigms. Here's a

look at some of the trends and new platforms in the programming language landscape:

1. Rust:

i). Safety and Performance: Rust has gained attention for its emphasis on memory safety without sacrificing performance. It offers a strong type system and ownership model, enabling safer concurrent programming.

ii). Systems Programming: Rust is favored for system-level programming, replacing languages like C/C++ in certain scenarios due to its safety guarantees and efficiency.

2. Swift:

i). IOS Development: Swift, developed by Apple, has become the primary language for IOS and macOS app development. Known for its readability and modern syntax, it aims to make programming for Apple platforms more accessible.

ii). Server-Side Development: Swift is also making headway in server-side development, offering a concise and powerful language for building web applications.

3. Kotlin:

i). Android Development: Kotlin has gained popularity as a modern language for Android app development. Its interoperability with Java and concise syntax has led to increased adoption within the Android development community.

ii). Multiplatform Development: Kotlin's multiplatform capabilities enable developers to write shared code across multiple platforms, including JVM, Android, iOS, and web.

4. Web Assembly (Wasm):

i). Universal Binary Format: Web Assembly is not a programming language but a binary instruction format for a stack-based virtual machine. It enables running code written in multiple languages on web browsers at near-native speeds.

ii). Cross-Platform Execution: Wasm allows code written in languages like C/C++, Rust, and others to be compiled and executed across various platforms beyond the web, including edge computing and IoT devices.

5. Julia:

i). Scientific Computing: Julia has gained traction in scientific and numerical computing due to its high performance, easy syntax, and powerful features for mathematical modeling, data analysis, and machine learning.

ii).Parallelism and Concurrency: Julia's design includes built-in support for parallelism and concurrent programming, making it suitable for computationally intensive tasks.

6. Golang (Go):

i). Concurrent Programming: Go has gained popularity for its simplicity and built-in support for concurrent programming through goroutines and channels. It's well-suited for building

scalable and concurrent systems.

ii).Cloud-Native Development: Go is commonly used in cloud-native development for its efficiency in creating microservices and distributed systems.

7. Elixir:

i). Functional Programming: Elixir, built on the Erlang VM, combines the functional programming paradigm with a focus on fault-tolerance and concurrency. It's used in building highly scalable and fault-tolerant systems.

ii). Real-Time Applications: Elixir is favoured for real-time applications, such as messaging platforms and IoT systems, where high concurrency and reliability are essential.

8. Haskell and Functional Languages:

Functional Programming Paradigm: Haskell and other functional languages continue to be influential due to their emphasis on immutability, higher-order functions, and type safety. They're used in academia, research, and niche domains requiring strong guarantees.

9. Blockchain and Smart Contracts:

Smart Contract Development: Solidity, for Ethereum, and other domain-specific languages (DSLs) are used in writing smart contracts for blockchain platforms, facilitating decentralized applications (dApps) and decentralized finance (DeFi).

10. Low-Code/No-Code Platforms:

Simplifying Development: Low-code/no-code platforms like Bubble, Out Systems, and others aim to democratize software development, allowing users with limited coding experience to create applications using visual interfaces and pre-built components.

3.4 TRANSLATORS OF HIGH-LEVEL LANGUAGES INTO MACHINE LANGUAGE

Interpreters are also computer programs that are used to translate high-level languages into machine language. You may write language interpreters in the C programming language. C language is used to write several computer language interpreters, such as the Python Interpreter, the MATLAB Interpreter, etc.

C plays a crucial role in the development of translators, such as compilers and interpreters, responsible for translating high-level languages into machine-readable code. Here's an exploration of how C is applied in creating these essential language translators:

Applications of C in Translators of High-Level Languages:

Translators, including compilers and interpreters, are instrumental in converting human-readable high-level code into machine-executable instructions. C programming language is widely used in various aspects of developing these language translators:

1. Compiler Development:

i). Frontend Processing: C is utilized in building the frontend of compilers responsible for lexical analysis (scanning), syntax

analysis (parsing), and semantic analysis of the source code.

ii). Intermediate Code Generation: Compilers often use C for generating intermediate representations of code (e.g., abstract syntax trees - ASTs) before translating them into target machine code.

2. Interpreter Implementation:

i). Interpreter Loop: C is employed in constructing interpreter loops, which execute high-level code directly without prior translation to machine code, interpreting and executing instructions on-the-fly.

ii). Runtime Environment: Interpreters written in C establish the runtime environment required to execute high-level language instructions efficiently.

3. Code Optimization and Generation:

i). Backend Processing: C aids in the backend of compilers, where it performs code optimization and translation of intermediate representations into target machine code or bytecode.

ii). Target-Dependent Optimizations: Compilers written in C often implement optimizations specific to the target architecture, leveraging C's ability to interact closely with hardware.

4. Development Tools and Utilities:

i). Language-Specific Tools: C is used in creating language-specific tools and utilities for debugging, profiling, and analyzing high-level code during translation processes.

ii). Parser Generators: Tools like Bison or Yacc generate C code

for parsers, enabling the implementation of parsers for various high-level languages.

5. Portability and Cross-Language Support:

i). Portability: C's portability allows for the development of translators that can be compiled and executed across different platforms, facilitating cross-platform language translation.

ii). Multilingual Translators: C-based translators can handle multiple high-level languages, allowing developers to create compilers or interpreters that support diverse programming languages.

6. Frameworks and Libraries:

Compiler Frameworks: C provides the groundwork for developing frameworks and libraries that aid in building translators, offering reusable components for lexical analysis, parsing, and code generation.

7. Low-Level System Interactions:

System Calls and Hardware Interaction: C's capability to interact closely with the underlying system and hardware resources is valuable in translators needing low-level access for optimization or system-specific functionality.

8. Open Source Contributions:

Community Development: Many open-source compilers, interpreters, and related tools are written in C, fostering contributions from a global community of developers to enhance and extend these language translators.

3.5 C PROGRAMMING PROJECTS

C programming projects are programs or tools that generate, plan, and manage various tasks or applications using the C programming language. Projects can assist you in learning and practicing C skills like file handling, command line parsing, and make files. Projects can also combine many applications into a single executable or library. C projects may be created for a variety of objectives, including financial management, artistic work, and teaching.

3.5.1 Basic C Projects

There are some basic C Projects which can be simply build:

a). Calculator:

You can build a simple calculator with C using switch cases or if-else statements. This calculator takes two operands and an arithmetic operator (+, -, *, /) from the user, however, you can expand the program to accept more than two operands and one operator by adding logic. Then, based on the operator entered by the user, it conducts the computation on the two operands. The input, however, must be in the format “number1 operator1 number2” (i.e. 2+4).

b). Student Record management system:

Using C language, you can also create a student management system. To handle students’ records (like Student’s roll number, Name, Subject, etc.) it employs files as a database to conduct file handling activities such as add, search, change, and remove entries.

It appears a simple project, but can be handy for schools or colleges that have to store records of thousands of students.

c). Calendar:

If you have ever lost track of which day of the week is today or the number of days in that particular month, you should build a calendar yourself. The Calendar is written in the C programming language, and this Calendar assists you in determining the date and day you require. We can implement it using simple if-else logic and switch-case statements. The display() function is used to display the calendar and it can be modified accordingly. It also has some additional functions. The GitHub link of the calendar has been provided below.

d). Phone Book:

This Phone book Project generates an external file to permanently store the user's data (Name and phone number). The phone book is a very simple C project that will help you understand the core concepts of capacity, record keeping, and data structure. This program will show you how to add, list, edit or alter, look at, and delete data from a record.

e). Unit Converter Project:

Forgot how to convert degree Fahrenheit to Celsius? Don't worry. We have a solution for you. This unit converter converts basic units such as temperature, currency, and mass.

3.5.2 Intermediate C Projects with Source Code

a). Mini Voting System:

An online voting system is a software platform that enables organizations to conduct votes and elections securely. A high-quality online voting system strikes a balance between ballot security, convenience, and the overall needs of a voting event. By collecting the input of your group in a systematic and verifiable manner, online voting tools and online election voting systems assist you in making crucial decisions. These decisions are frequently taken on a yearly basis – either during an event (such as your organization's AGM) or at a specific time of the year. Alternatively, you may conduct regular polls among your colleagues (e.g., anonymous employee feedback surveys).

b). Voting System:

With this voting system, users can enter their preferences and the total votes and leading candidate can be calculated. It's a straightforward C project that's simple to grasp. Small-scale election efforts can benefit from this.

c). Tic-tac-toe Game:

Tic-tac-toe, also known as noughts and crosses or Xs and Os, is a two-person paper and pencil game in which each player alternates marking squares in a three-by-three grid with an X or an O. The winner is the player who successfully places three of their markers in a horizontal, vertical, or diagonal row. You can implement this fun game using 2D arrays in the C programming language. It is important to use arrays while creating a Tic Tac Toe game in the C

programming language. The Xs and Os are stored in separate arrays and passed across various functions in the code to maintain track of the game's progress. You can play the game against the computer by entering the code here and selecting either X or O. The source code for the project is given below.

d). Matrix Calculator:

Mathematical operations are an everyday part of our lives. Every day, we will connect with many forms of calculations in our environment. Matrices are mathematical structures in which integers are arranged in columns and rows. In actual life, matrices are used in many applications. The most common application is in the software sector, where pathfinder algorithms, image processing algorithms, and other algorithms are developed. Some fundamental matrix operations are performed in this project, with the user selecting the operation to be performed on the matrix. The matrices and their sizes are then entered. It's worth noting that the project only considers square matrices.

e). Library Management System:

Library management is a project that manages and preserves electronic book data based on the demands of students. Both students and library administrators can use the system to keep track of all the books available in the library. It allows both the administrator and the student to look for the desired book. The C files used to implement the system are: main.c, searchbook.c, issuebook.c, viewbook.c, and more.

f). Electricity Bill Calculator

The Electricity Cost Calculator project is an application-based

micro project that predicts the following month's electricity bill based on the appliances or loads used. Visual studio code was used to write the code for this project. This project employs a multi-file and multi-platform strategy (Linux and Windows). People who do not have a technical understanding of calculating power bills can use this program to forecast their electricity bills for the coming months; however, an electricity bill calculator must have the following features:

- All loads' power rating
- Unit consumed per day
- Units consumed per month, and
- Total load calculation

g). Movie Ticket Booking System

The project's goal is to inform a consumers about the MOVIE TICKET BOOKING SYSTEM so that they can order tickets. The project was created with the goal of making the process as simple and quick as possible. The user can book tickets, cancel tickets, and view all booking records using the system. Our project's major purpose is to supply various forms of client facilities as well as excellent customer service. It should meet nearly all the conditions for reserving a ticket.

3.5.3 Advanced C Projects with Source Code

a). Snakes and Ladders Game:

Snakes and ladders, also known as Moksha Patam, is an ancient Indian board game for two or more players that is still considered a worldwide classic today. It's played on a gridded game board with

numbered squares. On the board, there are several “ladders” and “snakes,” each linking two distinct board squares. The dice value can either be provided by the user or it can be generated randomly. If after moving, the pointer points to the block where the ladder is, the pointer is directed to the top of the ladder. If unfortunately, the pointer points to the mouth of a snake after moving, the pointer is redirected to the tail of the snake. The objectives and rules of the game can be summarized as-

Objective – Given a snake and ladder game, write a function that returns the minimum number of jumps to take the top or destination position.

You can assume the dice you throw results in always favor of you, which means you can control the dice.

b). Lexical Analyzer:

The Lexical Analyzer program translates a stream of individual letters, which are generally grouped as lines, into a stream of lexical tokens. Tokenization, for example, of source code words and punctuation symbols. The project’s main goal/purpose is to take a C file and generate a sequence of tokens that can be utilized in the next stage of compilation. This should also account for any error handling requirements that may arise during tokenization.

c). Bus Reservation System:

This system is built on the concept of booking bus tickets in advance. The user can check the bus schedule, book tickets, cancel reservations, and check the bus status board using this system. When purchasing tickets, the user must first enter the bus number,

after which the system will display the entire number of bus seats along with the passengers' names, and the user must then enter the number of tickets, seat number, and person's name.

We will be using arrays, if-else logic, loop statements, and various functions like login(), cancel(), etc. to implement the project.

d). Dino Game:

This little project is a modest recreation of the Offline Google Chrome game Dinosaur Jump. The game can be played at any moment by the user. The entire project is written in the C programming language. The X key is used to exit the game, and the Space bar is used to leap. play and score as many points as you can; this is a fun, simple game designed specifically for novices, and it's simple to use and understand.

e). Pac-Man Game:

Pacman, like other classic games, is simple to play. In this game, you must consume as many small dots as possible to earn as many points as possible. The entire game was created using the C programming language. Graphics were employed in the creation of this game. To create the game, you have to first define the grid function to manage the grid structure. To control the movement, you can define functions such as move_right(), move_left(), move_up() and move_down(). C files to add ghosts and their functionalities, positions check, etc. can be added to make the game more fun. The customers will find this C Programming game to be simple to comprehend and manage.

3.6 CONCLUSION

C is clearly not an obsolete programming language, as evidenced

by the fact that many of the world's greatest businesses use it for their profession or company. On the contrary, it remains the most popular programming language for developers and back-end developers throughout the world. This event taught us about the practical use of C programming. Our research revealed that C is used in all hardware and software technologies, making it advantageous for both aspiring software developers and seasoned software specialists with a good command of C and the ability to construct sophisticated interfaces.

In this unit, we've gathered several C language projects and ideas for you. As the world's largest software development community, GitHub has amassed a massive collection of projects from programmers who constantly study and evaluate each other's work. Furthermore, because the platform supports a wide range of programming languages, there is a wealth of C project ideas on GitHub for anybody to draw inspiration from. It is your responsibility as a developer to think outside the box, devise imaginative solutions utilizing available resources, and contribute to the future of software. The projects/software are organized under various topics for the sake of clarity. So, if you're new to project development, begin by comprehending and evaluating a little project before moving on to a larger project scope and application.

BLOCK II: OPERATOR AND EXPRESSIONS OF ‘C’, CONTROL FLOW MECHANISMS

UNIT 4 FUNDAMENTAL ELEMENTS OF ‘C’

Structure

4.0 Introduction

4.1 Objectives

4.2 Data Types in C

4.2.1 Primitive Data Types

4.2.2 User-Defined Data Types

4.2.3 Derived Data Types

4.2.4 Lvalues and Rvalues in C

4.3 Variables

4.3.1 Variable definition in C

4.3.2 Declaring Variables

4.3.3 Initializing Variables

4.4 Conclusion

4.5 Unit based Questions /Answers

4.0 INTRODUCTION

In this unit introduces Computer programs usually work with different types of data and need a way to store the values being used. These values can be numbers. C language has two ways of storing number values—Data types and Variables—with many options for each. Data types and variables are the fundamental elements of each program. Simply speaking, a program is nothing else than defining them and manipulating them. A variable is a data storage location that has a value that can change during program execution. In contrast, a constant has a fixed value that

can’t change.

This unit is concerned with the basic elements used to construct simple C program statements. These elements include the C character set, identifiers and keywords, data types, constants, variables and arrays, declaration and naming conventions of variables.

This unit will explain to you the fundamental elements of language C.

4.1 OBJECTIVES

After completing this unit, you will be able to:

- define identifiers and data types in C;
- know name the identifiers as per the conventions;
- describe memory requirements for different types of variables; and
- define constants, symbolic constants and their use in programs.

4.2 DATA TYPES IN ‘C’

Each variable in C has an associated data type. It specifies the type of data that the variable can store like integer, character, floating, double, etc. Each data type requires different amounts of memory and has some specific operations which can be performed over it. The data type is a collection of data with values having fixed values, meaning as well as its characteristics.

The data types in C can be classified as follows:

Types	Description
Primitive Data Types	Primitive data types are the most basic data types that are used for representing simple values such as integers, float, characters etc.
User Defined	The user-defined data types are defined by the

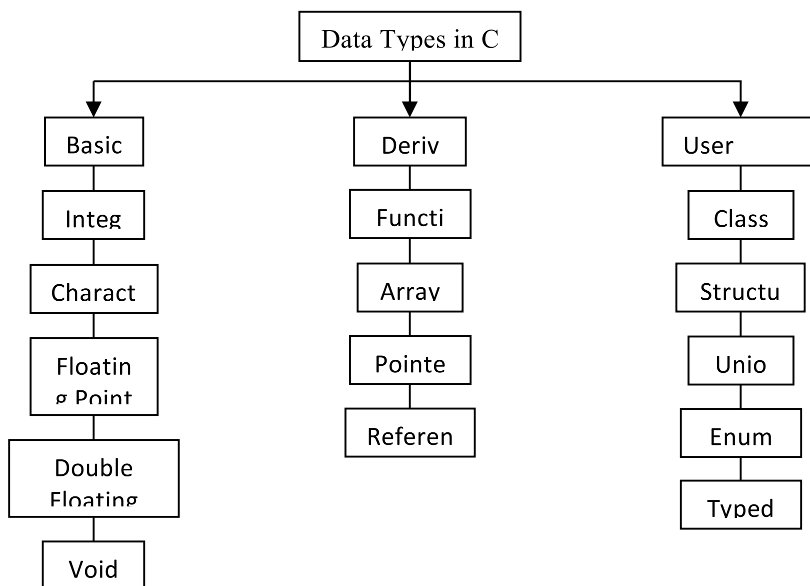
Data Types	user himself.
Derived Types	The data types that are derived from the primitive or built-in data types are referred to as derived data types.

Types	Data Types
Basic Data Type	Int, char, float, double
Derived Data Type	Array, Pointer, Structure, Union
Enumeration Data Type	Enum
Void Data Type	Void

4.2.1 Primitive Data Types (Basic Data Types)

The basic data types are integer-based and floating-point based. C language supports both signed and unsigned literals.

The memory size of the basic data types may change according to 32 or 64-bit operating system.



Different data types also have different ranges up to which they can store numbers. These ranges may vary from compiler to compiler. Below is a list of ranges along with the memory requirement and format specifiers on the **32-bit GCC compiler**.

i) Data Types and Storage

To store data inside the computer we need to first identify the type of data elements we need in our program. There are several different types of data, which may be represented differently within the computer memory. The data type specifies two things:

1. Permissible range of values that it can store.
2. Memory requirement to store a data type.

C Language provides four basic data types viz. int, char, float and double. Using these, we can store data in simple ways as single elements or we can group them together and use different ways (to be discussed later) to store them as per requirement.

ii). Integer Types

Integers are entire numbers without any fractional or decimal parts, and the int data type is used to represent them.

It is frequently applied to variables that include values, such as counts, indices, or other numerical numbers. The int data type may represent both positive and negative numbers because it is signed by default.

An int takes up 4 bytes of memory on most devices, allowing it to store values between around -2 billion and +2 billion.

Data Types	Memory Size	Range
Char	1 byte	-128 to 127
Signed char	1 byte	-128 to 127
Unsigned char	1 byte	0 to 255
Short	2 byte	-32,768 to 32,767
Signed short	2 byte	-32,768 to 32,767
Unsigned short	2 byte	0 to 65,535

Int	2 byte	-32,768 to 32,767
Signed int	2 byte	-32,768 to 32,767
Unsigned int	2 byte	0 to 65,535
Short int	2 byte	-32,768 to 32,767
Signed short int	2 byte	-32,768 to 32,767
Unsigned short int	2 byte	0 to 65,535
Long int	4 byte	-2,147,483,648 to 2,147,483,647
Signed long int	4 byte	-2,147,483,648 to 2,147,483,647
Unsigned long int	4 byte	0 to 4,294,967,295
Float	4 byte	
double	8 byte	
Long double	10 byte	

Short, long, signed, unsigned are called the data type qualifiers and can be used with any data type. A short int requires less space than int and long int may require more space than int. If int and short int takes 2 bytes, then long int takes 4 bytes.

Unsigned bits use all bits for magnitude; therefore, this type of number can be larger. For example signed int ranges from –32768 to +32767 and unsigned int ranges from 0 to 65,535. Similarly, char data type of data is used to store a character. It requires 1 byte. Signed char values range from –128 to 127 and unsigned char value range from 0 to 255.

To get the exact size of a type or a variable on a particular platform, you can use the sizeof operator. The expressions sizeof(type) yields the storage size of the object or type in bytes. Given below is an example to get the size of int type on any machine:

```
#include <stdio.h>
#include <limits.h>
Int main()
{
Printf("Storage size for int: %d \n", sizeof(int));
Return 0;
}
```

When you compile and execute the above program, it produces the following result on Linux:

```
Storage size for int: 4
```

4.2.2 User Defined Data Types

i). Character

Individual characters are represented by the char data type. Typically used to hold ASCII or UTF-8 encoding scheme characters, such as letters, numbers, symbols, or commas. There are 256 characters that can be represented by a single char, which takes up one byte of memory. Characters such as 'A', 'b', '5', or '$' are enclosed in single quotes.

Data Type	Storage Space	Format	Range of Values
Char	1 byte	%c	ASCII character set (-128 to 127)
Unsigned char	1 byte	%c	ASCII character set (0 to 255)

ii). Floating-Point Types

To represent integers, use the floating data type. Floating numbers can be used to represent fractional units or numbers with decimal places.

The float type is usually used for variables that require very good precision but may not be very precise. It can store values with an accuracy of about 6 decimal places and a range of about 3.4 x 1038 in 4 bytes of memory.

Type	Storage Size	Value Range	Precision
Float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
Double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
Long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

The header file float.h defines macros that allow you to use these values and other details about the binary representation of real

numbers in your programs. The following example prints the storage space taken by a float type and its range values:

```
#include <stdio.h>
#include <float.h>
int main()
{
printf("Storage size for float : %d \n", size of (float));
printf("Minimum float positive value: %E\n", FLT_MIN );
printf("Maximum float positive value: %E\n", FLT_MAX );
printf("Precision value: %d\n", FLT_DIG );
return 0;
}
```

iii). Double:

Use two data types to represent two floating integers. When additional precision is needed, such as in scientific calculations or financial applications, it provides greater accuracy compared to float.

Double type, which uses 8 bytes of memory and has an accuracy of about 15 decimal places, yields larger values. C treats floating point numbers as doubles by default if no explicit type is supplied.

```
Int age = 25;
Char grade = 'A';
Flaot temperature = 98.6;
Double pi = 3.14159265359;
```

In the example above, we declare four variables: an int variable for the person's age, a char variable for the student's grade, a float

variable for the temperature reading, and two variables for the number pi.

4.2.3 Derived Data Type

Beyond the fundamental data types, C also supports derived data types, including arrays, pointers, structures, and unions. These data types give programmers the ability to handle heterogeneous data, directly modify memory, and build complicated data structures.

Array

An array, a derived data type, lets you store a sequence of fixed-size elements of the same type. It provides a mechanism for joining multiple targets of the same data under the same name.

The index is used to access the elements of the array, with a 0 index for the first entry. The size of the array is fixed at declaration time and cannot be changed during program execution. The array components are placed in adjacent memory regions.

Here is an example of declaring and utilizing an array:

```
#include <stdio.h>
int main() {
int numbers[5]; // Declares an integer array with a size of 5 elemen
ts

// Assign values to the array elements
numbers[0] = 10;
numbers[1] = 20;
    numbers[2] = 30;
```

```
numbers[3] = 40;
numbers[4] = 50;
    // Display the values stored in the array
printf("Values in the array: ");
for (int i = 0; i < 5; i++) {
  printf("%d ", numbers[i]);
}
printf("\n");
return 0;
}
```

Output:

Values in the array: 10 20 30 40 50

The void Type

S.No.	Type and Description
1.	**Functions returning void** In C programming, some functions are designed not to return any value. Such functions are declared with a return type of void. This indicates that the function performs a task but does not send any result back to the caller. **Example:** void exit (int status);
2.	**Functions with void Parameters** Some functions in C do not require any input arguments. When no parameters are needed, void can be specified in the parameter list to explicitly indicate this. **Example:** int rand(void);
3.	**Void Pointers** A void * pointer is a special type of pointer in C that can hold the address of any data type, though it does not have a specific type itself. It is commonly used in functions dealing with raw memory, such as memory allocation routines. **Example:** void *malloc(size_t size);

4.3 VARIABLES

Understanding Variables in C

In C programming, a **variable** serves as a label for a memory location that can hold data to be used and modified by a program. Each variable has a **data type**, which defines:

- The **amount of memory** allocated,
- The **range of values** it can store, and
- The **operations** that can be performed on it.

Naming Rules for Variables

- Variable names can consist of **letters (A-Z, a-z)**, **digits (0-9)**, and the **underscore (_) character**.
- A variable name **must start** with either a letter or an underscore.
- C is a **case-sensitive** language, so Value and value would be treated as two different variables.

Types of Variables

Based on the fundamental data types introduced earlier, the basic types of variables include:

- int – for integers,
- float – for floating-point numbers,
- char – for characters,
- double – for double-precision floating-point numbers.

These types dictate the kind of data the variable can store and how much memory it occupies.

Type	Description
Char	Typically a single octet (one byte). This is an integer type.

Int	The most natural size of integer for the machine.
Float	A single-precision floating point value.
Double	A double-precision floating point value.
void	Represents the absence of type.

C programming language also allows to define various other types of variables, which we will cover in subsequent chapters like Enumeration, Pointer, Array, Structure, Union, etc. For this chapter, let us study only basic variable types.

Rules for declaring variable name

- Variable name may be a combination of alphabet, digits, or underscores and its length should not exceed eight characters.
- First character must be an alphabet.
- No commas or blank spaces are allowed in variable name.
- Among the special symbols, only underscore can be used in variable name.
- Example: emp_age and item_ 4•

4.4 CONCLUSION

C programming language because they define the kinds of information that variables can hold. They provide the data's size and format, enabling the compiler to allot memory and carry out the necessary actions. Data types supported by C include void, enumeration, derived, and basic types. In addition to floating-point types like float and double, basic data types in C also include integer-based kinds like int, char, and short. These forms can be signed or unsigned, and they fluctuate in size and range. To create dependable and efficient code, it is crucial to comprehend the memory size and scope of these types.

A few examples of derived data types are unions, pointers, structures, and arrays. Multiple elements of the same kind can be stored together in contiguous memory due to arrays. Pointers keep track of memory addresses, allowing for fast data structure operations and dynamic memory allocation. While unions allow numerous variables to share the same memory space, structures group relevant variables together.

Code becomes more legible and maintainable when named constants are defined using enumeration data types. Enumerations give named constants integer values to enable the meaningful representation of related data. The void data type indicates the lack of a particular type. It is used as a return type for both functions and function parameters that don't take any arguments and don't return a value. The void* pointer also functions as a general pointer that can store addresses of various types.

C programming requires a solid understanding of data types. Programmers can ensure adequate memory allocation, avoid data overflow or truncation, and enhance the readability and maintainability of their code by selecting the right data type. C programmers may create effective, dependable, and well-structured code that satisfies the requirements of their applications by having a firm understanding of data types.

UNIT 5 OPERATOR AND EXPRESSIONS OF ‘C’

Structure

5.0 Introduction

5.1 Objectives

5.2 Operators in ‘C’

5.2.1 Assignment Operators

5.2.2 Arithmetic Operators

5.2.3 Relational Operators

5.2.4 Logical Operators

5.2.5 Increment and Decrement Operators

5.2.6 Conditional Operators

5.2.7 Special Operators

5.2.8 Size of Operator

5.2.9 ‘C’ Short Hand

5.2.10 Priority of Operators

5.3 Expressions in ‘C’

5.3.1 Types Conversion in Expressions

5.3.1.1 Automatic Type Conversion

5.3.1.2 Casting a Value

5.4 Conclusion

5.5 Unit based Questions /Answers

5.0 INTRODUCTION

This unit explores the fundamental building blocks of C programming: variables, constants, and datatypes, along with their declaration. We'll then move on to using these declared elements within expressions. An expression combines operators and operands to achieve one or more of the following:

- Calculate a value.
- Identify an object or function.
- Produce side effects.

Essentially, an operator performs an action (or evaluation) on one or more operands, where an operand is a sub-expression that the operator acts upon.

This unit will detail the various types of operators available in C, covering their syntax and usage within the language. Unlike a basic calculator, computers can also solve logical expressions. Therefore, in addition to arithmetic operators, C also includes logical operators, which will be discussed in this unit.

This unit will explain to you the expressions and operators of C language.

5.1 OBJECTIVES

After completing this unit, you will be able to:

- write and evaluate arithmetic expressions;
- express and evaluate relational expressions;
- write and evaluate logical expressions;
- write and solve compute complex expressions (containing arithmetic, relational and logical operators), and
- check simple conditions using conditional operators.

5.2 OPERATORS IN 'C'

In programming, an operator is a special symbol used to perform

operations on variables, operands, or constants. Some operators require two operands to execute their function, while others can operate with just a single operand. Operators come in various forms, including arithmetic, assignment, increment, decrement, logical, conditional, comma, sizeof, and bitwise operators, among others. Essentially, an operator acts as an instruction to the computer, prompting it to carry out specific mathematical or logical manipulations on data stored in variables, which are then referred to as operands. The C programming language, in particular, is rich in built-in operators, providing a comprehensive set of these symbols for diverse operations. C language is rich in built-in operators and provides the following types of operators:

- Assignment Operators
- Arithmetic Operators
- Relational Operators
- Logical Operators
- Increment and decrement Operators
- Conditional Operators
- Special Operators

5.2.1 Assignment Operators

In this unit, we've come to understand that variables are essentially named locations within a computer's memory, designated to store various pieces of data. The fundamental question then arises: how do we actually place values into these variables? The C programming language provides a crucial mechanism for this purpose: the assignment operator. This operator's core function is quite straightforward yet powerful: it takes a value or the result of an expression found on the right-hand side of an operation and efficiently stores that information into the designated variable

located on the left-hand side.

The standard syntax for an assignment expression in C follows a clear pattern: `Variable = constant / variable / expression;`. A critical consideration here is data type compatibility. Generally, the data type of the variable on the left-hand side should align with the data type of the constant, variable, or expression on the right-hand side. This ensures that the data being assigned can be properly understood and stored. However, it's worth noting that C does offer some flexibility; in certain scenarios, automatic type conversions are possible, allowing for assignments between slightly different data types without explicit intervention. For example, you might assign an integer value to a floating-point variable, and the system would handle the conversion.

At their heart, assignment operators are indispensable tools used to effectively transfer the outcome of an operation or a direct value into a variable for later use. While there are various assignment operators in C (such as `+=`, `-=`, etc.), the most commonly and frequently used is the simple equals sign (`=`). This foundational operator is integral to nearly every C program, enabling the dynamic manipulation and storage of data as a program executes.

An expression with assignment operator is of the following form:

Identifier = expression;

Example:

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

```
        int i;
        i = 5;
        printf ("%d", i);
        i = i + 10;
        printf ("\n%d", i);
}
```

Output will be: 5

10

Expressions like i = i+10; , i = i-5; , i = i*2; , i = i/6, and i = i% 10 can be rewritten using shorthand assignment operators.

The shorthand assignment operators are of the following type:

```
V  op = expression;
```

This is equivalent to

```
V  = V op expression;
```

Example:

I = I + 5:   is equivalent to
i + =5;

I = I * (y+5);   is equivalent to   i*   =
(y+5);

5.2.2 Arithmetic Operators

The basic arithmetic operators in C are the same as in most other

computer languages, and correspond to our usual mathematical/algebraic symbolism. The following arithmetic operators are present in C:

Operator	Purpose
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder after integer division

The division operator (/) in C requires that its second operand is not zero, even when dealing with non-integer values. Relatedly, the modulus operator (%) calculates the remainder after the division of two operands. Like the division operator, its second operand must also be non-zero. Attempting to divide by zero is an undefined operation in computer systems and will typically lead to a run-time error. In C, arithmetic expressions are usually written in a straightforward, linear fashion; for example, "a divided by b" is simply expressed as a/b.

The operands used in C's arithmetic expressions can be of various data types, including integer, float, and double. To effectively develop C programs, it's essential to grasp the rules governing the implicit conversion of floating-point and integer values, which are outlined below:

- When an arithmetic operation occurs between two integers, the result will always be an integer.

- If an operator acts between two float values, the outcome will be a float result.
- When an operation involves an integer and a float, the result will be a float.
- *If the data type is double instead of float, then we get a result of double data type.*

Example:

if a = 25, b = 4

then   a + b = 29

        a - b = 21

        a * b = 100

        a/b = 6   (decimal parts truncated)

        a%b = 1

Example:

```
#include<stdio.h>
main()
{
        int a = 21;
        int b = 10;
        int c ;
        c = a + b;
        printf("Value of c is %d\n", c );                    /* Line
1 */
        c = a - b;
        printf("Value of c is %d\n", c );                    /* Line
2 */
        c = a * b;
        printf("Value of c is %d\n", c );                    /* Line
3 */
```

```
        c = a / b;
        printf("Value of c is %d\n", c );                    /* Line
4 */
        c = a % b;
        printf("Value of c is %d\n", c );                    /* Line
5 */
    c = a++;
        printf("Value of c is %d\n", c );                    /* Line
6 */
        c = a--;
        printf("Value of c is %d\n", c );                    /* Line
7 */
}
```

Output will be:

```
Value of c is 31          /* Line 1 */
Value of c is 11          /* Line 2 */
Value of c is 210         /* Line 3 */
Value of c is 2      /* Line 4 */
Value of c is1       /* Line 5 */
Value of c is 21          /* Line 6 */
Value of c is 22          /* Line 7 */
```

5.2.3 Relational Operators

In C programming, executable statements are designed to perform actions like calculations or data input/output, or to facilitate decision-making within a program. We can compare variables using relational operators. The various C relational operators and their meanings are summarized below. It's crucial to note that the equality operator uses two equal signs (`==`), not just one, which is

distinct from the assignment operator (`=`). This section also introduces a basic form of C's `if` control structure, enabling a program to make decisions based on the outcome of a given condition. If the condition evaluates to true, the statement within the `if` body is executed. Conversely, if the condition is false, the statement is skipped. Regardless of whether the body statement is executed, program execution always continues with the statement immediately following the `if` structure once it completes.

A relational operator is employed to compare two operands, determining if they are equal, unequal, or if one is greater or less than the other. These operands can be variables, constants, or expressions, and the comparison yields a numerical result. There are six distinct relational operators available in C.

Operator	Description	Example
= =	If the values of two operands are equal or not. If yes, then the condition becomes true.	(A == B) is not true.
! =	If the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	(A != B) is true.
>	If the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	(A > B) is not true.
<	If the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	(A < B) is true.
> =	If the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	(A >= B) is not true.

< =	If the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	(A <= B) is true.

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

ae-1 relational operator ae-2

Relational operators usually appear in statements which are inquiring about the truth of some particular relationship between variables. Normally, the relational operators in C are the operators in the expressions that appear between the parentheses.

For example,

(i) if (this num < minimum sofar) minimum sofar = this num

(ii) if (job == Teacher) salary == minimum wage

(iii) if (number of legs != 8) this bug = insect

(iv) if (degree of polynomial < 2) polynomial = linear

Let a simple C program containing the If statement (will be introduced in detail in the next unit). It displays the relationship between two numbers read from the keyboard.

Example:

```
/*Program to find relationship between two numbers*/

#include<stdio.h>
main ( )
{
int a, b;
printf ( “Please enter two integers: ”);
scanf (“%d%d”, &a, &b);
```

```
if (a <= b) printf (“ %d <= %d\n”,a,b);
else
printf (“%d > %d\n”,a,b);
}
```

OUTPUT:

Please enter two integers: 12 17

12 <= 17

Example:

```
/*Program to understand all the operators available in C*/

#include<stdio.h>
main ( )
{
            int a = 21;
 int b = 10;
 int c ;
 if( a == b )
{ printf("a is equal to b\n" );                    /* Line 1 */
}
 else
{
 printf("a is not equal to b\n" );                    /*  Line
1 */
}
 if ( a < b )
{
 printf("a is less than b\n" );                    /* Line 2 */
}
 else
{
 printf("a is not less than b\n" );                    /*  Line
```

```
2 */
}
 if ( a > b )
{
 printf("a is greater than b\n" );                              /*  Line
3 */
}
 else
{
            printf("a is not greater than b\n" );
           /* Line 3 */
           }
            /* Lets change value of a and b */
            a = 5;
            b = 20;
            if ( a <= b )
           {
            printf("a is either less than or equal to b\n" );
           /* Line 4 */
           }
            if ( b >= a )
           {
            printf("b is either greater than or equal to b\n" );
           /* Line 5 */
            }
            }
```

OUTPUT:

```
a is not equal to b                        /* Line 1 */
a is not less than b                       /* Line 2 */
a is greater than b                        /* Line 3 */
a is either less than or equal to b              /*  Line
```

4 */

b is either greater than or equal to b /* Line

5 */

5.2.4 Logical Operators

In C, much like other programming languages, logical operators are key for evaluating expressions that can either be true or false. When an expression involves these logical operations, it's assessed to determine one of those two Boolean outcomes. While we've only covered simple conditions so far, there's a good chance you'll need to test multiple conditions when making decisions. If that's the case, you could string together a bunch of separate if statements (we'll dive deeper into those later). Luckily, C gives us logical operators to make things easier, allowing us to combine those simple conditions into more complex ones.

Following table shows all the logical operators supported by C language. Assume variable A holds 1 and variable B holds 0, then:

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

Thus logical operators (AND and OR) combine two conditions and

logical NOT is used to negate the condition i.e. if the condition is true, NOT negates it to false and vice versa. Let us consider the following examples:

(i) Suppose the grade of the student is ‘B’ only if his marks lie within the range 65 to 75,if the condition would be:

```
if ((marks >=65) && (marks <= 75))
printf (“Grade is B\n”);
```

(ii) Suppose we want to check that a student is eligible for admission if his PCM is greater than 85% or his aggregate is greater than 90%, then,

```
if ((PCM >=85) ||(aggregate >=90))
printf (“Eligible for admission\n”);
```

Logical negation (!) enables the programmer to reverse the meaning of the condition. Unlike the && and || operators, which combines two conditions (and are therefore Binary operators), the logical negation operator is a unary operator and has one single condition as an operand. Let us consider an example:

```
if !(grade==’A’)
printf (“the next grade is %c\n”, grade);
```

The parentheses around the condition grade==A are needed because the logical operator has higher precedence than equality operator. In a condition if all the operators are present then the order of evaluation and associativity is provided in the table. The truth table of the logical AND (&&), OR (||) and NOT (!) are given below.

These table show the possible combinations of zero (false) and nonzero (true) values of X (expression1) and Y (expression2) and

only one expression in case of NOT operator.

The following table is the truth table for && operator.

Truth table for && operator

X	Y	X&&Y
0	0	0
1	0	0
0	1	0
1	1	1

The following table is the truth table for | | operator.

Truth table for && operator

X	Y	X| |Y
0	0	0
1	0	1
0	1	1
1	1	1

The following table is the truth table for | | operator.

Truth table for ! operator

X	!X
0	1
1	0

The following table is the operator precedence and associativity.

Truth table for (Logical operator precedence and associativity)

Operator	Associativity
!	Right to left
&&	Left to right
| |	Left to right

Example:

/*Program to understand all the logical operators available in C*/

```
#include<stdio.h>
main ( )
{
int a = 5;
int b = 20;
int c ;
if ( a && b )
{
printf("Line 1 - Condition is true\n" );
}
if ( a || b )
{
printf("Line 2 - Condition is true\n" );
}
/* lets change the value of a and b */
a = 0;
b = 10;
if ( a && b )
{
printf("Line 3 - Condition is true\n" );
}
else
{
printf("Line 3 - Condition is not true\n" );
}
if ( !(a && b) )
{
printf("Line 4 - Condition is true\n" );
```

```
}
}
```

OUTPUT:

```
Condition is true
/* Line 1 */
Condition is true
/* Line 2 */
Condition is not true
/* Line 3 */
Condition is true
/* Line 4 */
```

5.2.5 Increment and decrement Operators

C boasts two incredibly handy operators, `++` and `--`, known as the increment and decrement operators, respectively. You won't typically find these quite as prevalent in many other programming languages, making them a distinctive feature of C. Because they operate on just one piece of data, they're classified as unary operators. It's important to remember that these operators must be applied to variables; you can't use them directly on constants.

At their core, the `++` (increment) operator simply adds one to the value of its operand, while the `--` (decrement) operator does the opposite, subtracting one from the operand's value. What's more, these operators can be employed in a couple of different ways, which offers flexibility in how they're used within your code.

1) Prefix: When the operator used before the operand, it is termed as prefix.

e.g. ++A , --B

in this case the value of operand follow First Change Then Use (F.C.T.U) concept.

2) Postfix: When the operator used before the operand, it is termed as prefix.

e.g. A++ , B—

in this case the value of operand follow First Use Then Change (F.U.T.C) concept.

Example:

Postfix int N=10, R;

R = N++; // post increment

printf("R=%d \n N=%d ", R , N);

it will produce **output:**

R=11 (Because value will increment first, then value will assign)

N=11

Example:

```
#include<stdio.h>
Void main ( )
{
    int R, N=10;
    clrscr();
    R = ++N + --N + --N + N++ + --N ;
    printf( " R= %d \n N=%d" , R,N);
    getch();
    }
    The output will be:
    R=40
```

N=9

Explanation:

In this example there are for than one increment or decrement expressions are used, so that it follows the execution order **(postfix → operation → prefix).**

In this statement (++N + --N + --N + N++ + --N ;) all prefix expression execute first ,then it perform operations like addition/subtraction or assignment etc. and after that it perform all postfix operations. In the operation the value of all operand will give last modified value by prefix operations.

5.2.6 Conditional or Ternary Operators

In C programming, the conditional operator (?:) offers a compact alternative to the traditional if/else statement. It is the only ternary operator in C, meaning it operates on three operands. These operands form a conditional expression where the first part is a condition that is evaluated. If this condition is true, the second operand becomes the result of the expression; if the condition is false, the third operand is returned instead. This operator is especially useful for writing short decision-making expressions in a more concise and readable format.

The syntax is as follows:

(condition)? (expression1): (expression2);

If condition is true, expression1 is evaluated else expression2 is evaluated. Expression1/Expression2 can also be further conditional expression i.e. the case of nested if statement.

Let us see the following examples:

(i) x= (y<20) ? 9:10;

This means, if(y<20), then x=9 else x=10;

(ii) printf("%s\n", grade>=50? "Passed": "failed");

The above statement will print "passed" grade>=50 else it will print "failed".

(iii) (a>b) ? printf ("a is greater than b \n"): printf ("b is greater than a \n"); If a is greater than b, then first printf statement is executed else second printf statement is executed.

Example 1:

a = 10;

b = 15;

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

In this expression value of b will be assigned to x.

Example 2:

a = 10;

b = 15;

x = (a > b)?

printf ("First value is Greater");

printf ("Second value is greater");

In this expression the result will be-

5.2.7 Special Operators

The C language provides several special operators, including the comma operator, the `sizeof` operator, pointer operators (`&` and `*`), and member selection operators (`.` and `->`). The pointer operators will be explained in detail when the concept of pointers is introduced, while the member selection operators will be

covered during the discussion of structures and unions. For now, the focus will be on understanding the comma operator and the `sizeof` operator.

Comma Operator

This operator is used to link the related expressions together.

Example:

```
Int val, x, y;
value = (x= 10, y = 5, x+y);
```

it first assigns 10 to x then 5 to y finally sum x + y to value.

Size of Operator

The sizeof operator in C is evaluated at compile time and is used to determine the amount of memory (in bytes) that a given operand occupies. This operand can be a variable, a constant, or even a data type qualifier. It helps in understanding the memory requirements of different data types and expressions during program development.

Example:

```
int n;
n = size of (int);
printf("n=%d,\n",n);
n = size of (double);
printf("n=%d",n);
```

Output:

```
n = 2
n = 8
```

5.2.8 Size of Operators

C provides a compile-time unary operator called **size of** that can be used to compute the size of any object. The expressions such as:

Size of object and size of (type name)

result in an unsigned integer value equal to the size of the specified object or type in bytes. The result of the sizeof operator is an integer that represents the number of bytes needed to store an object of the operand's type. This object could be a variable, an array, or a structure. Arrays and structures are examples of data structures in C, which will be introduced in later units. The operand can be a basic data type such as int or double, or a derived type like a pointer or a structure.

Example:

Size of(char) = 1bytes

Size of(int) = 2 bytes

5.2.9 C Short Hand

C has a special shorthand that simplifies coding of certain type of assignment statements.

Example:

a = a+2;

can be written as:

a += 2;

The operator +=tells the compiler that a is assigned the value of a + 2; This shorthand works for all binary operators in C. The general form is:

variable operator = variable / constant / expression

These operators are listed below:

Operators	Examples	Meaning
+=	a+=2	a=a+2
-=	a-=2	a=a-2
=	a*=2	a = a*2
/=	a/=2	a=a/2
%=	a%=2	a=a%2

Operators	Examples	Meaning
&&=	a&&=c	a=a&&c
||=	a||=c	a=a||c

5.2.10 Priority Operators

Since all the operators we have studied in this unit can be used together in an expression, C uses a certain hierarchy to solve such kind of mixed expressions. The hierarchy and associatively of the operators discussed so far is summarized in Table 6. The operators written in the same line have the same priority. The higher precedence operators are written first.

Table: Precedence of the operators

Operators	Associativity
( )	Left to right
! ++ -- (type) size of	Right to left
/ %	Left to right
+ -	Left to right

<table>
<tr><td>< <= > >=</td><td>Left to right</td></tr>
<tr><td>== !=</td><td>Left to right</td></tr>
<tr><td>&&</td><td>Left to right</td></tr>
<tr><td>||</td><td>Left to right</td></tr>
<tr><td>?:</td><td>Right to left</td></tr>
<tr><td>= += -= *= /= %=</td><td>Left to right</td></tr>
<tr><td>&&= ||=</td><td>Left to right</td></tr>
<tr><td>,</td><td></td></tr>
</table>

5.3 EXPRESSIONS IN ‘C’

An expression is a combination of variables, constants, and operators arranged according to syntax of the language. Some examples of expressions are:

e.g.

```
c = (m + n) * (a - b);
temp = (a + b + 'c) / (d - c);
```

Expression is evaluated by using assignment statement.

Such a statement is of the form

Variable = expression;

The expression is evaluated first, then the value is assigned to the variable left hand side. But all the relevant variables must be assigned the values before evaluation of the expression.

5.3.1 Type conversion in Expressions

Type conversion in expressions refers to the process where values

of one data type are transformed into another to ensure compatibility within operations. This conversion can be implicit, where the programming language automatically handles the conversion, or explicit, where the programmer must manually convert the types using specific functions or methods. Implicit conversions usually occur in expressions involving mixed data types, ensuring that operations such as arithmetic or comparisons are performed correctly. Explicit conversions, often called type casting, are necessary when the automatic conversions do not yield the desired outcome or when precision and control over the data transformation are required. Both types of conversions are crucial for writing robust and error-free code, ensuring that expressions evaluate as intended.

5.3.1.1 Automatic Type conversion:

If the operands are different types, the lower type is automatically converted to the higher type before the operation proceeds. The result is of the higher type.

Given below is the sequence of rules that are applied while evaluating expressions.

Op-1	Op-2	Result
long double	any	long double
double	any	double
float	any	float
unsigned long int	any	unsigned long int
long int	any	long int
unsigned int	any	unsigned int

The final result of an expression is converted to the type of the variable on the left of the assignment sign before 'assigning value to it.

However, the following changes are introduced during the final assignment.

- float to int causes truncation of the fractional part.
- double to float causes rounding of digits.
- long int to int causes dropping of the excess higher order bits.

5.3.1.2 Casting a value:

Casting a value is forcing a type conversion in a way that is different from the auto conversion. The process is called type cast. The general form of casting is

(type_desired) expression;

where type_desired: standard C data types and expression : constant, variable or expression.

Example:

```
#include<stdio.h>
void main()
{
int total_marks=500,ob_marks=234;
float per1 , per2;
per1 = (ob_marks / total_marks) * 100;
per2 = ( float ) (ob_marks / total_marks) * 100;
printf(“ Percentage without type casting = %.2f”,per1);
printf(“ Percentage After type casting = %.2f”,per2);
getch();
}
```

Output:

```
Percentage without type casting = 0.00
Percentage After type casting = 46.80
```

In expression per2 = (float) (ob_marks / total_marks) * 100; division is converted to float, otherwise decimal part of the result of division would be lost and per1 would represent a wrong figure or zero.

5.4 CONCLUSION

In this unit, we really dug into the various kinds of operators C offers—think arithmetic, relational, and logical—and, more importantly, how they're put to use. As we move forward, you'll see just how crucial these operators become when we tackle C's other core building blocks, like control statements and arrays.

We also spent a good deal of time on type conversions. Seriously, understanding these is absolutely vital! So often, programmers get those head-scratching, unexpected results (what we call logical errors), and more often than not, they trace back to improper type conversions or simply forgetting to explicitly "type cast" to the desired format. It's a subtle but significant detail that can save a lot of debugging time.

On top of that, this unit gave us a peek into C's famous shorthand. C earns its reputation as a compact language precisely because it lets us write lengthy expressions in a much shorter, more efficient way. The conditional operator is a perfect example, offering a neat, condensed alternative to writing out a full if/else statement (something we'll dive into next). And let's not forget those increment/decrement operators; they're little gems that really trim down your code when embedded within expressions.

Given that logical operators are foundational and pop up everywhere—from all sorts of looping constructs to those crucial

if/else statements we're about to explore—making sure you've got a solid grasp on them is, frankly, non-negotiable.

.

UNIT 6 CONTROL FLOW MECHANISMS

Structure

6.0 Introduction

6.1 Objectives

6.2 Decision Control Statements

6.2.1 The if Statement

6.2.2 The if else Statement

6.2.3 The switch Statement

6.3 Loop Control Statements

6.3.1 The while Loop

6.3.2 The do-while Statement

6.3.3 The for Loop

6.3.4 The Nested Loop

6.4 The Goto Statement

6.5 The Break Statement

6.6 The Continue Statement

6.7 The Exit Function

6.8 Conclusion

6.9 Unit based Questions /Answers

6.0 INTRODUCTION

In any computer program, you've got a series of instructions meant to be carried out. But here's the thing: most programs don't just run straight through from start to finish. Often, a C program needs to make a logical check at a certain point. Depending on what that test reveals, the program will then take one of several possible actions—this is what we call branching. Similarly, in a selection process, the program picks and executes a specific block of statements from a few available options. And if you ever need a group of statements to keep running repeatedly until a certain

condition is met, that's where looping comes in handy. All these dynamic behaviors are managed through various control statements.

These control statements are essentially what dictate the "flow of control" in your program. They let us specify the exact order in which the computer should execute each instruction. Generally, most high-level procedural programming languages rely on three fundamental types of control statements:

- Sequence instructions
- Selection (or decision) instructions
- Repetition (or loop) instructions

Sequence instruction is pretty straightforward: it just means executing one instruction after another, exactly in the order they appear in your code file. This sequential execution is actually C's default behavior; unless you explicitly tell it otherwise with a control statement, the computer will simply move from one instruction to the very next one in line.

Selection, on the other hand, is all about making choices. It means running different sections of code based on whether a specific condition is true or false, or based on the value of a variable. This capability is what allows a program to adapt and take different actions depending on various situations. C, for its part, offers three distinct structures specifically designed for these selection processes.

- if

- if…else
- switch

Repetition/Looping means executing the same section of code more than once. A section of code may either be executed a fixed number of times, or while some condition is true. ***C provides three looping statements:***

- while
- do…while
- for

This unit introduces you the decision and loop control statements that are available in C programming language along with some of the example programs.

This unit will explain to you the expressions and operators of language C.

6.1 OBJECTIVES

After completing this unit, you will be able to:

- work with different control statements;
- know the appropriate use of the various control statements in programming;
- transfer the control from within the loops;
- use the goto, break and continue statements in the programs; and
- write programs using branching, looping statements.

6.2 DECISION CONTROL STATEMENT

In a C program, making a decision essentially means the code can, at a specific point, jump to a different section of the program, all based on the outcome of an expression. C offers several ways to implement these decisions. The if...else statement is arguably the most crucial, as it allows the program to choose between two distinct courses of action. Interestingly, you can also use this statement without the else part, making it a simpler if statement for conditional execution. For situations where you need to branch to multiple alternative code sections based on the value of a single variable, C provides another powerful decision control statement: the switch statement.

6.2.1 The if Statement

This control structure serves a clear purpose: to execute an instruction, or even a whole block of instructions, *only* if a specific condition is met. When you use an `if` statement, the program first evaluates an expression. Then, depending on whether that expression (which could be a relational check or any other condition) turns out to be "true" or "false," it directs the program's flow to a particular statement or a designated group of statements.

Different forms of implementation if-statement are:

- Simple if statement
- If-else statement
- Nested if-else statement
- Else if statement

Simple if statement It is used to execute an instruction or block of instructions only if a condition is fulfilled.

The syntax is as follows:

```
if (condition)
statement;
```

where condition is the expression that is to be evaluated. If this condition is **true**, statement is executed. If it is **false**, statement is ignored (not executed) and the program continues on the next instruction after the conditional statement.

If we want more than one statement to be executed, then we can specify a block of statements within the curly bracets { }.

The syntax is as follows:

```
if (condition)
{
block of statements;
}
```

Example:

Write a program to calculate the net salary of an employee, if a tax of 15% is levied on his gross-salary if it exceeds Rs. 10,000/- per month.

```
/*Program to calculate the net salary of an employee */

#include<stdio.h>
main( )
{
float gross_salary, net_salary;
printf("Enter gross salary of an employee\n");
```

```
scanf("%f ",&gross_salary );
if(gross_salary<10000)
net_salary = gross_salary;
net_salary=gross_salary - 0.15*gross_salary;
printf("\nNet salary is Rs.%.2f\n", net_salary);
}
```

6.2.2 The if_else Statement

If…else statement is used when a different sequence of instructions is to be executed depending on the logical value (True / False) of the condition evaluated.

Its form used in conjunction with if and the syntax is as follows:

```
if (condition)
Statement _1;
else
Statement_ 2;
statement_3;
```

Or

```
if (condition)
{
Statements_1_Block;
}
else
{
Statements_2_Block;
}
Statements _3_Block;
```

If the initial condition turns out to be true, the set of instructions within the first block (let's call it `Statements_1_Block`) will run. However, if that condition is false, then the program will skip that first block and instead execute the `Statements_2_Block` that follows the `else` part of the statement. Regardless of which block was executed, the program's control then seamlessly moves on to `Statements_3`, continuing the regular, sequential flow of the program.

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int no;
clrscr();
printf("\n Enter Number :");
scanf("%d",&no);
if(no>0)
{
printf("\n\n Number is greater than 0 !");
}
else
{
if(no==0)
{
printf("\n\n It is 0 !"); }
else
{
printf("Number is less than 0 !");
```

```
}
}
getch();
}
```

Output:

```
Enter Number: 0
              It is 0!
```

6.2.3 The switch Statement

This is a fantastic tool for creating **multiple** or **multiway branches** within your program's decision-making process. You see, when you start using a bunch of nested if-else statements to check many different conditions, your code can quickly become a tangled mess. It gets harder to read, and maintaining it becomes a real headache. To neatly sidestep this problem, C offers the **switch statement**.

The switch statement works by taking the value of an expression and comparing it against a series of predefined "case" values. If it finds a match, the program's control is then neatly transferred to that specific point, executing the code associated with that particular case.

Syntax:

```
switch(expression)
{
case expr1:
statements;
break;
```

```
case expr2:
statements;
break;
-----------------------------
-----------------------------
-----------------------------
case exprn:
statements;
break;
default:
statements;
}
```

Switch case, break are keywords.

expr1, expr2 are known as 'case labels.'

Statements inside case expression need not to be closed in braces.

break statement causes an exit from switch statement.

default case is optional case. When neither any match found, it executes.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int no;
clrscr();
printf("\n Enter any number from 1 to 3 :");
scanf("%d",&no);
switch(no)
{
case 1:
```

```
printf("\n\n It is 1 !");
break;
case 2:
printf("\n\n It is 2 !");
break;
case 3:
printf("\n\n It is 3 !");
break;
default:
printf("\n\n Invalid number !");
}
getch();
}
```

Output:

```
Enter any number from 1 to 3 : 3
        It is 3 !
```

a). Rules for declaring switch case:

- The case label should be integer or character constant.
- Each compound statement of a switch case should contain break statement to exit from case.
- Case labels must end with (:) colon.

b). Advantages of switch case:

- Easy to use.
- Easy to find out errors.
- Debugging is made easy in switch case.
- Complexity of a program is minimized.

6.3 LOOP CONTROL STATEMENTS

Loop control statements are used when a section of code may either be executed a fixed number of times, or while some condition is true. C gives you a choice of three types of loop statements, while, do- while and for.

- The while loop keeps repeating an action until an associated condition returns false. This is useful where the programmer does not know in advance how many times the loop will be traversed.
- The do while loop is similar, but the condition is checked after the loop body is executed. This ensures that the loop body is run at least once.
- The for loop is frequently used, usually where the loop will be traversed a fixed number of times.

6.3.1 The while Loop

When in a program a single statement or a certain group of statements are to be executed repeatedly depending upon certain test condition, then while statement is used.

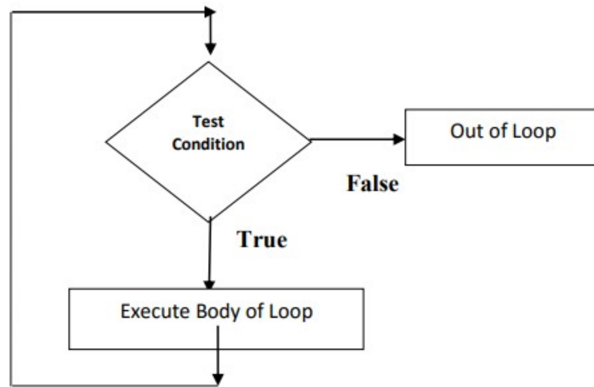
The syntax is as follows:

```
while (test condition)
{
body_of_the_loop;
}
```

Here, that test condition is essentially an expression that dictates how long the loop will keep running. The actual body of the

loop—which is a single statement or a group of statements tucked inside braces—gets executed repeatedly as long as this test condition evaluates to true. But the moment that condition turns false, the program's control immediately jumps out of the loop and moves on to the very first statement that follows the while loop.

It's worth noting that if the condition happens to be false right from the very beginning, the loop's body won't ever execute at all. That's why the while loop is sometimes called an entry-control loop; it always checks the condition before letting you into the loop's body..



Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int a;
        clrscr();
        a=1;
        while(a<=5)
{
        printf("MCMT \t");
        a+=1                    // i.e. a = a + 1
```

```
}
        getch();
}
```

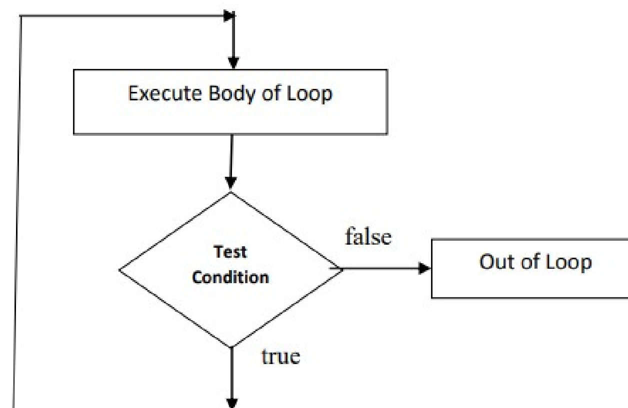
Output:

MCMT MCMT MCMT MCMT MCMT

6.3.2 The do-while Statement

There's another loop structure that's quite similar to the while statement, and that's the do...while loop. The main distinction between the two is that in a do...while loop, the condition that determines whether to keep looping is checked at the very end of each cycle, not at the beginning.

This difference has a crucial implication: the body of a do...while loop is guaranteed to execute at least once before the condition is even evaluated. After that initial run, the loop continues to repeat its body as long as the condition remains true. This is in contrast to a regular while loop, where if the condition is false from the start, the loop's body won't execute even once. That's precisely why they do...while loop is also known as an exit-control loop—it checks the condition as it's trying to exit the loop.



Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
          int a;
          clrscr();
          a=1;
do
{
          printf("MCMT\t");          // 5 times
          a+=1;                      // i.e. a = a + 1
          }
          while(a<=5);
          getch();
          }
```

Output:

```
MCMT  MCMT  MCMT  MCMT  MCMT
```

Infinite loop:

A looping process, in general, includes the following four steps:

- Setting of a counter.
- Execution of the statements in the loop.
- Testing of a condition for loop execution.
- Incrementing the counter.

Ideally, that test condition is designed to eventually signal the loop to stop and transfer control to the next part of the program. However, sometimes, for various reasons, it doesn't. When that happens, the program gets stuck in what's known as an infinite loop, where the loop's body just keeps executing endlessly. Obviously, you want to avoid these at all costs! If your program ever gets caught in such a loop, you can usually force it to stop by pressing Ctrl + C or Ctrl + Break on your keyboard.

Example:

```
#include<stdio.h>
void main()
{
        int i=1;
        while(i<=10)
{
        Printf(" i= %d\n",i);
}
}
```

This program will never terminate as variable i will always be less than 10. To get the loop terminated, an increment operation (i + +) will be required in the loop.

6.3.3 The for Loop

for statement makes it more convenient to count iterations of a loop and works well where the number of iterations of the loop is known before the loop is entered. The syntax is:

```
for (initialization; test condition; increment or decrement)
        {
        Statement(s);
        }
```

The primary goal of the for loop is much like the while loop: to repeatedly execute a statement or a block of code as long as a certain condition remains true. However, the for loop offers a neat advantage. It's specifically designed with built-in sections where you can easily handle the loop's setup (like initializing a counter

variable) and its ongoing maintenance (such as incrementing or decrementing that control variable). This makes it perfectly suited for performing actions that involve a counter or a fixed number of repetitions.

Here's how the for loop executes its actions:

1. Initialization: This is the very first thing that happens. You typically set up your counter variable here (e.g., int i = 0;). *Crucially, this step only executes once* when the loop begins.

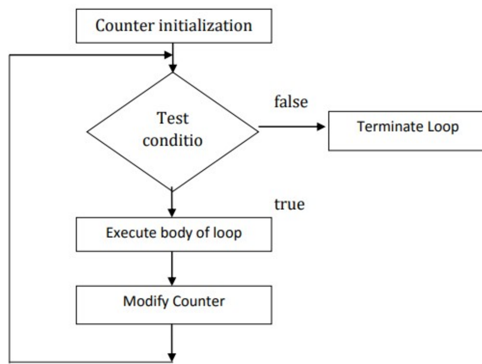
2. Condition Check: After initialization, the loop immediately checks its specified condition.

- If the condition is **true**, the loop will continue its operations.
- If the condition is **false**, the loop stops right then and there, and the program skips over the loop's body, moving on to whatever code comes after the loop.

3. Execute Loop Body: If the condition from step 2 was true, the statement or block of statements inside the loop (those curly braces {}) gets executed.

4. Update (Increment/Decrement): Once the loop's body has finished executing, the update expression (where you usually increment or decrement your counter variable, e.g., i++ or i--) is performed.

5. Loop Back: After the update, the control then jumps back to **step 2** (the condition check), and the cycle repeats until the condition eventually becomes false.



Features:

More concise

Easy to use

Highly flexible

More than one variable can be initialized.

More than one increments can be applied.

More than two conditions can be used.

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a;
clrscr();
for(i=0; i<5; i++);
{
Printf("MCMT!\t");          // 5 times
}
getch();
}
```

Output:

MCMT MCMT MCMT MCMT
MCMT

More About for Loop:

The for loop in C has several capabilities that are not found in other loop constructs. More than one variable can be initialized at a time in the for statement.

```
The Statement
p = 1;
for (n = 0; n < 17; + + n)
can be rewritten as for
(p = 1, n = 0; n < 17; ++ n)
```

The increment section may also have more than one part as given in the following

Example:

```
    for (n = 1, m = 50; n < = m; n = n+1, m = m-j)
{
    p = m/n;
    printf ("%d %d %d\n", n, m, p):
}
```

6.3.4 The Nested Loop

Loops within loops are called nested loops. An overview of nested while, for and do .. while loops is given below:

Nested while:

It is required when multiple conditions are to be tested.

The syntax is:

```
          while (condition 1)
    {
          ----------
          while (condition 2)
    {
          ----------
          while (condition n)
    {
          ----------
    }
    }
    }
```

Example:

Write a program to generate the following pattern:

```
1
1  2
1  2  3
1  2  3  4
```

```
/* Program to print the pattern */
#include<stdio.h>
main( )
{
int i,j;
for (i=1;i<=4;++i);
```

```
{
printf("%d\n",i);
for(j=1;j<=i;++j);
printf("%d\t",j);
}
}
```

Here, an inner for loop is written inside the outer for loop. For every value of i, j takes the value from 1 to i and then value of i is incremented and next iteration of outer loop starts ranging j value from 1 to i.

Example:

```
#include<stdio.h>
#include<conio.h>
main( )
{
Int i = 1, N;
Clrscr();
While(i<=5)
{
N=1;
While(N<=5)
Printf("%d", N);
}
printf("\n");
}
}
```

Ouput:

```
1 2 3 4 5
```

```
1  2  3  4  5
1  2  3  4  5
1  2  3  4  5
1  2  3  4  5
```

Nested for:

It is used when multiple set of iterations are required.

The syntax is:

```
for ( ; ; )
{
        ----------------
        ----------------
        ----------------
for ( ; ; )
{
        ----------------
        ----------------
for ( ; ; )
{
        ----------------

        ----------------
        ----------------
}
}
}
```

Example:

#include

```
#include
main()
{
int i , N;
clrscr();
for( i=1 ; i<= 5 ; i++ );
{
for( N=1 ;N<= 5 ; N++ );
{
printf(“ %d ” , N);
}
printf( “ \n”);
}
}
```

Ouput:

```
1  2  3  4  5
1  2  3  4  5
1  2  3  4  5
1  2  3  4  5
1  2  3  4  5
```

6.4 THE GOTO STATEMENT

The goto statement in C provides a way to unconditionally branch from one point in your program to another. Essentially, it breaks the normal, step-by-step sequential execution flow. To make this jump, the goto statement needs a label to pinpoint exactly where it should transfer control. A label is simply any valid variable name, and it must always be followed by a colon (:). You place this label immediately before the statement where you want the program's

execution to resume.

This statement is often referred to as a "jumping statement" because of its direct, non-sequential nature. While generally advised to be used sparingly, it can be quite useful for certain specific situations, particularly when you need to jump out of deeply nested loops. If an error occurs deep within multiple layers of loops, a simple break statement might only exit the innermost loop. In such complex scenarios, the goto statement offers a powerful way to make a clean, immediate exit from the entire structure.

The general forms of goto and label statements are:

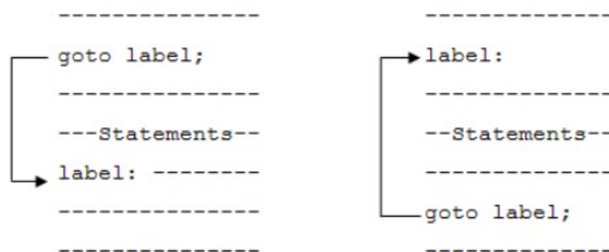


Figure:

```
while (condition)
{
   for( ;  ;  ;)
   {
      _ _ _
      _ _ _ _
      goto err;
      _ _ _ _
      _ _ _
   }
  err:
}
```

Example:

```
#include<stdio.h>
#include<conio.h>
main()
{
int i=1, j;
clrscr();
while(i<=3)
```

```
{
for(j=1;
j<=3; j++)
{
printf(" I=%d \t J=%d \n", i , j);
if(j==2) goto stop;
}
i = i + 1;
}
getch();
}
```

Output:

```
I=1     J=1
I=1     J=2
```

6.5 THE BREAK STATEMENT

Sometimes, it is required to jump out of a loop irrespective of the conditional test value. ***Break*** statement is used inside any loop to allow the control jump to the immediate statement following the loop.

The syntax is: ***break;***

When you're dealing with nested loops, the break statement acts specifically to exit only the innermost loop in which it's placed. It effectively jumps control out of that immediate loop. You'll find the break statement useful across all types of loops—whether it's a while, do-while, or for loop—and it's also commonly used within switch statements. Let's take a look at a program to see how the break statement works in action.

Example:

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

```
int i;
clrscr();
for(i=1; i<=20 ; i++);
{
if(i>5)
break;
printf("%d",i); // 5 times only
}
printf(" \nOut of loop");
getch();
}
```

Output:

```
1 2 3 4 5
Out of loop
```

6.6 THE CONTINUE STATEMENT

Sometimes, you'll find yourself in a situation where you need to bypass a specific section of a loop's body, but only when certain conditions are met. To handle this kind of scenario, C provides the `continue` statement.

While the `continue` statement operates similarly to the `break` statement in terms of altering loop flow, there's a key distinction: `continue` doesn't actually terminate the loop. Instead, when `continue` is encountered, it causes the current iteration to immediately stop, skipping any remaining statements within that iteration, and then jumps directly to the beginning of the "next" iteration. Essentially, it lets you "skip over" a part of the loop's body and move right on to the next cycle.

The syntax is: ***continue;***

Figure:

```
while (condition)
{
    continue;
}
```

Example:

```
#include<stdio.h>
#include<conio.h>
main()
{
        int i;
        clrscr();
        for(i=1; i<=10; i++)
{
        if( i>=6 && i <=8 )

        continue;
        printf("\t%d",i); // 6 to 8 is omitted
        }
        getch();
        }
```

Output: 1 2 3 4 5 9 10

6.7 THE EXIT( ) FUNCTION

The exit() function is used to terminate the execution of 'C'

program. It is a standard library function and uses header file stdlib.h.
The general form of exit( ) function is

exit (int status);

The difference between break and exit( ) is that former terminates the execution of loop in with it is written while exit( ) terminates the execution of program itself.

The status (in the general form of exit) is a value returned to the operation system after the termination. of the program.

The value zero "0" indicates that the termination is normal while value one "1" (Non-Zero) indicates different types of errors.

Example:

```
#include<stdio.h>
#include<conio.h>
main()
{
        int i;
        clrscr();
        for(i=1; ; i++)
{
        if(i>5)
        exit(0);
        printf("%d",i); // 5 times only
}
        printf(" \nOut of loop"); // control will not reached
        here
        getch();
}
```

Output:

```
1 2 3 4 5
```

6.8 CONCLUSION

Programs rarely just follow a straight, linear path of instructions from start to finish. More often than not, they need the ability to either repeat a section of code multiple times based on requirements or make crucial decisions to alter their flow. This is precisely why C equips us with powerful control and looping statements. In this unit, we've explored the different looping structures C offers: the `while` loop, the `do...while` loop, and the `for` loop.

Beyond just repeating code, we also looked at statements that give us finer control. The `break` statement, for instance, gives us the power to exit a loop even if its natural termination condition hasn't been met. This can be super handy for ending an infinite loop or simply forcing an early exit when needed. Then there's the `continue` statement, which causes the program to skip the remainder of the “current” iteration of a loop and immediately jump to the start of the ‘next’ one. And finally, the `goto` statement allows for an unconditional jump to any other point within your program. While it offers direct control, it's a feature you should use with great care, as its execution ignores any structural nesting, which can sometimes make code harder to follow or debug..

UNIT 7 ARRAYS AND STRINGS

Structure

7.0 Introduction

7.1 Objectives

7.2 Array Declaration

7.2.1 Syntax of Array Declaration

7.2.2 Size Specification

7.2.3 Accessing Array Elements

7.3 Array Initialization

7.3.1 Initialization of Array Elements in the Declaration

7.3.2 Character Array Initialization

7.4 Subscript

7.5 Passing Arrays to Functions

7.6 Processing the Arrays

7.7 Multi-Dimensional Arrays

7.6.1 Multi-Dimensional Arrays Declaration

7.6.2 Initialization Two-Dimensional Arrays

7.8 Declaration and Initialization of Strings

7.9 Display of Strings Using Different Formatting Techniques

7.10Array of Strings

7.11Built-in String Functions and Applications

7.11.1 Strlen Function

7.11.2 Strcpy Function

7.11.3 Strcmp Function

7.11.4 Strcat Function

7.11.5 Strlwr Function

7.11.6 Strrev Function

7.11.7 Strspn Function

7.12Other String Functions

7.13Conclusion

7.14Unit based Questions /Answers

7.0 INTRODUCTION

Let's face it, as programs grow larger and more complex, managing all that data can quickly become a real headache. Variable names might get longer just to stay unique, and frankly, dealing with an ever-increasing list of distinct variable names can make it tough for a programmer to focus on the truly important task of writing correct and efficient code. This is where arrays come in as a lifesaver. They offer a neat mechanism for grouping several related data items under a single identifier, which dramatically simplifies the whole data management process.

Think about it: many programming challenges involve working with multiple, related pieces of data that share common traits—like a list of student marks, a series of temperatures, or a collection of enrolment numbers. While you could create individual variables for each item, that would be a tedious and cumbersome approach. Arrays provide the perfect solution for handling these collections efficiently.

Now, let's talk about character arrays, which we commonly refer to as strings. In C, a string is fundamentally just a single-dimensional array designed to hold characters. C doesn't actually have an inherent "string" data type built-in; it relies on these character arrays. Sometimes, you'll need to process individual characters within a string. However, for many other problems, you'll want to manipulate strings as complete, cohesive entities. Luckily, C provides a powerful set of string-oriented library functions for just

this purpose. Most C compilers include these functions (for tasks like comparing strings, copying them, or concatenating them) within the `<string.h>` header file. It's important to remember that these string functions typically operate on character arrays that are null-terminated—meaning they have a special null character (`\0`) at the very end to mark where the string finishes.

This unit will explain to you the Array elements and String Functions of language C.

7.1 OBJECTIVES

After completing this unit, you will be able to:

- declare and use arrays of one dimension;
- initialize arrays;
- use subscripts to access individual array elements;
- write programs involving arrays;
- do searching and sorting; and
- handle multi-dimensional arrays.
- define, declare and initialize a string;
- discuss various formatting techniques to display the strings; and
- discuss various built-in string functions and their use in manipulation of strings.

7.2 ARRAY DECLARATION

An array is essentially a specific kind of data structure that allows you to store a fixed-size, ordered collection of items, all of which must be of the same type. While it's used for holding a collection

of data, it's often more helpful to simply think of an array as a group of variables, all sharing the same data type.

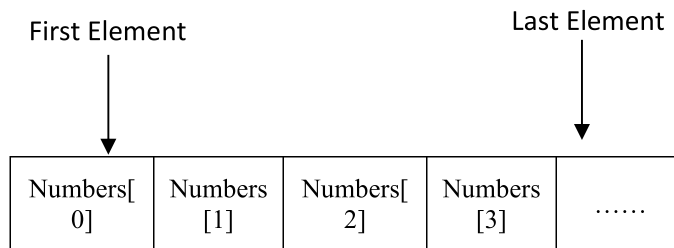
Instead of going through the tedious process of declaring many individual variables—like `number0`, `number1`, and so on, all the way up to `number99`—you can just declare one single array variable, perhaps named `numbers`. Then, to refer to each individual spot, you simply use an index (like `numbers[0]`, `numbers[1]`, and so forth, up to `numbers[99]`). This way, you access a specific element in the array using its position, or index.

Under the hood, all the elements within an array are stored in memory locations that are right next to each other, or contiguous. The very first element always sits at the lowest memory address, and the last element resides at the highest address within that allocated block..

The characteristic features of an array.

- Array is a data structure storing a group of elements, all of which are of the same data type.
- All the elements of an array share the same name, and they are distinguished from one another with the help of an index.
- Random access to every element using a numeric index (subscript).
- A simple data structure, used for decades, which is extremely useful.
- Abstract Data type list is frequently associated with the array data structure.

The declaration of an array is just like any variable declaration with additional size part, indicating the number of elements of the array.



7.2.1 Syntax of Array Declaration

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows:

```
data_type arrayName [ arraySize ];
```

This particular setup is what we refer to as a single-dimensional array. When you're declaring one, remember that the `arraySize` (how big it is) absolutely has to be a whole number, a constant, and larger than zero. As for the `type` of data it holds, that can be any of the valid data types C supports.

The following are some of declarations for arrays:

```
int char [80];
float farr [500];
static int iarr [80];
char charray [40];
```

There are two restrictions for using arrays in C:

- The amount of storage for a declared array has to be specified at compile time before execution. This means that an array has a fixed size.

- The data type of an array applies uniformly to all the elements; for this reason, an array is called a homogeneous data structure.

7.2.2 Size Specification

When declaring the size of an array, it's a really good practice to use a symbolic constant (like a `#define`'d value) instead of just punching in a fixed integer number. Why? Because it makes modifying your program much, much easier down the line. If you ever need to change the maximum size of that array, you can simply update the value of that one symbolic constant, and every place in your code that refers to it will automatically adjust. It saves a lot of hassle!.

To declare size as 50 use the following symbolic constant, SIZE, defined:

```
#define SIZE 50
```

This example shows how to declare and read values in an array to store marks of the students of a class.

Example:

Write a program to declare and read values in an array and display them.

```
/* Program to read values in an array*/
# include < stdio.h >
# define SIZE 5                    /* SIZE is a symbolic
constant */
```

```
main ( )
{
int i = 0;                              /* Loop variable */
int stud_marks[SIZE];  /* array declaration */
/* enter the values of the elements */
for( i = 0;i<size;i++)
{
printf (“Element no. =%d”,i+1);
printf(“ Enter the value of the element:”);
scanf(“%d”,&stud_marks[i]);  }  printf(“\nFollowing
are the values stored in the corresponding array
elements: \n\n”); for( i = 0; i<size;i++)
{
printf(“Value    stored    in    a[%d]    is    %d\n”i,
stud_marks[i]);
}
}
```

OUTPUT:

```
Element no. = 1 Enter the value of the element = 11
Element no. = 2 Enter the value of the element = 12
Element no. = 3 Enter the value of the element = 13
Element no. = 4 Enter the value of the element = 14
Element no. = 5 Enter the value of the element = 15
```

Following are the values stored in the corresponding array elements:

```
Value stored in a[0] is 11
Value stored in a[1] is 12
Value stored in a[2] is 13
```

```
Value stored in a[3] is 14
Value stored in a[4] is 15
```

7.2.3 Accessing Array Elements

To get to a specific item within an array, you essentially "index" the array's name. This means you put the index number of the element you want right inside square brackets, immediately following the array's name.

Example:

```
double salary = balance[9];
```

The above statement will take the 10th element from the array and assign the value to salary variable.

Example:

To use concepts viz. declaration, assignment, and accessing 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;
}
```

The following Result:

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

7.3 ARRAY INITIALIZATION

Arrays can be initialized at the time of declaration. The initial values must appear in the order in which they will be assigned to the individual array elements, enclosed within the braces and separated by commas.

7.3.1 Initialization of Array Elements in the Declaration

The values are assigned to individual array elements enclosed within the braces and separated by comma.

Syntax of array initialization is:

data type array-name [size] = {val 1, val 2, ...val n};

val 1 is the value for the first array element, val 2 is the value for the second element, and val n is the value for the n array element.

Note: When initializing the values at the time of declaration, then there is no need to specify the size. Let the given example:

```
int digits [10] = {1,2,3,4,5,6,7,8,9,10};
int digits[ ] = {1,2,3,4,5,6,7,8,9,10};
int vector[5] = {12,-2,33,21,13};
float temperature[10] ={ 31.2, 22.3, 41.4, 33.2, 23.3, 32.3, 41.1, 10.8, 11.3, 42.3};
double width[ ] = { 17.33333456, -1.212121213, 222.191345 };
int height[ 10 ] = { 60, 70, 68, 72, 68 };
```

7.3.2 Character Array Initialization

In C, strings are essentially implemented as arrays of characters. What's interesting is that they're initialized a bit differently than other arrays. There's a special little character, called the null character (`\0`), which C *automatically* places at the very end of every string. So, when you assign a string constant to an external or static character array, you often don't need to specify the array's size yourself. The compiler is smart enough to figure it out automatically, and that calculated size will always include room for that crucial `\0` character tacked on at the end.

Example:

char thing [3] = “TIN”;
char thing [ ] = “TIN”;

The above two statements the assignments are done differently. The first statement is not a string but simply an array storing three characters ‘T’, ‘I’ and ‘N’ and is same as writing: char thing [3] = {‘T’, ‘I’, ‘N’}; whereas, the second one is a four character string TIN\0.

7.4 SUB SCRIPT

To the individual element in an array, a subscript is used. to the statement used in

Example:

scanf (“ % d”, &stud_marks[i]);

Subscript is an integer type constant or variable name whose value ranges from 0 to SIZE - 1 where SIZE is the total number of elements in the array.

To individual elements of an array of size 5: Consider the following declarations: char country[ ] = “India”; int stud[ ] = {1, 2, 3, 4, 5}; Here both arrays are of size 5. This is because the country is a char array and initialized by a string constant “India” and every string constant is terminated by a null character ‘\0’.

Example:

```
/* Program to find the maximum marks among the marks of 10
students*/
# include < stdio.h >
# define SIZE 10                /* SIZE is a symbolic constant */
main ( )
{
int i = 0;
int max = 0;
int stud_marks[SIZE];                /* array declaration */

/* enter the values of the elements */
for( i = 0;i<size;i++)
{
printf("Student no.=%d",i+1);
printf(" Enter the marks out of 50:");
scanf("%d",&stud_marks[i]);
}
/* find maximum */
for( i = 0;i<size;i++)
{
If(stud_marks[i]>max)
max = stud_marks[ i ];
}
printf("\n\nThe maximum of the marks obtained among all the 10
students is: %d ",max); }
```

OUTPUT:

```
Student no. = 1  Enter the marks out of 50: 10
Student no. = 2  Enter the marks out of 50: 17
```

Student no. = 3 Enter the marks out of 50: 23

Student no. = 4 Enter the marks out of 50: 40

Student no. = 5 Enter the marks out of 50: 49

Student no. = 6 Enter the marks out of 50: 34

Student no. = 7 Enter the marks out of 50: 37

Student no. = 8 Enter the marks out of 50: 16

Student no. = 9 Enter the marks out of 50: 08

Student no. = 10 Enter the marks out of 50: 37

7.5 PASSING ARRAYS TO FUNCTIONS

When you need to send a single-dimensional array as an argument to a function, C gives you a few options for how you declare that formal parameter inside the function. Interestingly, all three common declaration methods end up achieving the same result: they each signal to the compiler that the function is expecting to receive an integer pointer. The good news is, you can apply a similar approach when passing multi-dimensional arrays as formal parameters as well.

Way-1

Formal parameters as a pointer:

```
void myFunction(int *param)
{
 .
.
.
}
```

Way-2

Formal parameters as a sized array:

```
void myFunction(int param[10])
{
.
.
.
}
```

Way-3

Formal parameters as an unsized array:

```
void myFunction(int param[])
{
.
.
.
}
```

Example:

Okay, let's look at an example. Imagine a function designed to calculate an average: it takes an array as input, along with one other argument, and then uses those inputs to figure out and return the average of the numbers provided in that array. Here's how it would generally work:

```
double getAverage(int arr[], int size)
    {
        int i;
```

```
        double avg;
        double sum;

        for (i = 0; i < size; ++i)
    {
        sum += arr[i];
    }
        avg = sum / size;
        return avg;
    }
```

Now, let us call the function:

```
#include<stdio.h>
/* function declaration */

double getAverage(int arr[], int size);
int main ()
{
/* an int array with 5 elements */
int balance[5] = {1000, 2, 3, 17, 50};
double avg;

/* pass pointer to the array as an argument */
avg = getAverage( balance, 5 ) ;

/* output the returned value */
printf( "Average value is: %f ", avg );
return 0;
}
```

When the code is compiled together and executed, it produces the

following result: Average value is: 214.400000

7.6 PROCESSING THE ARRAYS

For certain applications the assignment of initial values to elements of an array is required. This means that the array be defined globally (extern) or locally as a static array.

Example:

Write a program to display the average marks of each student, given the marks in 2 subjects for 3 students.

```
/* Program to display the average marks of 3 students */
# include < stdio.h >
# define SIZE 3
main()
{
int i = 0;
float stud_marks1[SIZE];       /* subject 1array declaration */
float stud_marks2[SIZE];       /*subject 2 array declaration */
float total_marks[SIZE];
float avg[SIZE];
printf("\n Enter the marks in subject-1 out of 50 marks: \n");
for( i = 0;i<size;i++)
{
printf("Student no. =%d",i+1);
printf(" Please enter the marks= ");
scanf("%f",&stud_marks2[i]);
}
for(i=0;i<size;i++)
{
```

```
total_marks[i]=stud_marks1[i]+ stud_marks2[i];
avg[i]=total_marks[i]/2;
printf("Student no.=%d, Average= %f\n",i+1, avg[i]);
}
}
```

OUTPUT:

Enter the marks in subject-1out of 50 marks:

Student no. = 1 Enter the marks= 23

Student no. = 2 Enter the marks= 35

Student no. = 3 Enter the marks= 42

Enter the marks in subject-2 out of 50 marks:

Student no. = 1 Enter the marks= 31

Student no. = 2 Enter the marks= 35

Student no. = 3 Enter the marks= 40

Student no. = 1 Average= 27.000000

Student no. = 2 Average= 35.000000

Student no. = 3 Average= 41.000000

7.7 MULTI-DIMENSIONAL ARRAYS

C programming language allows multidimensional arrays. Here is the general form of a multidimensional array declaration:

```
type name[size1][size2]...[sizeN];
```

Example:

The following declaration creates a three-dimensional integer array:

```
int three dim[5][10][4];
```

Imagine you're diving into the world of programming a chess game. A standard chessboard, as we know, is an 8-by-8 grid. So, what's the best way to digitally represent that? You'd likely turn to a two-dimensional array. This structure perfectly mirrors the chessboard, allowing you to store the positions of all the chess pieces. With a two-dimensional array, you use two indices to pinpoint any single square, which is actually quite similar to how "algebraic notation" works in chess circles to record games and problems. It's a pretty intuitive fit!

Here's a cool principle: in theory, there's absolutely no limit to how many subscripts (or dimensions) an array can have. Any array with more than one dimension falls under the umbrella of multi-dimensional arrays. While we humans might struggle to visualize objects beyond three dimensions, representing and working with highly multi-dimensional arrays poses no problem whatsoever for computers. They handle it with ease!

7.7.1 Multi-Dimensional Arrays Declaration

To declare an array of two dimensions as follows:

datatype array_name[size1][size2];

In the example of, variable_type is the name of some type of variable, such as int. Also, size1 and size2 are the sizes of the array’s first and second dimensions, respectively.

Example:

Remember, because C arrays are zero-based, the indices on each side of the chessboard array run 0 through 7, rather than 1 through 8. The effect is the same: a two-dimensional array of 64 elements. int chessboard [8][8];

7.7.2 Initialization Two-Dimensional Arrays

The simplest kind of multi-dimensional array you'll encounter is the two-dimensional array. You can essentially think of it as a list, where each item on that list is actually another one-dimensional array.

To declare a two-dimensional integer array of size [x][y], To write something as:

type arrayName [x][y];

For an `m x n` array, you're looking at a total of `m * n` individual elements. To figure out how much space it will take up in memory, you simply multiply that total number of elements by the size of each individual element. This calculated amount is precisely how much memory needs to be set aside, or "reserved," for your array. When these elements are actually stored, they're typically laid out in memory row by row.

int table [2] [3] = { 1,2,3,4,5,6 };

It means that element

table [0][0] = 1;

table [0][1] = 2;

table [0][2] = 3;

table [1][0] = 4;

table [1][1] = 5;

table [1][2] = 6;

The neutral order in which the initial values are assigned can be altered by including the groups in { } inside main enclosing brackets, the initialization as:

int table [2] [3] = { {1,2,3}, 36
{4,5,6} };

When you're initializing multi-dimensional arrays, the values placed within the innermost set of braces are assigned to the array elements where the *last subscript* (or index) changes most quickly. If you provide fewer values than a row can hold, the remaining elements in that row will automatically be filled with zeros. It's important to remember, though, that you can't supply more values than the defined size of the row; you must stick within its specified capacity.

int table [2] [3] = { { 1, 2, 3},{ 4}};

It assigns as

table [0][0] = 1;

table [0][1] = 2;

table [0][2] = 3;

table [1][0] = 4;

table [1][1] = 0;

table [1][2] = 0

7.8 DECLARATION AND INITIALIZATION OF STRINGS

Strings in C are essentially just one-dimensional arrays made up of characters, with one crucial difference: they're always ended by a special null character, written as '\0'. This '\0' acts like a sentinel, signaling the very end of the string.

So, when you declare and initialize a string, say, with the word "Hello," you're actually creating a null-terminated string that contains those five letters followed immediately by the '\0'. This means that the character array you use to hold "Hello" needs to be one slot larger than the number of characters in the word itself, just to make room for that essential null terminator.

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

The rule of array initialization, they can write the statement as:

```
char greeting[] = "Hello";
```

In C, strings are essentially collections of characters, digits, and symbols, all bundled together and enclosed within quotation marks. Simply put, you can think of a string as a "character array." A crucial detail is that the very end of every string is always marked by a special null character, represented as '\0'.

The memory of the string in C/C++:

Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

The null character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array.

string:

```
#include
int main ()
{ char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
printf("Greeting message: %s\n", greeting );
return 0;
}
```

Result:

Greeting message: Hello

Initialization of Strings:

A string in C is simply a sequence of characters. To declare a string, specify the data type as char and place the number of characters in the array in square brackets after the string name.

The syntax is:

char string-name[size];

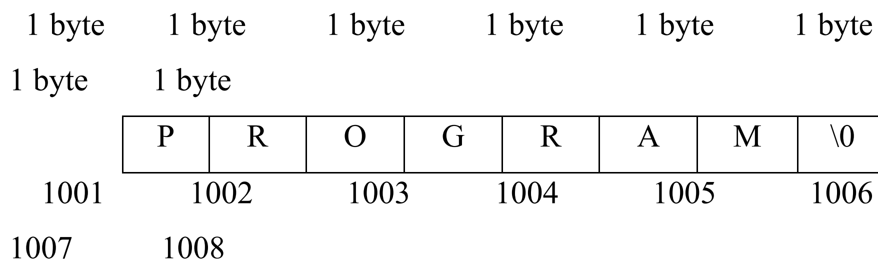
Example:

char name[20];
char address[25];
char city[15];

The string can be initialized as:

char name[8] = {'P', 'R', 'O', 'G', 'R', 'A', 'M', '\0'};

Each character of string occupies 1 byte of memory (on 16 bit computing). The size of character is machine dependent, and varies from 16 bit computers to 64 bit computers. The characters of strings are stored in the contiguous (adjacent) memory locations.



The good news is that the C compiler is smart enough to automatically insert the null character (`\0`) at the end of every string literal you use. This means you generally don't have to manually initialize that null terminator yourself.

When working with string constants (those bits of text enclosed in double quotes), you have a couple of straightforward ways to

handle them. You can directly assign them to `char` pointers. Alternatively, you can assign a string constant to a `char` array—you might even omit the size specification and let the compiler figure it out, or you can explicitly define the size. Just remember, if you specify the size, always make sure to leave that extra spot for the essential null character! Now, let's look at a couple of code examples to see this in action:

```
/* Fragment 1 */
{
char *s;
s=hello”;
printf(“%s\n”,s);
}
/* Fragment 2 */
{
char s[100];
  strcpy(s, “ hello”);
  printf(“%s\n”,s);
}
```

These two fragments produce the same output, but their internal behaviour is quite different. In fragment 2, you cannot say s = "hello";. To understand the differences, you have to understand how the string constant table works in C.

Example:

Write a program to read a name from the keyboard and display message Hello onto the monitor Program.
/*Program that reads the name and display the hello along with your name*/

```
#include main()
{
char name[10];
printf("\nEnter Your Name : ");
scanf("%s", name);
printf("Hello %s\n", name);
}
```

OUTPUT:

```
Enter Your Name:  Raghav
  Hello Raghav
```

7.9 ARRAY OF STRINGS

Array of strings are multiple strings, stored in the form of table. Declaring array of strings is same as strings, except it will have additional dimension to store the number of strings. Syntax is:

```
char array-name[size][size];
```

Example:

```
char names[5][10];
```

where names is the name of the character array and the constant in first square brackets will gives number of string we are going to store, and the value in second square bracket will gives the maximum length of the string.

7.10 BUILT-IN STRING FUNCTION AND APPLICATIONS

The header file contains some string manipulation functions. The following is a list of the common string managing functions in C.

7.10.1 Strlen Function

The strlen function returns the length of a string. It takes the string name as argument. The syntax is as follows:

n = strlen (str);

where str is name of the string and n is the length of the string, returned by strlen function.

7.10.2 Strcpy Function

In C, you cannot simply assign one character array to another. You have to copy element by element. The string library contains a function called strcpy for this purpose. The strcpy function is used to copy one string to another.

The syntax is:

strcpy(str1, str2);

where str1, str2 are two strings. The content of string str2 is copied on to string str1.

7.10.3 Strcmp Function

The strcmp function, found within C's standard string library, is specifically designed to compare two strings. It does this by checking them character by character, starting from the beginning. The comparison halts either when it finds a difference in the ASCII values of corresponding characters, or when it reaches the end of either string.

What it returns is an integer value. This integer tells you about the relationship between the two strings:

- If strcmp returns zero (0), it means the two strings are identical.
- A negative value indicates that the first string is "less than" the second string (alphabetically, based on ASCII values).
- A positive value signifies that the first string is "greater than" the second string.

The syntax is:

n = strcmp(str1, str2);

where str1 and str2 are two strings to be compared and n is returned value of differed characters.

7.10.4 Strcat Function

The strcat function is used to join one string to another. It takes two strings as arguments; the characters of the second string will be appended to the first string.

The syntax is:

strcat(str1, str2);

where str1 and str2 are two string arguments, string str2 is appended to string str1.

7.10.5 Strlwr Function

The strlwr function converts upper case characters of string to lower case characters.

The syntax is:

strlwr(str1);

where str1 is string to be converted into lower case characters.

7.10.6 Strrev Function

The strrev funtion reverses the given string.

The syntax is:

strrev(str);

where string str will be reversed.

7.10.7 Strspn Function

The strspn function returns the position of the string, where first string mismatches with second string.

The syntax is:

n = strspn (first, second);

where first and second are two strings to be compared, n is the number of character from which first string does not match with second string.

7.11 OTHER STRING FUNCTIONS

strncpy function

The **strncpy** function same as strcpy. It copies characters of one string to another string up to the specified length.

The syntax is:
strncpy(str1, str2, 10);
where str1 and str2 are two strings. The 10 characters of string str2 are copied onto string str1.

stricmp function

The stricmp function is same as strcmp, except it compares two strings ignoring the case (lower and upper case).

The syntax is:

n = stricmp(str1, str2);

strncmp function The strncmp function is same as strcmp, except it compares two strings up to a specified length.

The syntax is:

n = strncmp(str1, str2, 10);

where 10 characters of str1 and str2 are compared and n is returned value of differed characters.

strchr function

The strchr funtion takes two arguments (the string and the character whose address is to be specified) and returns the address of first occurrence of the character in the given string.

The syntax is:

cp = strchr (str, c);

where str is string and c is character and cp is character pointer.

7.12 CONCLUSION

In this unit, we've learned that as programs expand in size and complexity, effectively managing data becomes increasingly challenging. Relying solely on unique, often lengthy variable names for every piece of data can quickly overwhelm a programmer and divert focus from the critical task of accurate coding. This is where arrays prove invaluable. They provide a unified name for a collection of data items, allowing individual members to be accessed simply by their index. We've explored the fundamental purpose of arrays, how to declare them, and how to assign values. A key characteristic is that all array elements are stored in sequential memory locations, and without exception, C arrays are indexed starting from 0, going up to one less than their declared size.

A crucial point about array declarations is their rigidity regarding size: the dimensions you specify must be constant expressions that can be determined when the program is compiled, not during runtime. Regarding initialization, global and static array elements are automatically set to 0 by default, whereas elements in automatic (local) arrays will contain whatever "garbage" values were previously in those memory locations. Within C, a character array is specifically employed to represent a character string, with its end explicitly marked by a byte set to 0, commonly known as a NULL character (`\0`).

At their heart, strings are simply sequences of characters. For them to behave correctly and be usable with C's string functions, they absolutely *must* be null-terminated. This means you always need to account for that `\0` when dealing with strings, especially if you're allocating memory dynamically for them. The `string.h` library provides a wealth of useful functions for string manipulation. However, losing that crucial `\0` character can lead to significant and hard-to-track bugs. Therefore, it's vital to ensure you always copy the `\0` when duplicating strings, include it when creating new ones, and verify that any receiving string is adequately sized to hold the source string plus its `\0`. Lastly, if you're pointing a character pointer to a sequence of characters, confirm that those characters are properly terminated with a `\0`.

BLOCK III: STRINGS, TOOLS FOR MODULAR PROGRAMMING AND POINTERS

UNIT 8 FUNCTIONS

Tools for Modular Programming

8.0 Introduction

8.1 Objectives

8.2 Definition of a Function

8.3 Declaration of a Function

8.4 Function Prototypes

8.5 Calling a Function

8.6 Function Arguments

8.7 The Return Statement

8.7.1 Call by Value

8.7.2 Call by Reference

8.8 Types of Variables and Storage Classes

8.8.1 Automatic Variables

8.8.2 External Variables

8.8.3 Static Variables

8.8.4 Register Variables

8.9 Types of Function Invoking

8.10 Recursion

8.11 Conclusion

8.12 Unit based Questions /Answers

8.0 INTRODUCTION

To make programming simple and easy to debug, we break a larger program into smaller subprograms which perform 'well defined tasks'. These subprograms are called functions. So far we have defined a single function main ( ).

After reading this unit you will be able to define many other functions and the main( ) function can call up these functions from several different places within the program, to carry out the required processing.

Functions are very important tools for Modular Programming, where we break large programs into small subprograms or modules (functions in case of C). The use of functions reduces complexity and makes programming simple and easy to understand.

In this unit, we will discuss how functions are defined and how are they accessed from the main program? We will also discuss various types of functions and how to invoke them. And finally you will learn an interesting and important programming technique known as Recursion, in which a function calls within itself.

This unit will explain to you the functions of language C.

8.1 OBJECTIVES

After completing this unit, you will be able to:

- the need of functions in the programming;
- how to define and declare functions in 'C' Language;
- different types of functions and their purpose;
- how the functions are called from other functions;
- how data is transferred through parameter passing, to functions and the Return statement; recursive functions; and
- the concept of 'Call by Value' and its drawbacks.
- understand the concept and use pointers;
- address and use of indirection operators;
- make pointer type declaration, assignment and initialization;
- use null pointer assignment;
- use the pointer arithmetic;
- handle pointers to functions;

- see the underlying unit of arrays and pointers; and
- understand the concept of dynamic memory allocation

8.2 DEFINITION OF A FUNCTION

A function is a group of statements that together perform a task. Every C program has at least one function, which is main(), and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.

A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

The C standard library provides numerous built-in functions that your program can call. For example, strcat() to concatenate two strings, memcpy() to copy one memory location to another location, and many more functions. A function can also be referred as a method or a sub-routine or a procedure, etc.

Functions are the C building blocks where every program activity occurs. It is a self contained program segment that carries out some specific, well-defined task. Every C program must have a function. One of the function must be main( ).

The general form of a function definition in C programming language is as follows:
return_type function_name(parameter list)

```
{
        body of the function
}
```

C functions can be classified into two categories.

Library functions: Predefined in the standard library of C. Need is just to include the library.

User defined functions: It has to be-developed by the user at the time of program writing.

Need of user Defined Functions

If a program is divided into functional parts, then each part may be independently coded and later combined into a single unit. This approach clearly results in a number of advantages.

- Length of program can be reduced by using function.
- Reusability of function increases.
- It is easy to use.
- Debugging is more suitable (easier) for programs.
- It is easy to understand the actual logic of a program.
- Highly suited in case of large programs.
- By using functions in a program, it is possible to construct modular and structured programs.

Example:

```
/* function returning the max between two numbers */
int max(int num1, int num2)
{
/* local variable declaration */
int result;
```

```
if (num1 > num2)
result = num1;
else
result = num2;
return result;
}
```

8.3 DECLARATION OF A FUNCTION

Before defining the function, it is appropriate to declare the function along with its prototype. In function prototype, the return value of function, type, and number of arguments are specified. The declaration of all functions statement should be first statements in main( ) or we can also declare globally for accessing all function within program. The general form of function declaration is

```
<return_type> <function_name> ( [<argument_list>] );
```

A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately. A function declaration has the following parts:

```
return_type function_name(parameter list );
```

For the above defined function max(),the function declaration is as follows:

```
int max (int num1, int num2);
```

Parameter names are not important in function declaration, only their type is required, so the following is also a valid declaration:

```
int max(int , int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

8.4 FUNCTION PROTOTYPES

Function prototypes are desirable because they facilitate error checking between calls to a function and corresponding function definition. They also help the compiler to perform automatic type conversions on function parameters. When a function is called, actual arguments are automatically converted to the types in function definition using normal rules of assignment.
The function Definition is, the task assigned to the function, that user declare. The general form of a function definition is:

```
return_type function_name (declarations of formal argument list)
{
        local variable declarations;
        executable statement 1;
        executable statement 2;
        ----------------------------
        ----------------------------
        ----------------------------
        executable statement n;
        return (expression);
}
```

Where return_type represents the data type of the value which is returned. The type specification can be omitted if the function returns an integer or a character. The formal argument list is a list of variables separated by commas that receive the values from main program when function is called.

The last statement in the body of function is return (expression). It is used to return the computed result, if any, to the calling program.

8.5 CALLING A FUNCTION

A function can be called by specifying its name followed by a list of arguments enclosed in parentheses and separated by commas. If a function call does not require any arguments, an empty pair of parentheses must follow the function name.

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value.

Example:

```
#include<stdio.h>
/* function declaration */
        int max(int num1, int num2);
        int main ()
{
/* local variable definition */
        int a = 100;
        int b = 200;
        int ret;
```

```
/* calling a function to get max value */
        ret = max (a, b);
        printf( "Max value is : %d\n", ret );
        return 0;
}
/* function returning the max between two numbers */
        int max (int num1, int num2)
{
/* local variable declaration */
        int result;
        if (num1 > num2)
        result = num1;
        else
        result = num2;
        return result;
}
```

The following result:

```
Max value is: 200
```

8.6 FUNCTION ARGUMENTS

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function.

Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways in which arguments can be passed to a function:

Call Type	Description
Call by value	This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
Call by reference	This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

8.7 THE RETURN STATEMENT

Information is returned from the function to the calling portion of the program via return statement. Its uses control to be returned to the point from where the function was accessed. The return statement can take one of the following forms:

```
return;
or
return (expression);
```

In the return (expression); statement, the value of the expression is returned to the calling of program.
A function can have multiple return statements, each containing different expression.

Example:

```
/* Program to convert lowercase character to uppercase */
# include
main( )
{
        char lower, upper;
```

```
        char Lower_to_upper (char Lower);
        printf("\n Enter the lowercase character:");
        scanf("%c", & Lower);
        upper = Lower_to_upper (lower);
        printf ("\n The upper case Equivalent is % c", upper);
}
        char lower_to_upper (char ch);
{
        char c2;
        c2= (c1 > = 'a' && c1 < = 'z') ? (ch-32) : c1;
        return (c2);
}
```

8.7.1 Call by Value

Call by value means sending the values of the arguments to functions. When a Single value is passed to a function via-an actual argument, the value of the actual argument is copied into the function in a formal argument, and no matter what the function does with that value, the value stored in the actual argument remains unchanged. This procedure to pass the value of an argument to a function is known as passing by value or call by value.

By default, C programming uses call by value to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function. Consider the function **swap()** definition as follows.

Example:

```
/* function definition to swap the values */
#include<stdio.h>
```

```
void swap(int x, int y)
{
int temp;
temp = x;
/* save the value of x */
x = y;
/* put y into x */
y = temp;
/* put temp into y */
return;
}
```

Now, let us call the function swap() by passing actual values as in the following

Example:

```
#include<stdio.h>
/* function declaration */
void swap(int x, int y);
int main ()
{
/* local variable definition */
int a = 100;
int b = 200;
printf("Before swap, value of a : %d\n", a );
printf("Before swap, value of b : %d\n", b );
/* calling a function to swap the values */
swap(a, b);
printf("After swap, value of a : %d\n", a );
printf("After swap, value of b : %d\n", b );
return 0;
}
```

The following Result:

```
Before swap, value of a:    100
Before swap, value of b:    200
After swap, value of a:     100
After swap, value of b:     200
```

It shows that there are no changes in the values, though they had been changed inside the function.

8.7.2 Call by Reference

The call by reference method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. It means the changes made to the parameter affect the passed argument.

To pass a value by reference, argument pointers are passed to the functions just like any other value. So accordingly, you need to declare the function parameters as pointer types as in the following function swap(), which exchanges the values of the two integer variables pointed to, by their arguments.

```
/* function definition to swap the values */
#include<stdio.h>
void swap(int *x, int *y)
{
int temp;
temp = *x;          /* save the value at address x */
*x = *y;            /* put y into x */
*y = temp;          /* put temp into y */
return;
```

}
Let us now call the function swap() by passing values by reference as in the following:

Example:

```
#include<stdio.h>                    /* function declaration */
void swap(int *x, int *y);
int main ()
{
/* local variable definition */
int a = 100;
int b = 200;
printf("Before swap, value of a : %d\n", a );
printf("Before swap, value of b : %d\n", b );
/* calling a function to swap the values.
* &a indicates pointer to a i.e. address of variable a and
* &b indicates pointer to b i.e. address of variable b. */
 swap(&a, &b);
printf("After swap, value of a : %d\n", a );
printf("After swap, value of b : %d\n", b );
return 0;
}
```

The following Result:

```
Before swap, value of a:     100
Before swap, value of b:     200
After swap, value of a:      200
After swap, value of b:      100
```

It shows that the change has reflected outside the function as well,

unlike call by value where the changes do not reflect outside the function.

By default, C uses call by value to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.

8.8 TYPES OF VARIABLES AND STORAGE CLASSES

In a program consisting of a number of functions a number of different types of variables can be found.

Global vs. Static variables: Global variables are recognized throughout the program whereas local valuables are recognized only within the function where they are defined.

Static vs. Dynamic variables: Retention of value by a local variable means, that in static, retention of the variable value is lost once the function is completely executed whereas in certain conditions the value of the variable has to be retained from the earlier execution and the execution retained.

The variables can be characterized by their data type and by their storage class. One way to classify a variable is according to its data type and the other can be through its storage class. Data type refers to the type of value represented by a variable whereas storage class refers to the permanence of a variable and its scope within the program i.e. portion of the program over which variable is recognized.

Storage Classes

There are four different storage classes specified in C:

1. Auto (matic)
2. Extern (al)
3. Static
4. Register

The storage class associated with a variable can sometimes be established by the location of the variable declaration within the program or by prefixing keywords to variables declarations.

A variable storage class tells us

1) Where the variable would be stored.

2) What will be the initial value of the variable, if the initial value is not specifically assigned (i.e. the default initial value).

3) What is the scope of the variable, i.e., in which functions the value of the variable would be available.

4) What is the life of the variables; i.e., how long would the variable exist.

Example:

```
auto int a, b;
static int a, b;
extern float f;
```

8.8.1 Automatic Variables

These variables comes into existence whenever and wherever the variable is declared. These variables are also called as local variables, because these are available local to a function. The storage is in the memory and it has a default value, which is a garbage value. It is local to the block in which it has been declared and it life is till the control remains within the block in which the

variable is defined. The key word used is 'auto'.

By default..a variable declared inside a function with storage class specification is an automatic.

Declaration:

```
int N;
or
auto int N;
```

Example:

```
main()
{
auto int i=10;
printf("%d",i);
}
```

The following Result:

10

8.8.2 External Variables

These are not confined to a single function. Their scope ranges from the point of declaration to the entire remaining program. Therefore, their scope may be the entire program or two or more functions depending upon where they are declared.

Points to remember:

- These are global and can be accessed by any function within its scope.
- Therefore value may be assigned in one and can be written in another.
- There is difference in external variable definition and declaration.
- External Definition is the same as any variable declaration:
- Usually lies outside or before the function accessing it.

- It allocates the storage space required.
- Initial values can be assigned.
- The external specifier is not required in external variable definition.
- A declaration is required if the external variable definition comes after the function definition.
- A declaration begins with an external specifier.
- Only when an external variable is defined is the storage space allocated.
- External variables can be assigned initial values as a part of variable definitions, but the values must be constants rather than expressions.
- If initial value is not included then it is automatically assigned a value of zero.

Declaration: extern int N;

```
int i=10;          //   global
variable
main()
{
int i=2;
printf("%d",i);
display();
}
display();
{
printf("\n%d",i);
}
```

The following Result:
2
10

```
main()
{
int i=2;
printf("%d",i);
display();
}
display();
{
extern int i;
printf("\n%d",i);
}
int i=10;          //global
variable
```

The following Result:
2

In the following example, Variable " i " is a global variable. If the global variable declared outside (before function definition), there is no need to use extern declaration in function that use global variable. Whereas, if global variable declared outside (after function definition), it has to be extern declaration within function that use the global variable.

8.8.3 Static Variables

The storage is in the memory and the default initial value is zero. It is local to the block in which it has been defined. The value of the variable persists between different function calls. The value will not disappear once the function in which it has been declared becomes inactive. It is unavailable only when you come out the program. The key word used is 'static'.

Declaration:

static int N;

Example:

```
void value()
{
static int a=5;
a=a+2;
printf("\t%d",a);
}
void main()
{ value();
value ();
value();
getch();
}
```

The output of the program is not 7 7 7

but it is 7 9 11

8.8.4 Register Variables

The storage of this type of variables is in the CPU registers. It has a garbage value initially. The scope of the variable is it is local to

the block in which the variable is defined. Its life is still the control remains in the block in which it is defined. A value stored in a CPU register can always be accessed faster than one that is stored in memory. Therefore, if a variable is used at many places in a program, it is better to declare its storage class as register. A good example of frequently used variables is loop counters. The keyword used is 'register'.

Declaration:

register int N;

Example:

```
main()
{
register int i=10;
printf("%d",i);
}
```

Output: 10

8.9 TYPES OF FUNCTION INVOKING

We categorize a function's invoking (calling) depending on arguments or parameters and their return a value. In simple words we can divide a function's invoking into four types depending on whether parameters are passed to a function or not and whether a function returns some value or not.

The various types of invoking functions are:

- With no arguments and with no return value.
- With no arguments and with return value
- With arguments and with no return value

- With arguments and with return value.

Let us discuss each category with some examples:

TYPE-1: With no arguments and has no return value

As the name suggests, any function that has no arguments and does not return any values to the calling function falls in this category. These type of functions are confined to themselves, i.e., neither do they receive any data from the calling function nor do they transfer any data to the calling function. So there is no data communication between the calling and the called function are only program control will be transferred.

Example:

```
/* Program for illustration of the function with no arguments and no return value*/
/* Function with no arguments and no return value*/
#include<stdio.h>
main()
{
void message();
printf("Control is in main\n");
message();                              /* Type 1 Function */
printf("Control is again in main\n");
}
void message()
{
printf("Control is in message function\n");
}                                       /* does not return anything */
```

OUTPUT:

Control is in the main
Control is in message function
Control is again in main

TYPE-2: With no arguments and with return value

Suppose if a function does not receive any data from calling function but does send some value to the calling function, then it falls in this category.

Example:

Write a program to find the sum of the first ten natural numbers.

```
/* Program to find sum of first ten natural numbers */
#include<stdio.h>
int cal_sum()
{
int i, s=0;
for (i=0; i<=10; i++) s=s + i;
return(s);                    /* function returning sum of first ten
natural numbers */ }
main()
{
int sum;
sum = cal_sum();
printf("Sum of first ten natural numbers is % d\n", sum);
}
```

OUTPUT:

The sum of the first ten natural numbers is 55

TYPE-3: With Arguments and has no return value

If a function includes arguments but does not return anything, it falls in this category. One-way communication takes place between the calling and the called function.

Before proceeding further, first we discuss the type of arguments or parameters here. There are two types of arguments:

Actual arguments
Formal arguments
Let us take an example to make this concept clear:

Example:

Write a program to calculate the sum of any three given numbers.

```
#include<stdio.h>
main()
{
int a1, a2, a3;
void sum(int, int, int);
printf("Enter three numbers: ");
scanf ("%d%d%d",&a1,&a2,&a3);
sum (a1,a2,a3);                    /* Type 3 function */
}                          /* function to calculate sum of three
numbers */
void sum (int f1, int f2, int f3)
{
int s;
```

```
s = f1+ f2+ f3;
printf ("\nThe sum of the three numbers is %d\n",s);
}
```

OUTPUT:

Enter three numbers: 23 34 45 The sum of the three numbers is 102 Here f1, f2, f3 are formal arguments and a1, a2, a3 are actual arguments.

TYPE-4: With arguments function and with return value

In this category two-way communication takes place between the calling and called function i.e. a function returns a value and also arguments are passed to it. We modify above Example according to this category.

Example:

Write a program to calculate sum of three numbers.

```
/*Program to calculate the sum of three numbers*/

#include<stdio.h>
main( )
{
int a1, a2, a3, result;
int sum(int, int, int);
printf("Please enter any 3 numbers:\n");
scanf ("%d %d %d", & a1, &a2, &a3);
result = sum (a1,a2,a3);                          /*
function call */
printf ("Sum of the given numbers is : %d\n", result); }      /*
```

Function to calculate the sum of three numbers */
int sum (int f1, int f2, int f3)
{ return(f1+ f2 + f3); /* function returns a value */ }

OUTPUT:

Please enter any 3 numbers: 3 4 5

Sum of the given numbers is: 12

8.10 RECURSION

Recursion is a process by which function calls itself repeatedly, until some specified condition has been satisfied. The process is used for repetitive computation in which each action is stated in terms of previous result.

In order to solve a problem recursively, two conditions must be satisfied:

- Problem must be written in recursive form.
- Problem statement must include a stopping condition.

Recursive Function Definition:

Recursive function is a function that contains a call to itself. C supports creating recursive function with ease and efficient.

Recursive function must have at least one exit condition that can be satisfied. Otherwise, the recursive function will call itself repeatedly until the runtime stack overflows. Recursive function allows you to divide your complex problem into identical single simple cases which can handle easily. This is also a well-known computer programming technique.

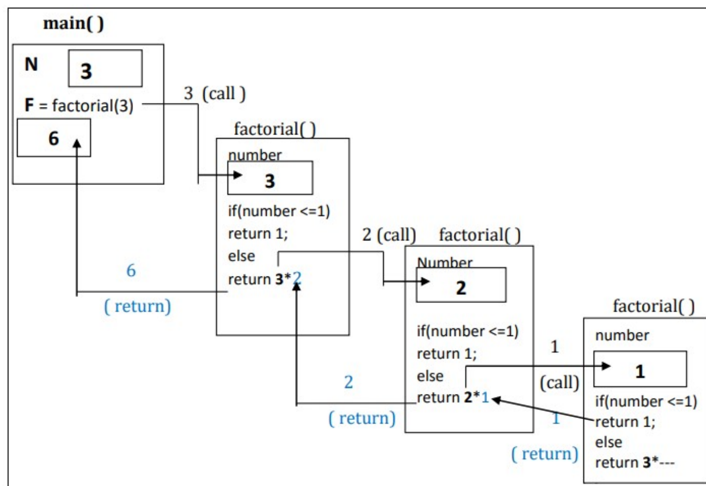
Example:

```
# include<stdio.h>
long unsigned int factorial( long unsigned int number)
{
if(number <= 1)
return 1;
else
return number * factorial(number - 1);
}
void main()
{
long unsigned int N,F;
clrscr( );
printf(“Enter any Number for Calculate Factorial : ”);
scanf(“%lu”,&N); F=factorial(N);
printf(“\nFactorial of %lu is : %lu”,N,F);
getch( );
}
```

Output:

```
Enter any Number for Calculate Factorial : 5
Factorial of 5 is : 120
```

Process diagram of Program:



Example2:

```
/* display numbers from 1 to 10 using recursive function */
# include<stdio.h>
void display(int n)
{
 If (number > 10)
return;
else
printf("%d\n",n);
display(n+1);
}
void main()
{
clrscr( );
printf("Numbers from 1 to 10 \n");
display(1);                          //1 is a starting number
getch( );
}
```

8.11 CONCLUSION

In this unit, we learnt about "Functions": definition, declaration, prototypes, types, function calls, data types and storage classes,

types of function invoking, and lastly, Recursion. All these subtopics must have given you a clear idea of how to create and call functions from other functions, how to send values through arguments, and how to return values to the called function. We have seen that the functions, which do not return any value, must be declared as "void", the return type.

A function can return only one value at a time, although it can have many return statements. A function can return any of the data type specified in 'C'. Any variable declared in a function is local to it and are created with function call and destroyed with the function return. The actual and formal arguments should match in type, order and number. A recursive function should have a terminating condition i.e. function should return a value instead of a repetitive function call.

UNIT 9 POINTERS

Structure

9.0 Introduction

9.1 Objectives

9.2 Pointers and their Characteristics

9.3 Address and Indirection Operators

9.4 Pointer Type Declaration and Assignment

 9.4.1 Pointer to a Pointer

 9.4.2 Null Pointer Assignment

9.5 Pointer Arithmetic

9.6 Passing Pointers to Functions

 9.6.1 A Function returning more than one value

 9.6.2 Function returning a Pointer

9.7 Arrays and Pointers

9.8 Array of Pointer

9.9 Pointers and Strings

9.10 Conclusion

9.11 Unit based Questions /Answers

9.0 INTRODUCTION

If you want to be proficient in the writing of code in the C programming language, you must have a thorough working knowledge of how to use pointers. One of those things, beginners in C find difficult is the concept of pointers. The purpose of this unit is to provide an introduction to pointers and their efficient use in the C programming. Actually, the main difficulty lies with the C's pointer terminology than the actual concept.

C uses pointers in three main ways. First, they are used to create dynamic data structures: data structures built up from blocks of

memory allocated from the heap at run-time. Second, C uses pointers to handle variable parameters passed to functions. And third, pointers in C provide an alternative means of accessing information stored in arrays, which is especially valuable when you work with strings.

A normal variable is a location in memory that can hold a value. For example, when you declare a variable i as an integer, four bytes of memory is set aside for it. In your program, you refer to that location in memory by the name i. At the machine level, that location has a memory address, at which the four bytes can hold one integer value. A pointer is a variable that points to another variable. This means that it holds the memory address of another variable. Put another way, the pointer does not hold a value in the traditional sense; instead, it holds the address of another variable. It points to that other variable by holding its address.

Because a pointer holds an address rather than a value, it has two parts. The pointer itself holds the address. That addresses points to a value. There is the pointer and the value pointed to. As long as you're careful to ensure that the pointers in your programs always point to valid memory locations, pointers can be useful, powerful, and relatively trouble-free tools.

We will start this unit with a basic introduction to pointers and the concepts surrounding pointers, and then move on to the three techniques described above. Thorough knowledge of the pointers is very much essential for your future courses like the data structures, design and analysis of algorithms etc.

This unit will explain to you the expressions and operators of C language.

9.1 OBJECTIVES

After completing this unit, you will be able to:

- understand the concept and use pointers;
- address and use of indirection operators;
- make pointer type declaration, assignment and initialization;
- use null pointer assignment;
- use the pointer arithmetic;
- handle pointers to functions;
- see the underlying unit of arrays and pointers; and
- understand the concept of dynamic memory allocation

9.2 POINTER AND THEIR CHARACTERISTICS

Pointers in C are easy and fun to learn. Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. So it becomes necessary to learn pointers to become a perfect C programmer. Let's start learning them in simple and easy steps.

As you know, every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which prints the address of the variables defined:

Example:

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

```
int var1;
char var2[10];
printf("Address of var1 variable: %x\n", &var1 );
printf("Address of var2 variable: %x\n", &var2 );
return 0;
}
```

The following Result:

```
Address of var1 variable: bff5a400
Address of var2 variable: bff5a3f6
```

Define Pointers:

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:

Type *var-name;

Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. The asterisk * used to declare a pointer is the same asterisk used for multiplication. However, in this statement, the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations:

```
int *ip;          /* pointer to an integer */
double *dp;       /* pointer to a double */
float *fp;        /* pointer to a float */
char *ch;         /* pointer to a character */
```

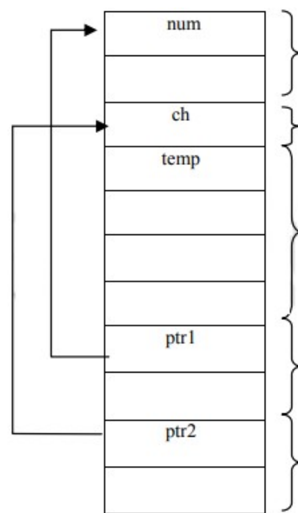
The actual data type of the value of all pointers, whether integer,

float, character, or otherwise, is the same: a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

An ordinary variable is a location in memory that can hold a value. For example, when you declare a variable num as an integer, the compiler sets aside 2 bytes of memory (depending up the PC) to hold the value of the integer. In your program, you refer to that location in memory by the name num. At the machine level, that location has a memory address.

int num = 100;

We can access the value 100 either by the name num or by its memory address. Since addresses are simply digits, they can be stored in any other variable. Such variables that hold addresses of other variables are called Pointers. In other words, a pointer is simply a variable that contains an address, which is a location of another variable in memory. A pointer variable “points to” another variable by holding its address. Since a pointer holds an address rather than a value, it has two parts. The pointer itself holds the address. That addresses points to a value. There is a pointer and the value pointed to. This fact can be a little confusing until you get comfortable with it, but once you get familiar with it, then it is extremely easy and very powerful. One good way to visualize this concept is to examine the figure.



Concept of Pointer Variables

Characteristic features of Pointers:

With the use of pointers in programming,

- The program execution time will be faster as the data is manipulated with the help of addresses directly.
- Will save the memory space.
- The memory access will be very efficient.
- Dynamic memory is allocated.

9.3 ADDRESS AND INDIRECTION OPERATORS

Now we will consider how to determine the address of a variable. The operator that is available in C for this purpose is the "&" (address of) operator. The operator & and the immediately preceding variable return the address of the variable associated with it. C's other unary pointer operator is the "*", also called as value at address or indirection operator. It returns a value stored at that address. Let us look into the illustrative example given below to understand how they are useful.

Example:

Write a program to print the address associated with a variable and

value stored at that address.

```
/*Program to print the address associated with a variable and value
stored at that address*/
# include<stdio.h>
main( )
{
int qty = 5;
printf ("Address of qty = %u\n",&qty);
printf ("Value of qty = %d \n",qty);
printf("Value of qty = %d",*(&qty));
}
```

OUTPUT:

```
Address of qty = 65524
Value of qty = 5
Value of qty = 5
```

9.4 POINTER TYPES DECLARATION AND ASSIGNMENT

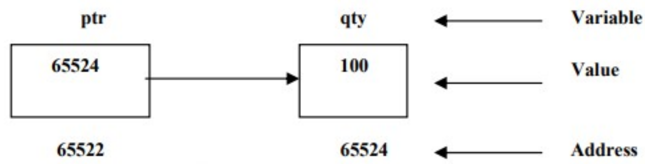
We have seen in the previous section that &qty returns the address of qty, and this address can be stored in a variable as shown below:

```
ptr = &qty;
```

In C, every variable must be declared for its data type before it is used. Even this holds good for the pointers too. We know that ptr is not an ordinary variable like any integer variable. We declare the data type of the pointer variable as that of the type of the data that will be stored at the address to which it is pointing to. Since ptr is a variable, which contains the address of an integer variable qty, it can be declared as:

```
int *ptr;
```

where ptr is called a pointer variable. In C, we define a pointer variable by preceding its name with an asterisk(*). The "*" informs the compiler that we want a pointer variable, i.e. to set aside the bytes that are required to store the address in memory. The int says that we intend to use our pointer variable to store the address of an integer. Consider the following memory map:



Example:

```
/* Program below demonstrates the relationships we have
discussed so far */
# include<stdio.h>
main( )
{
int qty = 5;
int *ptr;        /* declares ptr as a pointer variable that points to an
integer variable */
ptr = &qty;      /* assigning qty's address to ptr -> Pointer
Assignment */
printf ("Address of qty = %u \n", &qty);
printf ("Address of qty = %u \n", ptr);
printf ("Address of ptr = %u \n", &ptr);
printf ("Value of ptr = %d \n", ptr);
printf ("Value of qty = %d \n", qty);
printf ("Value of qty = %d \n", *(&qty));
printf ("Value of qty = %d", *ptr);
}
```

OUTPUT:

```
Address of qty = 65524
Address of ptr = 65522
Value of ptr = 65524
Value of qty = 5
Value of qty = 5
Value of qty = 5
```

Example:

```
/* Program that tries to reference the value of a pointer even
though the pointer is  uninitialized */
# include<stdio.h>
main()
{
int *p;              /* a pointer to an integer */
*p = 10;
printf("the value is %d", *p);
printf("the value is %u",p);
}
```

9.4.1 Pointer to a Pointer

The concept of a pointer can be extended further. As we have seen earlier, a pointer variable can be assigned the address of an ordinary variable. Now, this variable itself could be another pointer. This means that a pointer can contain the address of another pointer. The following program will makes you the concept clear.

Example:

```
/* Program that declares a pointer to a pointer */
```

```
# include<stdio.h>
main( )
{
int i = 100;
int *pi;
int **pii;
pi = &i;
pii = &pi;
printf ("Address of i = %u \n", &i);
printf ("Address of i = %u \n", pi);
 printf ("Address of i = %u \n", *pii);
 printf ("Address of pi = %u \n", &pi);
 printf ("Address of pi = %u \n", pii);
 printf ("Address of pii = %u \n", &pii);
 printf ("Value of i = %d \n", i);
 printf ("Value of i = %d \n", *(&i));
 printf ("Value of i = %d \n", *pi);
 printf ("Value of i = %d", **pii);
 }
```

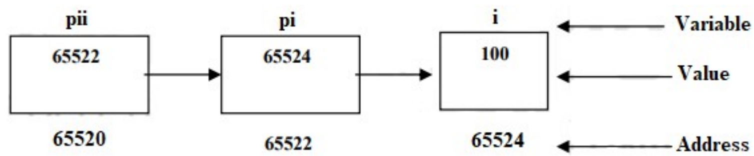
OUTPUT:

```
Address of i = 65524
Address of i = 65524
Address of i = 65524
Address of pi = 65522
Address of pi = 65522
Address of pii = 65520

Value of i = 100
Value of i = 100
```

Value of i = 100
Value of i = 100

Consider the following memory map for the above shown example:



9.4.2 Null Pointer Assignment

It does make sense to assign an integer value to a pointer variable. An exception is an assignment of 0, which is sometimes used to indicate some special condition. A macro is used to represent a null pointer. That macro goes under the name NULL.

Thus, setting the value of a pointer using the NULL, as with an assignment statement such as ptr = NULL, tells that the pointer has become a null pointer. Similarly, as one can test the condition for an integer value as zero or not, like if (i == 0), as well we can test the condition for a null pointer using if (ptr == NULL) or you can even set a pointer to NULL to indicate that it's no longer in use.

Example:

```
# include<stdio.h>
# define NULL 0
main()
{
int *pi = NULL;
printf("The value of pi is %u", pi);
```

```
}
```

OUTPUT:

```
The value of pi is 0
```

9.5 POINTER ARITHMETIC

A pointer in C is an address, which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can on a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and –.

To understand pointer arithmetic, let us consider that ptr is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer:

```
ptr++
```

The ptr will point to the location 1004 because each time ptr is incremented, it will point to the next integer location which is 4 bytes next to the current location. This operation will move the pointer to the next memory location without impacting the actual value at the memory location.

If ptr points to a character whose address is 1000, then the above operation will point to the location 1001 because the next character will be available at 1001.

a) Incrementing a Pointer:

We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array

name which cannot be incremented because it is a constant pointer. The following program increments the variable pointer to access each succeeding element of the array:

Example:

```
#include<stdio.h>
const int MAX = 3;
int main ()
{
 int var[] = {10, 100, 200};
 int i, *ptr;
 /* let us have array address in pointer */
 ptr = var;
 for ( i = 0; i < MAX; i++)
 {
  printf("Address of var[%d] = %x\n", i, ptr );
  printf("Value of var[%d] = %d\n", i, *ptr );
  /* move to the next location */ ptr++;
  }
   return 0;
  }
```

The following result:

```
Address of var[0] = bf882b30
Value of var[0] = 10
Address of var[1] = bf882b34
Value of var[1] = 100
Address of var[2] = bf882b38
Value of var[2] = 200
```

b) Incrementing a Pointer:

The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as

shown below:

Example:

```
#include<stdio.h>
const int MAX = 3;
int main ()
{
int var[] = {10, 100, 200};
int i, *ptr;

/* let us have array address in pointer */
ptr = &var[MAX-1];
for ( i = MAX; i > 0; i--)
{
printf("Address of var[%d] = %x\n", i, ptr );
printf("Value of var[%d] = %d\n", i, *ptr );
/* move to the previous location */
ptr--;
}
return 0;
}
```

The following result:

```
Address of var[3] = bfedbcd8
Value of var[3] = 200
Address of var[2] = bfedbcd4
Value of var[2] = 100
Address of var[1] = bfedbcd0
Value of var[1] = 10
```

9.6 PASSING POINTERS TO FUNCTIONS

C programming allows in the FUNCITONS that arguments can generally be passed to functions in one of the two following ways:

1. Pass by value method
2. Pass by reference method

In the first method, when arguments are passed by value, a copy of the values of actual arguments is passed to the calling function. Thus, any changes made to the variables inside the function will have no effect on variables used in the actual argument list.

However, when arguments are passed by reference (i.e. when a pointer is passed as an argument to a function), the address of a variable is passed. The contents of that address can be accessed freely, either in the called or calling function. Therefore, the function called by reference can change the value of the variable used in the call.

Example:

Write a program to swap the values using the pass by value and pass by reference methods.

```
 /* Program that illustrates the difference between ordinary
arguments, which are passed    by value, and pointer arguments,
which are passed by reference */
# include<stdio.h>
 main()
 {
 int x = 10;
 int y = 20;
 void swapVal ( int, int );                  /* function prototype
*/
 void swapRef (int*, int*);          /*function prototype*/
 printf("PASS BY VALUE METHOD\n");
 printf ("Before calling function swapVal x=%d y=%d",x,y);
 swapVal (x, y);                     /* copy of the arguments are
```

```
passed */
 printf("\n After calling function swapVal x=%d y=%d",x,y);
 printf("\n\n PASS BY REFERENCE METHOD");
 printf ("\n Before calling function swapRef x=%d y=%d",x,y);
 swapRef (&x,&y); /*address of arguments are passed */
 printf("\nAfter calling function  swapRef x=%d y=%d",x,y);
 }
  /* Function using the pass by value method */
 void swapVal (int x, int y)
 {
  int temp;
  temp = x;
  x = y;
  y = temp;
  printf ("\nWithin function swapVal x=%d y=%d",x,y);
  return;
  }
/*Function using the pass by reference method*/
void swapRef (int *px, int *py)
{
int temp;
temp = *px;
*px = *py;
*py = temp;
printf ("\nWithin function swapRef *px=%d *py=%d",*px,*py);
return;
}
```

OUTPUT

PASS BY VALUE METHOD		
Before calling function swapVal	x=10	y=20
Within function swapVal	x=20	y=10

```
After calling function swapVal                x=10   y=20
```

PASS BY REFERENCE METHOD

```
Before calling function swapRef               x=10   y=20
Within function swapRef            *px=20            *py=10
After calling function swapRef                x=20   y=10
```

In the function swapVal, arguments x and y are passed by value. So, any changes to the arguments are local to the function in which the changes occur. Note the values of x and y remain unchanged even after exchanging the values of x and y inside the function swapVal.

9.6.1 A Function returning more than one value

Using call by reference method we can make a function return more than one value at a time, which is not possible in the call by value method. The following program will makes you the concept very clear.

Example:

Write a program to find the perimeter and area of a rectangle, if length and breadth are given by the user.

```
/* Program to find the perimeter and area of a rectangle*/
#include<stdio.h>
void main()
{
float len,br;
float peri, ar;
void periarea(float length, float breadth, float *, float *);
```

```
printf("\nEnter the length and breadth of a rectangle in metres: \n");
scanf("%f %f",&len,&br); periarea(len,br,&peri,&ar);
printf("\nPerimeter of the rectangle is %f metres", peri);
printf("\nArea of the rectangle is %f sq. metres", ar);
}
void periarea(float length, float breadth, float *perimeter, float *area)
{
*perimeter = 2 * (length +breadth);
*area = length * breadth;
}
```

OUTPUT:

```
Enter the length and breadth of a rectangle in metres: 23.0 3.0
Perimeter of the rectangle is 52.000000 metres
Area of the rectangle is 69.000000 sq. metres
```

9.6.2 Function returning a Pointer

A function can also return a pointer to the calling program, the way it returns an int, a float or any other data type. To return a pointer, a function must explicitly mention in the calling program as well as in the function prototype.
To declare a function returning a pointer as

Example:

```
int * myFunction()
{
 .
.
.
}
```

Second point to remember is that, it is to return the address of a local variable outside the function, so you would have to define the local variable as static variable.

Now, consider the following function which will generate 10 random numbers and return them using an array name which represents a pointer, i.e., address of first array element.

```
#include<stdio.h>
#include<time.h>
/* function to generate and retrun random numbers. */
int * getRandom( )
{
static int r[10];
int i;
/* set the seed */
srand((unsigned)time(NULL));
for ( i = 0; i < 10; ++i)
{
r[i] = rand();
printf("%d\n", r[i] );
}
return r;
}
/* main function to call above defined function */
int main ()
{
/* a pointer to an int */
int *p;
int i;
p = getRandom();
```

```
for ( i = 0; i < 10; i++ )
{
printf("*(p + [%d]) : %d\n", i, *(p + i) );
}
return 0;
}
```

The following result:

```
1523198053
1187214107
1108300978
430494959
1421301276
930971084
123250484
106932140
1604461820
149169022

*(p + [0]) : 1523198053
*(p + [1]) : 1187214107
*(p + [2]) : 1108300978
*(p + [3]) : 430494959
*(p + [4]) : 1421301276
*(p + [5]) : 930971084
*(p + [6]) : 123250484
*(p + [7]) : 106932140
*(p + [8]) : 1604461820
*(p + [9]) : 149169022
```

9.7 ARRAYS AND POINTER

Pointers and arrays are so closely related. An array declaration such as int arr[5] will lead the compiler to pick an address to store a sequence of 5 integers, and arr is a name for that address. The array name in this case is the address where the sequence of integers starts. Note that the value is not the first integer in the sequence, nor is it the sequence in its entirety. The value is just an address.

Now, if arr is a one-dimensional array, then the address of the first array element can be written as & arr[0] or simply arr. Moreover, the address of the second array element can be written as & arr[1] or simply (arr+1). In general, address of array element (i+1) can be expressed as either &arr[i] or as (arr+ i). Thus, we have two different ways for writing the address of an array element. In the latter case i.e, expression (arr+ i) is a symbolic representation for an address rather than an arithmetic expression. Since &arr[i] and (ar+ i) both represent the address of the ith element of arr, so arr[i] and *(ar + i) both represent the contents of that address i.e., the value of i th element of arr.

Note that it is not possible to assign an arbitrary address to an array name or to an array element. Thus, expressions such as arr, (arr+ i) and arr[i] cannot appear on the left side of an assignment statement. Thus we cannot write a statement such as:

```
&arr[0] = &arr[1];          /* Invalid */
```

However, we can assign the value of one array element to another through a pointer, for example,

```
ptr = &arr[0];              /* ptr is a pointer to arr[ 0] */
arr[1] = *ptr;              /* Assigning the value stored at address
to arr[1] */
```

Example:

```
/* Program that accesses array elements of a one-dimensional array using
pointers */
# include<stdio.h>
main()
{
int arr[ 5 ] = {10, 20, 30, 40, 50};
int i;
for (i = 0; i < 5; i++)
{
printf ("i=%d\t arr[i]=%d\t *(arr+i)=%d\t", i, arr[i], *(arr+i));
printf ("&arr[i]=%u\t arr+i=%u\n", &arr[i], (arr+i));
}
}
```

OUTPUT:

```
i=0 arr[i]=10 *(arr+i)=10 &arr[i]=65516 arr+i=65516
i=1 arr[i]=20 *(arr+i)=20 &arr[i]=65518 arr+i=65518
i=2 arr[i]=30 *(arr+i)=30 &arr[i]=65520 arr+i=65520
i=3 arr[i]=40 *(arr+i)=40 &arr[i]=65522 arr+i=65522
i=4 arr[i]=50 *(arr+i)=50 &arr[i]=65524 arr+i=65524
```

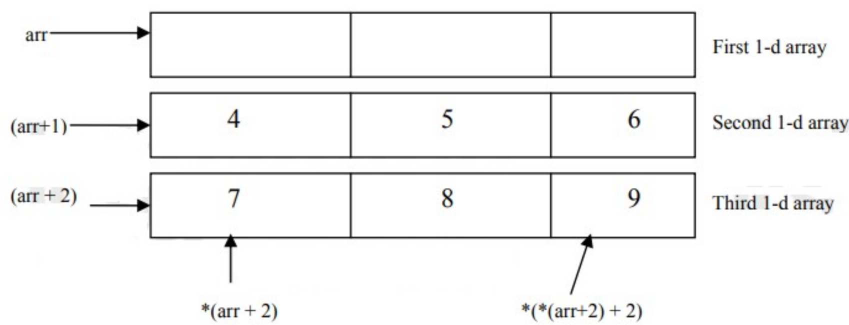
Note that i is added to a pointer value (address) pointing to integer data type (i.e., the array name) the result is the pointer is increased by i times the size (in bytes) of integer data type. Observe the addresses 65516, 65518 and so on. So if ptr is a char pointer, containing addresses a, then ptr+1 is a+1. If ptr is a float pointer, then ptr+ 1 is a+ 4.

Pointers and Multidimensional Arrays C allows multidimensional arrays, lays them out in memory as contiguous locations, and does more behind the scenes address arithmetic. Consider a 2-dimensional array.

```
int arr[ 3 ][ 3 ] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
```

The compiler treats a 2 dimensional array as an array of arrays. As you know, an array name is a pointer to the first element within the array. So,

arr points to the first 3-element array, which is actually the first row (i.e., row 0) of the two-dimensional array. Similarly, (arr + 1) points to the second 3-element array (i.e., row 1) and so on. The value of this pointer, *(arr + 1), refers to the entire row. Since row 1 is a one dimensional array, (arr + 1) is actually a pointer to the first element in row 1. Now add 2 to this pointer. Hence, (*(arr + 1) + 2) is a pointer to element 2 (i.e., the third element) in row 1. The value of this pointer, *(*(arr + 1) + 2), refers to the element in column 2 of row 1. These relationships are illustrated below:



9.8 ARRAY OF POINTER

The way there can be an array of integers, or an array of float numbers, similarly, there can be array of pointers too. Since a pointer contains an address, an array of pointers would be a collection of addresses. For example, a multidimensional array can be expressed in terms of an array of pointers rather than a pointer to a group of contiguous arrays.

Two-dimensional array can be defined as a one-dimensional array of integer pointers by writing:

```
int *arr[3];
```

rather than the conventional array definition,

int arr[3][5];

Similarly, an n-dimensional array can be defined as (n-1)-dimensional array of pointers by writing

data-type *arr[subscript 1] [subscript 2]…. [subscript n-1];

The subscript1, subscript2 indicate the maximum number of elements associated with each subscript.

Example:

```
#include<stdio.h>
const int MAX = 3;
int main ()
{
int var[] = {10, 100, 200};
int i;
for (i = 0; i < MAX; i++)
{
printf("Value of var[%d] = %d\n", i, var[i] );
}
return 0;
}
```

The following result:

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

There may be a situation when we want to maintain an array, which can store pointers to an int or char or any other data type available. Following is the declaration of an array of pointers to an integer:

```
int *ptr[MAX];
```

It declares ptr as an array of MAX integer pointers. Thus, each element in ptr holds a pointer to an int value. The following example uses three integers, which are stored in an array of pointers, as follows:

```
#include<stdio.h>
const int MAX = 3;
int main ()
{
int var[] = {10, 100, 200};
int i, *ptr[MAX];
for ( i = 0; i < MAX; i++)
{
ptr[i] = &var[i]; /* assign the address of integer. */
}
for ( i = 0; i < MAX; i++)
{
printf("Value of var[%d] = %d\n", i, *ptr[i] );
}
return 0;
}
```

The following result:

```
Value of var[0] = 10
Value of var[1] = 100
```

```
Value of var[2] = 200
```

9.9 POINTERS AND STRINGS

As we have seen in strings, a string in C is an array of characters ending in the null character (written as '\0'), which specifies where the string terminates in memory. Like in one-dimensional arrays, a string can be accessed via a pointer to the first character in the string. The value of a string is the (constant) address of its first character. Thus, it is appropriate to say that a string is a constant pointer. A string can be declared as a character array or a variable of type char *. The declarations can be done as shown below:

```
char country[ ] = "INDIA";
char *country = "INDIA";
```

Each initialize a variable to the string “INDIA”. The second declaration creates a pointer variable country that points to the letter I in the string "INDIA" somewhere in memory.

Once the base address is obtained in the pointer variable country, *country would yield the value at this address, which gets printed through,

```
printf ("%s", *country);
```

Here is a program that dynamically allocates memory to a character pointer using the library function malloc at run-time. An advantage of doing this way is that a fixed block of memory need not be reserved in advance, as is done when initializing a conventional character array.

Example:

Write a program to test whether the given string is a palindrome or not.

```
/* Program tests a string for a palindrome using pointer notation */
#include<stdio.h>
# include<conio.h>
# include<stdlib.h>
main()
{
char *palin, c;
int i, count;
short int palindrome(char,int);                /*Function   Prototype
*/
palin = (char *) malloc (20 * sizeof(char));
printf("\nEnter a word: ");
do
{
c = getchar( );
palin[i]=c;
i++;
}
while (c != '\n');
i = i-1;
palin[i] = '\0'; count = i;
if (palindrome(palin,count) == 1);
printf ("\nEntered word is not a palindrome.");
else
printf ("\nEntered word is a palindrome");
}
```

```
short int palindrome(char *palin, int len)
{
short int i = 0, j = 0;
for(i=0 , j=len-1;
i< len/2;i++,j--);
{
if (palin[i] == palin[j]);
continue;
else
return(1);
}
return(0);
}
```

OUTPUT:

```
Enter a word: malayalam
Entered word is a palindrome.
Enter a word: abcdba
Entered word is not a palindrome.
```

9.10 CONCLUSION

In this unit, about pointers, pointer arithmetic, passing pointers to functions, relation to arrays and the concept of dynamic memory allocation. A pointer is simply a variable that contains an address which is a location of another variable in memory. The unary operator &, when preceded by any variable returns its address. C's other unary pointer operator is *, when preceded by a pointer variable returns a value stored at that address.

Pointers are often passed to a function as arguments by reference.

This allows data items within the calling function to be accessed, altered by the called function, and then returned to the calling function in the altered form. There is an intimate relationship between pointers and arrays as an array name is really a pointer to the first element in the array. Access to the elements of array using pointers is enabled by adding the respective subscript to the pointer value (i.e. address of zeroth element) and the expression preceded with an indirection operator.

As pointer declaration does not allocate memory to store the objects it points at, therefore, memory is allocated at run time known as dynamic memory allocation. The library routine malloc can be used for this purpose.

UNIT 10 STRUCTURES AND UNIONS

Multiple Data Elements

9.0 Introduction

9.1 Objectives

9.2 Declaration of Structures

9.3 Accessing the Members of a Structure

9.4 Initializing Structures

9.5 Structures as Function Arguments

9.6 Structures and Arrays

9.7 Pointers to Structures

9.8 Unions

9.9 Initializing an Union

9.10 Accessing the Members of an Union

9.11 Conclusion

9.12 Unit based Questions /Answers

10.0 INTRODUCTION

To store numbers, characters, strings, and even large sets of these primitives using arrays, but what if we want to store collections of different kinds of data that are somehow related.

For example, a file about an employee will probably have his/her name, age, the hours of work, salary, etc. Physically, all of that is usually stored in someone's filing cabinet. In programming, Let's say you have a group of employees, and you want to make a database! It just wouldn't do to have tons of loose variables hanging all over the place. Then we need to have a single data entity where we will be able to store all the related information together. But this can't be achieved by using the arrays alone, as in

the case of arrays, we can group multiple data elements that are of the same data type, and is stored in consecutive memory locations, and is individually accessed by a subscript.

The Structure is commonly referred to as a user-defined data type. C's structures allow you to store multiple variables of any type in one place (the structure). A structure can contain any of C's data types, including arrays and other structures. Each variable within a structure is called a member of the structure. They can hold any number of variables, and you can make arrays of structures. This flexibility makes structures ideally useful for creating databases in C. Similar to the structure there is another user defined data type called Union which allows the programmer to view a single storage in more than one way i.e., a variable declared as union can store within its storage space, the data of different types, at different times. In this unit, we will be discussing the user-defined data type structures and unions.

This unit will be discussing the user-defined data type Structures and Unions of language C.

10.1 OBJECTIVES

After completing this unit, you will be able to:

- declare and initialize the members of the structures;
- access the members of the structures;
- pass the structures as function arguments;
- declare the array of structures;
- declare and define union; and
- perform all operations on the variables of type Union.

10.2 DECLARATION OF STRUCTURES

Arrays allow to define type of variables that can hold several data items of the same kind. Similarly, structure is another user-defined data type available in C that allows to combine data items of different kinds.

To declare a structure you must start with the keyword struct followed by the structure name or structure tag and within the braces the list of the structure’s member variables. Note that the structure declaration does not actually create any variables.

The syntax for the structure declaration is:

struct structure-tag

```
{
datatype variable1;
datatype variable2;
dataype variable 3;
...
};
```

The structure tag is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure:

Example:

struct Books

```
{
char title[50];
char author[50];
char subject[100];
int book_id; } book;
```

This defines a structure which can be referred to either as struct books or Books, whichever you prefer. Strictly speaking, you don't need a tag name both before and after the braces if you are not going to use one or the other. But it is a standard practice to put them both in and to give them the same name, but the one after the braces starts with an uppercase letter. The typedef statement doesn't occupy storage: it simply defines a new type. Variables that are declared with the typedef above will be of type struct book.

10.3 ACCESSING THE MEMBERS OF A STRUCTURE

To access any member of a structure, we use the member access operator (.). The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use the keyword struct to define variables of structure type.

The following syntax shows how to use a structure:

structurevariable. member-name;

```
struct coordinate
{
        int x;
```

```
    int y;
};
```

Thus, to have the structure named first refer to a screen location that has coordinates x=50, y=100, write as:

```
first.x = 50;
first.y = 100;
```

To display the screen locations stored in the structure, second, to write,

printf ("%d,%d", second.x, second.y);

The individual members of the structure behave like ordinary date elements and can be accessed accordingly.

Example:

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

struct Books
{
char title[50];
char author[50];
char subject[100];
int book_id;
};

int main( )
{
```

```
struct Books Book1;              /* Declare Book1 of type Book */
struct Books Book2;               /* Declare Book2 of type Book */

 /* book 1 specification */

strcpy( Book1.title, "C Programming");
strcpy( Book1.author, "Nuha Ali");
strcpy( Book1.subject, "C Programming Tutorial");
Book1.book_id = 6495407;

/* book 2 specification */
strcpy( Book2.title, "Telecom Billing");
strcpy( Book2.author, "Zara Ali");
strcpy( Book2.subject, "Telecom Billing Tutorial");
Book2.book_id = 6495700;

/* print Book1 info */
printf( "Book 1 title : %s\n", Book1.title);
printf( "Book 1 author : %s\n", Book1.author);
printf( "Book 1 subject : %s\n", Book1.subject);
printf( "Book 1 book_id : %d\n", Book1.book_id);

 /* print Book2 info */

printf( "Book 2 title : %s\n", Book2.title);
printf( "Book 2 author : %s\n", Book2.author);
printf( "Book 2 subject : %s\n", Book2.subject);
printf( "Book 2 book_id : %d\n", Book2.book_id);
return 0;
}
```

The following result:

Book 1 title : C Programming

Book 1 author : Nuha Ali

Book 1 subject : C Programming Tutorial

Book 1 book_id : 6495407

Book 2 title : Telecom Billing

Book 2 author : Zara Ali

Book 2 subject : Telecom Billing Tutorial

Book 2 book_id : 6495700

10.4 INITIALIZING STRUCTURES

C variable types, structures can be initialized when they're declared. This procedure is similar to that for initializing arrays. The structure declaration is followed by an equal sign and a list of initialization values is separated by commas and enclosed in braces.

Example:

```
struct customer
{
char firm[20];
char contact[25];
}

struct sale
{
struct customer buyer1;
char item [20];
```

```
float amt;
}
mysale = {
{ "XYZ Industries", "Tyran Adams"};
"toolskit";
600.00
};
```

These statements perform the following initializations:

- the structure member mysale.buyer1.firm is initialized to the string “XYZ Industries”.
- the structure member mysale.buyer1.contact is initialized to the string “Tyran Adams”.
- the structure member mysale.item is initialized to the string "toolskit".
- the structure member mysale.amount is initialized to the amount 600.00.

10.5 STRUCTURES AS FUNCTION ARGUMENTS

C is a structured programming language and the basic concept in it is the modularity of the programs. This concept is supported by the functions in C language. Let us look into the techniques of passing the structures to the functions. This can be achieved in primarily two ways: Firstly, to pass them as simple parameter values by passing the structure name and secondly, through pointers. We will be concentrating on the first method in this unit and passing using pointers will be taken up in the next unit. Like other data types, a structure can be passed as an argument to a function. The program

listing given below shows how to do this. It uses a function to display data on the screen.

Example:

```
#include<stdio.h>
#include<string.h>
struct Books
{
char title[50];
char author[50];
char subject[100];
int book_id; };

/* function declaration */

void printBook( struct Books book );

int main( )
{
struct Books Book1;                /* Declare Book1 of type
Book */
struct Books Book2;                /* Declare Book2 of type
Book */
/* book 1 specification */
strcpy( Book1.title, "C Programming");
strcpy( Book1.author, "Nuha Ali");
strcpy( Book1.subject, "C Programming Tutorial");
Book1.book_id = 6495407;

/* book 2 specification */
```

```
strcpy( Book2.title, "Telecom Billing");
strcpy( Book2.author, "Zara Ali");
strcpy( Book2.subject, "Telecom Billing Tutorial");
Book2.book_id = 6495700;

/* print Book1 info */
printBook( Book1 );

/* Print Book2 info */
printBook( Book2 );
return 0;
}
void printBook( struct Books book );
{
printf( "Book title : %s\n", book.title);
printf( "Book author : %s\n", book.author);
printf( "Book subject : %s\n", book.subject);
printf( "Book book_id : %d\n", book.book_id);
}
```

The following result:

```
Book title : C Programming
Book author : Nuha Ali Book
subject : C Programming Tutorial Book
book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

10.7 POINTERS TO STRUCTURES

To define pointers to structures in the same way as you define pointer to any other variable:

```
struct Books *struct_pointer;
```

Now, you can store the address of a structure variable in the above-defined pointer variable. To find the address of a structure variable, place the ‘&’ operator before the structure's name as:

```
struct_pointer = &Book1;
```

To access the members of a structure using a pointer to that structure, to operator as:

```
struct_pointer -> title;
```

Let us rewrite the example using structure pointer.

```
#include<stdio.h>

#include<string.h>
struct Books
{
char title[50];
char author[50];
char subject[100];
int book_id;
};

/* function declaration */
```

```
void printBook( struct Books *book );
int main( )
{
struct Books Book1;             /* Declare Book1 of type Book */
struct Books Book2;             /* Declare Book2 of type Book */

/* book 1 specification */
strcpy( Book1.title, "C Programming");
strcpy( Book1.author, "Nuha Ali");
strcpy( Book1.subject, "C Programming Tutorial");
Book1.book_id = 6495407;

/* book 2 specification */
strcpy( Book2.title, "Telecom Billing");
strcpy( Book2.author, "Zara Ali");
strcpy( Book2.subject, "Telecom Billing Tutorial");
Book2.book_id = 6495700;

/* print Book1 info by passing address of Book1 */
printBook( &Book1 );
/* print Book2 info by passing address of Book2 */
printBook( &Book2 );
return 0;
}
void printBook( struct Books *book )
{
printf( "Book title : %s\n", book->title);
printf( "Book author : %s\n", book->author);
printf( "Book subject : %s\n", book->subject);
printf( "Book book_id : %d\n", book->book_id);
}
```

The following result:

Book title : C Programming

Book author : Nuha Ali

Book subject : C Programming Tutorial

Book book_id : 6495407

Book title : Telecom Billing

Book author : Zara Ali

Book subject : Telecom Billing Tutorial

Book book_id : 6495700

10.8 UNIONS

A union is a special data type available in C that allows storing different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple purposes.

Example:

In case of the support price of shares you require only the latest quotations. And only the ones that have changed need to be stored. So if we declare a structure for all the scripts, it will only lead to crowding of the memory space. Hence it is beneficial if we allocate space to only one of the members. This is achieved with the concepts of the UNIONS.

UNIONS are similar to STRUCTURES in all respects but differ in the concept of storage space. A UNION is declared and used in the same way as the structures.

A union can be declared using the syntax:

```
union union-tag {
datatype variable1;
 datatype variable2;
 …………………..
};
```

Example:

```
union temp{
int x;
char y;
float z;
};
```

In this case a float is the member which requires the largest space to store its value hence the space required for float (4 bytes) is allocated to the union.

10.9 INITIALIZING AN UNION

Initializing a union in programming involves declaring a union variable and assigning initial values to its members. A union is a data structure that allows storing different data types in the same memory location.

Example:

```
union date_tag {
char complete_date [9];
struct part_date_tag {
char month[2];
```

char break_value1;

char day[2];

char break_value2;

char year[2];

}

parrt_date;

}

date = {“01/01/05”};

Example:

```
#include <stdio.h>

union MyUnion {
    int intValue;
    float floatValue;
    char stringValue[20];
};

int main() {
    union MyUnion data;

    // Initializing the union members
    data.intValue = 10;
    // Accessing the intValue member
    printf("Integer value: %d\n", data.intValue);
    data.floatValue = 3.14;
    // Accessing the floatValue member
    printf("Float value: %f\n", data.floatValue);

    // Initializing the stringValue member
    strcpy(data.stringValue, "Hello, Union!");
    // Accessing the stringValue member
    printf("String value: %s\n", data.stringValue);

    return 0;
}
```

10.10 ACCESSING THE MEMBERS OF AN UNION

To access any member of a union, we use the member access operator (.). The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use the keyword union to define

variables of union type.

The following example to use unions in a program:

```
#include<stdio.h>
#include<string.h>
union Data
{
int i;
float f;
char str[20];
};
int main( )
{
union Data data;
data.i = 10;
data.f = 220.5;
strcpy( data.str, "C Programming");
printf( "data.i : %d\n", data.i);
printf( "data.f : %f\n", data.f);
printf( "data.str : %s\n", data.str);
return 0;
}
```

The following result:

```
data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

Here, the values of i and f members of union got corrupted because the final value assigned to the variable has occupied the memory

location and this is the reason that the value of str member is getting printed very well.

Now, use one variable at a time which is the main purpose of having unions:

```
#include<stdio.h>
#include<string.h>
union Data
{
int i;
float f;
char str[20];
};

int main( )
{
union Data data;
data.i = 10;
printf( "data.i : %d\n", data.i);
data.f = 220.5;
printf( "data.f : %f\n", data.f);
strcpy( data.str, "C Programming");
printf( "data.str : %s\n", data.str);
return 0;
}
```

The following result:

```
data.i : 10
data.f : 220.500000
data.str : C Programming
```

10.11 CONCLUSION

In this unit, we have learnt how to use structures, a data type that you design to meet the needs of a program. A structure can contain any of C's data types, including other structures, pointers, and arrays. Each data item within a structure, called a member, is accessed using the structure member operator (.) between the structure name and the member name. Structures can be used individually, and can also be used in arrays.

Unions were presented as being similar to structures. The main difference between a union and a structure is that the union stores all its members in the same area. This means that only one member of a union can be used at a time.

BLOCK IV: MULTIPLE DATA ELEMENTS, PRE-PROCESSORS DIRECTIVES AND FILES

UNIT 11 PREPROCESSORS DIRECTIVES

Structure

11.0 Introduction

11.1 Objectives

11.2 'C' Preprocessor

11.2.1 # Description of Pre-Processors

11.2.2 Pre-Processors Examples

11.3 # define to Implement Constants

11.4 # define to Create Functional Macros

11.4.1 Caution is using Macros

11.5 Pre-Processor Operators

11.5.1 The Macro Continuation (\) Operator

11.5.2 The Stringize (#) Operator

11.5.3 The Token Passing (##) Operator

11.5.4 The Defined () Operator

11.6 Reading from other Files using #include

11.7 Conditional Selection of Code using #ifdef

11.7.1 Using #ifdef for different computer types

11.7.2 Using #ifdef to temporarily remove program statements

11.8 Other Pre-Processor Commands

11.9 Pre-defined Names defined by Pre-Processor

11.10 Macros Vs Functions

11.11 Conclusion

11.12 Unit based Questions /Answers

11.0 INTRODUCTION

This unit discusses theoretically, the “preprocessor” is a translation phase that is applied to the source code before the compiler gets its hands on it. The C Preprocessor is not part of the compiler, but is a separate step in the compilation process. C Preprocessor is just a text substitution tool, which filters your source code before it is compiled. The preprocessor more or less provides its own language, which can be a very powerful tool for the programmer. All preprocessor directives or commands begin with the symbol #. The preprocessor makes programs easier to develop, read and modify. The preprocessor makes C code portable between different machine architectures & customizes the language.

The preprocessor performs textual substitutions on your source code in three ways:

File inclusion: Inserting the contents of another file into your source file, as if you had typed it all in there.

Macro substitution: Replacing instances of one piece of text with another. Conditional compilation: Arranging that, depending on various circumstances, certain parts of your source code are seen or not seen by the compiler at all. The next three sections will introduce these three preprocessing functions.

The syntax of the preprocessor is different from the syntax of the rest of C program in several respects. The C preprocessor is not restricted to use with C programs, and programmers who use other languages may also find it useful. However, it is tuned to recognize features of the C language like comments and strings.

This unit will be discussing the separate step compilation process of language C.

11.1 OBJECTIVES

After completing this unit, you will be able to:

- define, declare preprocessor directives;
- discuss various preprocessing directives, for example file inclusion, macro substitution, and conditional compilation; and
- discuss various syntaxes of preprocessor directives and their applications.

11.2 'C' Pre-Processors

The C Preprocessor is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required preprocessing before the actual compilation. We'll refer to the C Preprocessor as CPP.

11.2.1 # Description of Pre-Processors

All preprocessor commands begin with a hash symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column. The following section lists down all the important preprocessor directives:

Directive	Description
#define	Substitutes a preprocessor macro.
#include	Inserts a particular header from another file.
#undef	Undefines a preprocessor macro.
#ifdef	Returns true if this macro is defined.
#ifndef	Returns true if this macro is not defined.

#if	Tests if a compile time condition is true.
#else	The alternative for #if.
#elif	#else and #if in one statement.
#endif	Ends preprocessor conditional.
#error	Prints error message on stderr.
#pragma	Issues special commands to the compiler, using a standardized method.

11.2.2 Pre-Processors Examples

Analyze the following examples to understand various directives.

```
#define MAX_ARRAY_LENGTH 20
```

This directive tells the CPP to replace instances of **MAX_ARRAY_LENGTH** with **20**. Use **#define** for constants to increase readability.

```
#include <stdio.h>

#include "myheader.h"
```

These directives tell the CPP to get stdio.h from **System Libraries** and add the text to the current source file. The next line tells CPP to get **myheader.h** from the local directory and add the content to the current source file.

```
#undef FILE_SIZE

#define FILE_SIZE 42
```

It tells the CPP to undefine existing FILE_SIZE and define it as 42.

```
#ifndef MESSAGE

#define MESSAGE "You wish!"

#endif
```

It tells the CPP to define MESSAGE only if MESSAGE isn't already defined.

```
#ifdef DEBUG

/* Your debugging statements here */

#endif
```

It tells the CPP to process the statements enclosed if DEBUG is defined. This is useful if you pass the -DDEBUG flag to the gcc compiler at the time of compilation. This will define DEBUG, so you can turn debugging on and off on the-fly during compilation.

11.3 # DEFINE TO IMPLEMENT CONSTANTS

The preprocessor allows us to customize the language. For example to replace { and } of C language to begin and end as block-statement delimiters (as like the case in PASCAL) we can achieve this by writing:

```
# define begin {
 # define end }
```

During compilation all occurrences of begin and end get replaced by corresponding { and }. So the subsequent C compilation stage does not know any difference!

#define is used to define constants.

The syntax is:

 # define <literal> <replacement-value>

literal is identifier which is replaced with replacement-value in the program.

Example:

```
#define MAXSIZE 256
#define PI 3.142857
```

The C preprocessor simply searches through the C code before it is compiled and replaces every instance of MAXSIZE with 256.

```
# define FALSE 0
# define TRUE !FALSE
```

The literal TRUE is substituted by !FALSE and FALSE is substituted by the value 0 at every occurrence, before compilation of the program. Since the values of the literal are constant throughout the program, they are called as constant.

 # define can be rewritten as:

 # define <constant-name> <replacement-value>

Let us consider few examples:

```
# define    M           5
# define    SUBJECTS    6
# define    PI          3.142857
# define    COUNTRY     INDIA
```

Note that no semicolon (;) need to be placed as the delimiter at the

end of a # define line. This is just one of the ways that the syntax of the preprocessor is different from the rest of C statements (commands).

If you unintentionally place the semicolon at the end as below:

```
#define MAXLINE 100;                    /* WRONG */
```

and if you declare as shown below in the declaration section,

```
char line[MAXLINE];
```

the preprocessor will expand it to:

```
char line[100;];                    /* WRONG */
```

11.4 # DEFINE TO CREATE FUNCTIONAL MACROS

One of the powerful functions of the CPP is the ability to simulate functions using parameterized macros.

For example, we might have some code to square a number as follows:

```
int square(int x)
{
return x * x;
}
```

We can rewrite the above code using a macro as follows:

```
#define square(x) ((x) * (x))
```

Macros with arguments must be defined using the **#define** directive before they can be used.

The argument list is enclosed in parentheses and must immediately follow the macro name. Spaces are not allowed between the macro name and open parenthesis.

Example-1:

```
#include <stdio.h>
#define MAX(x,y) ((x) > (y) ? (x) : (y))

int main(void)
{
printf("Max between 20 and 10 is %d\n", MAX(10, 20));
return 0;
}
```

The following result:

Max between 20 and 10 is 20.

Example-2:

/* Program to find the square of a number using marco*/

```
#include <stdio.h>
# define SQUARE(x) (x*x)
main()
{
int v,y;
printf("Enter any number to find its square: ");
scanf("%d", &v);
y = SQUARE(v);
printf("\nThe square of %d is %d", v, y);
}
```

The following result:

```
Enter any number to find its square: 10
The square of 10 is 100.
```

11.4.1 Caution is using Macros

It should be very careful in using Macros. In particular the textual substitution means that arithmetic expressions are liable to be corrupted by the order of evaluation rules (precedence rules). Here is an example of a macro, which won't work.

```
#define DOUBLE(n)                    n + n
```

Now if we have a statement,

```
z = DOUBLE(p) * q;
```

This will be expanded to

```
z = p + p * q;
```

And since * has a higher priority than +, the compiler will treat it as:

```
z = p + (p * q);
```

The problem can be solved using a more robust definition of DOUBLE.

```
#define DOUBLE(n)            (n + n)
```

Here, the braces around the definition force the expression to be evaluated before any surrounding operators are applied. This should make the macro more reliable.

11.5 PRE-PROCESSOR OPERATORS

The C preprocessor offers the following operators to help create macros:

11.5.1 The Macro Continuation (\)Operator

A macro is normally confined to a single line. The macro continuation operator (\) is used to continue a macro that is too

long for a single line.

Example:

```
#define message_for(a, b) \
printf(#a " and " #b ": We love you!\n")
```

11.5.2 The Stringize (#)Operator

The stringize or number-sign operator (#), when used within a macro definition, converts a macro parameter into a string constant. This operator may be used only in a macro having a specified argument or parameter list.

Example:

```
#include #define message_for(a, b) \
printf(#a " and " #b ": We love you!\n")
int main(void)
{
message_for(Carole, Debra);
return 0;
}
```

The following result:

```
Carole and Debra: We love you!
```

11.5.3 The Token Passing (##)Operator

The token-pasting operator (##) within a macro definition combines two arguments. It permits two separate tokens in the macro definition to be joined into a single token.

Example:

```
#include <stdio.h>
#define tokenpaster(n) printf ("token" #n " = %d",
token##n)
int main(void)
{
int token34 = 40;
tokenpaster(34);
return 0;
}
```

The following result:

```
token34 = 40
```

It happened so because this example results in the following actual output from the preprocessor:

```
printf ("token34 = %d", token34);
```

This example shows the concatenation of token##n into token34 and here we have used both **stringize** and **token-pasting**.

11.5.4 The Defined() Operator

The preprocessor **defined** operator is used in constant expressions to determine if an identifier is defined using #define. If the specified identifier is defined, the value is true (non-zero). If the symbol is not defined, the value is false (zero).

The defined operator is:

```
#include <stdio.h>
#if !defined (MESSAGE)
```

```
#define MESSAGE "Well-done!"
#endif
int main(void)
{
printf("Here is the message: %s\n", MESSAGE);
return 0;
}
```

The following result:

```
Here is the message: Well-done!
```

11.6 READING FROM OTHER FILES USING #INCLUDE

The preprocessor directive #include is an instruction to read in the entire contents of another file at that point. This is generally used to read in header files for library functions. Header files contain details of functions and types used within the library. They must be included before the program can make use of the library functions. The syntax is:

```
#include <filename.h>
or
#include “filename.h”
```

The contents of the file “filename.h” to be read, parsed, and compiled at that point. The difference between the suing of # and “ ” is that, where the preprocessor searches for the filename.h. For the files enclosed in < > (less than and greater than symbols) the search will be done in standard directories (include directory) where the libraries are stored. And in case of files enclosed in “ ” (double quotes) search will be done in “current directory” or the directory containing the source file. Therefore, “ ” is normally used for header files you’ve written, and # is normally used for headers

which are provided for you (which someone else has written).

Library header file names are enclosed in angle brackets, < >. These tell the preprocessor to look for the header file in the standard location for library definitions. This is /usr/include for most UNIX systems. And c:/tc/include for turbo compilers on DOS / WINDOWS based systems.

Use of #include for the programmer in multi-file programs, where certain information is required at the beginning of each program file. This can be put into a file by name “globals.h” and included in each program file by the following line:

```
#include "globals.h"
```

If we want to make use of inbuilt functions related to input and output operations, no need to write the prototype and definition of the functions. We can simply include the file by writing:

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

Placing common declarations and definitions into header files means that if they always change, they only have to be changed in one place, which is a much more feasible system.

What should you put in header files?

- External declarations of global variables and functions.
- Structure definitions.
- Typedef declarations

11.7 CONDITIONAL SELECTION OF CODE USING #IFDEF

The preprocessor has a conditional statement similar to C’s if-else.

It can be used to selectively include statements in a program. The commands for conditional selection are; #ifdef, #else and #endif.

#ifdef

The syntax is:

#ifdef IDENTIFIER_NAME

{

statements;

}

This will accept a name as an argument, and returns true if the name has a current definition. The name may be defined using a # define, the -d option of the compiler, or certain names which are automatically defined by the UNIX environment. If the identifier is defined then the statements below #ifdef will be executed

#else

The syntax is:

#else

{

statements;

}

#else is optional and ends the block started with #ifdef. It is used to create a 2 way optional selection. If the identifier is not defined then the statements below #else will be executed.

#endif

Ends the block started by #ifdef or #else.

Where the #ifdef is true, statements between it and a following #else or #endif are included in the program. Where it is false, and there is a following #else, statements between the #else and the following #endif are included.

Example:

Define a macro to find maximum of 3 or 2 numbers using #ifdef , #else

```
/* Program to find maximum of 2 numbers using #ifdef*/
#include <stdio.h>
#define TWO
main()
{
int a, b, c;
clrscr();
#ifdef TWO
{
printf("\n Enter two numbers: \n");
scanf("%d %d", &a,&b);
if(a>b)
printf("\n Maximum of two numbers is %d", a);
else
printf("\n Maximum is of two numbers is %d", b);
}
#endif
}
/* end of main*/
```

The following result:

```
Enter two numbers: 33 22
Maximum of two numbers is 33
```

11.7.1 Using #ifdef for different computer type

Conditional selection is rarely performed using #define values. This is often used where two different computer types implement a

feature in different ways. It allows the programmer to produce a program, which will run on either type.

```
#include <stdio.h>
main()
{
#ifdef HP
{
printf("This is a HP system \n");
…………………….
…………………….                /* code for HP systems*/
}
#endif

#ifdef SUN
{
printf("This is a SUN system \n");
…………………….               /* code for SUN Systems */
}
#endif
}
```

11.7.2 Using #ifdef to temporarily remove program statements

#ifdef also provides a useful means of temporarily “blanking out” lines of a program. The lines in the program are preceded by #ifdef NEVER and followed by #endif. Of course, you should ensure that the name NEVER isn’t defined anywhere.

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

```
#ifdef NEVER
{ …………………….
  …………………….                    /* code is skipped */
#endif
}
```

11.8 OTHER PRE-PROCESSOR COMMANDS

Other preprocessor commands are:

#ifndef If this macro is not defined

#if Test if a compile-time condition is true

#else The alternative for #if. This is part of an #if preprocessor statement and works in the same way with #if that the regular C else does with the regular if.

#elif enables us to establish an "if…else…if .." sequence for testing multiple conditions.

 # line #line number "string" – informs the preprocessor that the number is the next line of input. "string" is optional and names the next line of input. This is most often used with programs that translate other languages to C. For example, error messages produced by the C compiler can reference the file name and line numbers of the source files instead of the intermediate C (translated) source files.

#pragma It is used to turn on or off certain features. Pragmas vary from compiler to compiler. Pragmas available with Microsoft C compilers deals with formatting source listing and placing

comments in the object file generated by the compiler. Pragmas available with Turbo C compilers allow writing assembly language statements in a C program.

A control line of the form

```
#pragma        token-sequence
```

This causes the processor to perform an implementation-dependent action. An unrecognized pragma is ignored.

11.9 Pre-defined Names defined by Pre-Processor

ANSI C defines a number of macros. Although each one is available for use in programming, the predefined macros should not be directly modified.

Macro	Description
DATE	The current date as a character literal in "MMM DD YYYY" format.
TIME	The current time as a character literal in "HH:MM: SS" format.
FILE	This contains the current filename as a string literal.
LINE	This contains the current line number as a decimal constant.
STDC	Defined as 1 when the compiler complies with the ANSI standard.

Example:

```
#include <stdio.h>
main()
{
printf("File :%s\n", __FILE__ );
printf("Date :%s\n", __DATE__ );
printf("Time :%s\n", __TIME__ );
printf("Line :%d\n", __LINE__ );
```

```
printf("ANSI :%d\n", __STDC__ );
}
```

When the file **test.c** is compiled and executed, it produces the following result:

File :test.c

Date :Dec 04 2023

Time :13:26:14

Line :8 ANSI :1

11.10 MACROS VS FUNCTIONS

We have discussed about macros, any computations that can be done on macros can also be done on functions. But there is a difference in implementations and in some cases it will be appropriate to use macros than function and vice versa.

Macros	Functions
Macro calls are replaced with macro expansions (meaning).	In function call, the control is passed to a function definition along with arguments, and definition is processed and value may be returned to call.
Macros run programs faster but increase the program size.	Functions make program size smaller and compact.
If macro is called 100 numbers of times, the size of the program will increase.	If function is called 100 numbers of times, the program size will not increase.
It is better to use Macros, when the definition is very small in size.	It is better to use functions, when the definition is bigger in size.

11.11 CONCLUSION

The preprocessor makes programs easier to develop and modify. The preprocessor makes C code more portable between different machine architectures and customize the language. The C Preprocessor is not part of the compiler, but is a separate step in

the compilation process. All preprocessor lines begin with #. C Preprocessor is just a text substitution tool on your source code in three ways: File inclusion, Macro substitution, and Conditional compilation. File inclusion - inserts the contents of another file into your source file. Macro Substitution - replaces instances of one piece of text with another. Conditional Compilation - arranges source code depending on various circumstances.

UNIT 12 DYNAMIC MEMORY ALLOCATIONS

Allocation and De-allocation of Memory

12.0 Introduction
12.1 Objectives
12.2 Dynamic Memory Allocation
12.3 Resizing and Releasing Memory
12.4 Introduction to Memory Allocation in C.
12.4.1 Static Memory Allocation
12.4.2 Dynamic Memory Allocation
12.4.3 Difference Static & Dynamic Memory Allocation
12.5 malloc() stands for "memory allocation."
12.6 calloc() stands for "contiguous allocation."
12.7 realloc() stands for "reallocate allocation."
12.8 free() stands for release allocation
12.9 Conclusion
12.10 Unit-based Questions /Answers

12.0 INTRODUCTION

This unit discusses dynamic memory allocation in the C programming language is a crucial aspect of managing memory resources during program execution. Unlike static memory allocation, which occurs at compile time, dynamic memory allocation allows for the allocation and deallocation of memory at runtime, offering flexibility in memory usage. Much of the power of pointers stems from their ability to track dynamically allocated memory. The management of this memory through pointers forms the basis for many operations, including those used to manipulate complex data structures.

C program executes within a *runtime system*. This is typically the environment provided by an operating system. The runtime system supports the stack and heap along with other program behavior.

Memory management is central to all programs. Sometimes memory is managed by the runtime system implicitly, such as when memory is allocated for automatic variables. In this case, variables are allocated to the enclosing function's stack frame. In the case of static and global variables, memory is placed in the application's data segment, where it is zeroed out. This is a separate area from executable code and other data managed by the runtime system. Instead of having to allocate memory to accommodate the largest possible size for a data structure, only the actual amount required needs to be allocated.

This unit will discuss the dynamic memory allocation process of C language.

12.1 OBJECTIVES

After completing this unit, you will be able to:

- Learn how to allocate and free memory, and to control dynamic arrays of any type of data in general and structures in particular.
- Practice and train with dynamic memory in the world of work oriented applications.
- To know about the pointer arithmetic
- How to create and use array of pointers.

12.2 DYNAMIC MEMORY ALLOCATION

The Creating and maintaining dynamic structures requires dynamic

memory allocation the ability for a program to obtain more memory space at execution time to hold new values, and to release space no longer needed. While doing programming, if you are aware about the size of an array, then it is easy and you can define it as an array.

Example:

To store a name of any person, it can go max 100 characters as follows:

```
char name[100]
```

But now let us consider a situation where you have no idea about the length of the text you need to store, for example you want to store a detailed description about a topic. Here we need to define a pointer to character without defining how much memory is required and later based on requirement we can allocate memory.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
        char name[100];
        char *description;
        strcpy(name, "Raghav Rai");
        /* allocate memory dynamically */
        description = malloc( 200 * sizeof(char) );
        if( description == NULL )
        {
        fprintf(stderr, "Error - unable to allocate required
memory\n");
        }
```

```
else
{
strcpy( description, "Raghav Rai a MVM student in class
11th");
}
printf("Name = %s\n", name );
printf("Description: %s\n", description );
}
```

The following result:

```
Name = Raghav Rai
Description: Raghav Rai a MVM student in class 11th
```

In case this program can be written using calloc(); only thing is you need to replace malloc with calloc as follows:

```
calloc(200, sizeof(char));
```

The complete control and pass any size value while allocating memory, unlike arrays where once the size is defined, then after you cannot change it.

12.3 RESIZING AND RELEASING MEMORY

When your program comes out, operating system automatically release all the memory allocated by your program but as a good practice when you are not in need of memory anymore then you should release that memory by calling the function free().

Alternatively, you can increase or decrease the size of an allocated memory block by calling the function realloc().

Lets the program once again and make use of realloc() and free() functions:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
	char name[100];
	char *description;
	strcpy(name, "Raghav Rai");

	/* allocate memory dynamically */
	description = malloc( 30 * sizeof(char) );
	if( description == NULL )
	{
	fprintf(stderr, "Error - unable to allocate required memory\n");
	}
	else
	{
	strcpy( description, " Raghav Rai a MVM student in class 11th.");
	}
	/* suppose you want to store bigger description */
	description = realloc( description, 100 * sizeof(char) );
	if( description == NULL )
	{
	fprintf(stderr, "Error - unable to allocate required memory\n");
	}
	else
	{
```

```
strcat( description, "He is in class 11th");
}
printf("Name = %s\n", name );
 printf("Description: %s\n", description );
 /* release memory using free() function */
free(description);
}
```

The following result:

```
Name = Raghav Rai
Description: Raghav Rai a MVM student. He is in class 11th
```

This example without re-allocating extra memory, and strcat() function will give an error due to lack of available memory in description.

12.4 INTRODUCTION TO MEMORY ALLOCATION IN ‘C’

Memory allocations, in general, mean where computer programs and services are executed to reserve partially or complete space or virtual memory of a computer, this process is known as memory allocation.

This process is hardware operation and is achieved by memory management through Operating systems and software applications. In general, there are static and dynamic memory allocations, whereas, in C programming language, we will see about dynamic memory allocation where programs are allocated during run time in memory and static memory allocation is a process of allocating memory while writing the C program which means memory is allocated at compile time.

12.4.1 Static Memory Allocation in 'C'

As we discussed static memory allocation is the allocation of memory for the data variables when the computer programs start. This type of allocation is applied to only global variables, file scope variables and also to those variables that are declared as static. This type of allocation is having a drawback when you are allocating memory, we should know the exact memory before allocating as this process allocates fixed memory and cannot be changed after allocating.

There are a few features of static memory allocation. They are: this type of allocation allocates variables permanently; hence the memory in this type of allocation cannot be reused and is, therefore, less efficient. This allocation uses the stack for implementing the allocation process.

Example:

```
#include <stdio.h>
void play
{
        int x;
}
        int main()
{
        int y;
        int c[10];
        return 1;
}
```

A variable can internally or externally be declared as static in which its value persists until the end of the program, where this

can be done using the keyword static before the variable declaration. There can be internal or external static variables that are declared inside or outside the function.

Example:

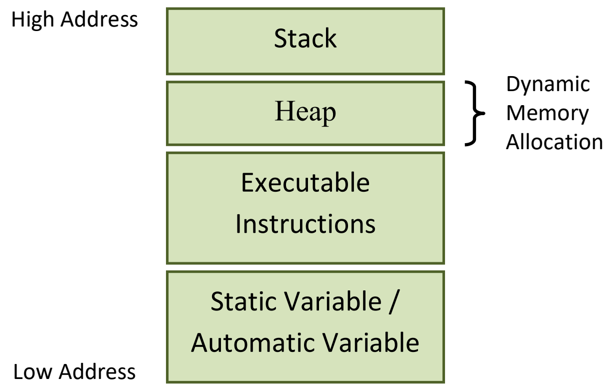
```
#include<stdio.h>
void stat(void);
int main()
{
        int i;
        for(i=1; i<=3 ; i++)
        stat();
        return 1;
}
void stat(void)
{
        static int n = 0;
        n = n+1;
        printf("n = %d""\n", n);
}
```

The following result:

```
n = 1
n = 2
n = 3
```

12.4.2 Dynamic Memory Allocation in 'C'

As discussed above dynamic memory allocation is allocation of memory during runtime or during program execution. Dynamic memory allocation provides different functions in the C programming language.



a). Stack: In this section, local variable or automatic variable and information regarding the address of function call are stored such as stack pointer.

b). Heap: This is the part of memory where dynamic memory allocation take place. Now for dynamic memory allocation following standard library functions are essential that are defined in the standard library.

12.4.3 Difference Static & Dynamic Memory Allocation

Let's understand the difference between static memory allocation and dynamic memory allocation.

Static Memory Allocation	Dynamic Memory Allocation
Memory is allocated at compile time.	Memory is allocated at run time.
Memory cannot be increased while executing program.	Memory can be increased while executing program.
Used in array.	Used in linked list.

Methods used for dynamic memory allocation.

malloc()	allocates single block of requested memory.

calloc()	allocates multiple block of requested memory.
realloc()	reallocates the memory occupied by malloc() or calloc() function.
rree()	free the dynamically allocated memory.

12.5 MALLOC() STANDS FOR MEMORY ALLOCATION

The malloc() function allocates single block of requested memory at runtime. This function reserves a block of memory of given size and returns a pointer of type void. This means that we can assign it to any type of pointer using typecasting. It doesn't initialize memory at execution time, so it has garbage value initially. If it fails to locate enough space (memory) it returns a NULL pointer.

Syntax:

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

Example:

int *x; x = (int*)malloc(100 * sizeof(int)); //memory space allocated to variable x

free(x); //releases the memory allocated to variable x

This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

Example:

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

```
int main()
{
        int num, i,
        *ptr, sum = 0;
        printf("Enter number of elements: ");
        scanf("%d", &num);
        ptr = (int*) malloc(num * sizeof(int));         //memory
allocated using malloc
        if(ptr == NULL)
          {
        printf("Error! memory not allocated.");
        exit(0);
          }
        printf("Enter elements of array: ");
        for(i = 0; i < num; ++i)
         {
        scanf("%d", ptr + i);
        sum += *(ptr + i);
         }
        printf("Sum = %d", sum);
        free(ptr);
        return 0;
         }
```

12.6 CALLOC() STANDS FOR MEMORY ALLOCATION

Calloc() is another memory allocation function that is used for allocating memory at runtime. calloc function is normally used for allocating memory to derived data types such as arrays and structures. The calloc() function allocates multiple blocks of requested memory.

It initially initialize (sets) all bytes to zero.If it fails to locate enough space(memory) it returns a NULL pointer. The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size.

Syntax:

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

calloc() required 2 arguments of type count, size-type.

Count will provide number of elements; size-type is data type size

Example:

```
int*arr;
arr=(int*)calloc(10, sizeof(int));                    // 20 byte
cahr*str; str=(char*)calloc(50, siceof(char)); // 50 byte
```

Example:

```
struct employee
{
char *name;
int salary;
};
typedef struct employee emp;
emp *e1;
e1 = (emp*)calloc(30,sizeof(emp));
```

Example:

```
#include <stdio.h>
#include <stdlib.h>
    int main()
    {
    int num, i, *ptr, sum = 0;
    printf("Enter number of elements: ");
    scanf("%d", &num);
    ptr = (int*) calloc(num, sizeof(int));
```

```
if(ptr == NULL)
{
printf("Error! memory not allocated.");
exit(0);
}
printf("Enter elements of array: ");
for(i = 0; i < num; ++i)
{
scanf("%d", ptr + i);
sum += *(ptr + i);
}
printf("Sum = %d", sum);
free(ptr);
return 0;
}
```

12.7 REALLOC() STANDS FOR MEMORY ALLOCATION

Changes memory size that is already allocated to a variable. Or If the previously allocated memory is insufficient or more than required, you can change the previously allocated memory size using realloc().

If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size. By using realloc() we can create the memory dynamically at middle stage. Generally, by using realloc() we can reallocation the memory. Realloc() required 2 arguments of type void*, size_type. Void* will indicates previous block base address, size-type is data type size. Realloc() will creates the memory in bytes format and initial value is garbage.

Syntax:

ptr=realloc(ptr, new-size)

Example:

```
int *x;
x=(int*)malloc(50 * sizeof(int));
x=(int*)realloc(x,100); //allocated a new memory to variable x
```

Example:

```
void*realloc(void*, size-type);
int *arr;
arr=(int*)calloc(5, sizeof(int));
…………………
…………………
…………………

arr=(int*)realloc(arr,sizeof(int)*10);
```

Example:

```
#include <stdio.h>
#include <stdlib.h>
        int main()
{
        int *ptr, i , n1, n2;
        printf("Enter size of array: ");
        scanf("%d", &n1);
        ptr = (int*) malloc(n1 * sizeof(int));
        printf("Address of previously allocated memory: ");
        for(i = 0; i < n1; ++i)
        printf("%u\t",ptr + i);
        printf("\nEnter new size of array: ");
        scanf("%d", &n2);
        ptr = realloc(ptr, n2);
```

```
    for(i = 0; i < n2; ++i)
    printf("%u\t", ptr + i);
return 0;
}
```

12.8 FREE() STANDS FOR MEMORY ALLOCATION

When your program comes out, the operating system automatically releases all the memory allocated by your program but as a good practice when you are not in need of memory anymore then you should release that memory by calling the function free().

The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit. Or Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on its own. You must explicitly use free() to release the space.

Syntax:

free(ptr);

Example:

```
#include
#include
int main()
{
int num, i, *ptr, sum = 0;
printf("Enter number of elements: ");
scanf("%d", &num);
ptr = (int*) malloc(num * sizeof(int));                    //memory
allocated using malloc
```

```
if(ptr == NULL)
{
printf("Error! memory not allocated.");
exit(0);
}
printf("Enter elements of array: ");
for(i = 0; i < num; ++i)
{
scanf("%d", ptr + i);
sum += *(ptr + i);
}
printf("Sum = %d", sum);
free(ptr);
return 0;
}
```

12.9 CONCLUSION

Memory allocation in C programming language is simple using static memory allocation which allocates memory during compile time or we can say before the program execution and it also has another type known as dynamic memory allocation which allocates memory during run time or allocating memory during program execution which uses four different functions such as malloc(), calloc(), free() and realloc(). There are different pros and cons of both methods.

UNIT 13 ERROR HANDLING TECHNIQUES

Structure

13.0 Introduction
13.1 Objectives
13.2 Error Handling in ‘C’
13.2.1 NSError
13.2.2 Patterns for error handling
13.2.3 Errno values and meaning
13.3 Different types of errors exit in ‘C’
13.4 The Perror () Function
13.5 The Sterror () Function
13.6 Uses of the Error Function
13.7 The uses of clearer () Function
13.8 Divide by Zero Errors
13.9 Program Exit Status
13.10 Conclusion
13.11 Unit-based Questions /Answers

13.0 INTRODUCTION

This unit discusses as such C programming does not provide direct support for error handling, but being a system programming language, it provides you access at lower level in the form of return values. Most of the C or even Unix function calls return -1 or NULL in case of any error and set an error code **errno.** It is set as a global variable and indicates an error occurred during any function call. You can find various error codes defined in the header file. So a C programmer can check the returned values and can take appropriate action depending on the return value. It is a good practice to set errno to 0 at the time of initializing a program.

A value of 0 indicates that there is no error in the program.

This unit will discuss the error handling system process of the language C.

13.1 OBJECTIVES

After completing this unit, you will be able to:

- 'C' programming error handling is provided with NSError class available in the Foundation framework.
- An NSError object encapsulates richer and more extensible error information than is possible using only an error code or error string.
- An NSError object are an error domain (represented by a string)
- A domain-specific error code and a user info dictionary containing application-specific information.

13.2 ERROR HANDLING IN 'C'

Error handling is a big part of writing software, and when it is done poorly, the software becomes difficult to extend and to maintain. Programming languages like C++ or Java provide exceptions and destructors that make error handling easier. Such mechanisms are not natively available for C, and the literature on good error handling in C is widely scattered over the internet.

Errors are the problems or the faults that occur in the program, which make the behavior of the program abnormal, and experienced developers can also make these faults. Programming errors are also known as bugs or faults, and the process of

removing these bugs is known as **debugging**.

13.2.1 NSError

C programs use NSError objects to convey information about runtime errors that users need to be informed about. In most cases, a program displays this error information in a dialog or sheet. But it may also interpret the information and either ask the user to attempt to recover from the error or attempt to correct the error on its own.

NSError Object consists of –

Domain: The Error domain can be one of the predefined NSError domains an arbitrary string describing a custom domain and domain must not be nil.

Code: The error code for the error.

User Info: The User Info dictionary for the error and user Info may be nil.

Example:

```
NSString *domain = @"com.MyCompany.MyApplication.ErrorDomain";
NSString *desc = NSLocalizedString(@"Unable to complete the process", @"");
NSDictionary *userInfo = @{ NSLocalizedDescriptionKey : desc };
NSError *error = [NSError errorWithDomain:domain code: 101 userInfo:userInfo];
```

13.2.2 Pattern for Error Handling

A collected knowledge on good error handling in the form of C error-handling patterns and a running example that applies the

patterns. The patterns provide good practice design decisions and elaborate on when to apply them and which consequences they bring. For a programmer, these patterns remove the burden of making many fine-grained decisions. Instead, a programmer can rely on the knowledge presented in these patterns and use them as a starting point to write good code.

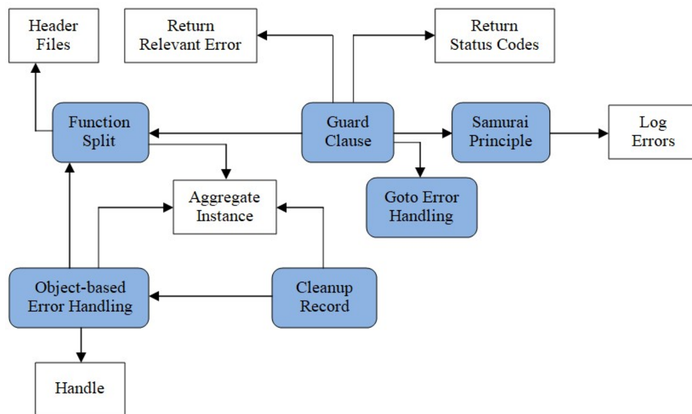


Figure: An overview of the patterns and their relationships

Pattern_Name	Description
Function Split	The function has several responsibilities, which makes the function hard to read and maintain. Therefore, split it up. Take a part of a function that seems useful on its own, create a new function with that, and call that function.
Guard Clause	The function is hard to read and maintain because it mixes pre-condition checks with the main program logic of the function. Therefore, check whether you have mandatory pre-conditions and immediately return from the function if these pre-conditions are not met.
Samurai Principle	When returning error information, you assume that the caller checks for this information. However, the caller can simply omit this check and the error might go unnoticed. Therefore, return from a function victorious or not at all. If there is a situation for which you know that an error cannot be handled, then abort the program.
Goto Error Handling	Code gets difficult to read and maintain if it acquires and cleans up multiple resources at different places within a function. Therefore, have all resource cleanup and error handling at the end of the function. If a resource cannot be

	acquired, use the `goto` statement to jump to the resource cleanup code.
Cleanup Record	It is difficult to make a piece of code easy to read and maintain if this code acquires and cleans up multiple resources, particularly if those resources depend on one another. Therefore, call resource acquisition functions as long as they succeed, and store which functions require cleanup. Call the cleanup functions depending on these stored values.
Object-Based Error Handling	Having multiple responsibilities in one function, such as resource acquisition, resource cleanup, and usage of that resource, makes that code difficult to implement, read, maintain, and test. Therefore, put initialization and cleanup into separate functions, similar to the concept of constructors and destructors in object-oriented programming.

13.2.3 Errno Values and Meaning

errno is a global variable indicating the error occurred during any function call, and it is defined inside <errno.h> header file. When a function is called in C, a variable named errno is automatically assigned a code (value) which can be used to identify the type of error that has been encountered. Different code values for errno mean different types of errors.

error value	Error
1	Operation not permitted
2	No such file or directory
3	No such process
4	Interrupted system call
5	I/O error
6	No such device or address
7	The argument list is too long
8	Exec format error
9	Bad file number
10	No child processes
11	Try again
12	Out of memory
13	Permission denied

Example:

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

```
int main()
{
    // If a file is opened which does not exist,
    // then it will be an error and corresponding
    // errno value will be set

    FILE* fp;
    // opening a file which does not exist
    fp = fopen("Error Handling.txt", "r");
    printf("Value of errno: %d\n", errno);
    return 0;
}
```

13.3 DIFFERENT TYPES OF ERRORS EXIT IN 'C.'

These errors are detected either during the time of compilation or execution. Thus, the errors must be removed from the program for the successful execution of the program.

There are mainly five types of errors exist in C programming:

- Syntax error
- Run-time error
- Linker error
- Logical error
- Semantic error

a). Syntax error:

Syntax errors are also known as the compilation errors as they occurred at the compilation time, or we can say that the syntax errors are thrown by the compilers. These errors are mainly occurred due to the mistakes while typing or do not follow the

syntax of the specified programming language. These mistakes are generally made by beginners only because they are new to the language. These errors can be easily debugged or corrected.

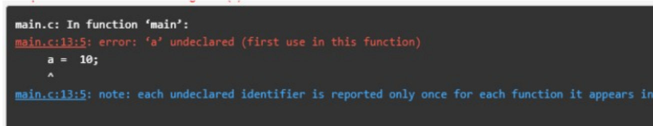
The syntax errors are:

- If miss the parenthesis (}) while writing the code.
- Displaying the value of a variable without its declaration.
- If you miss the semicolon (;) at the end of the statement.

Example:

```
#include <stdio.h>
int main()
{
   a = 10;
   printf("The value of a is : %d", a);
  return 0;
}
```

The following result:



Example:

```
#include <stdio.h>
int main()
{
   a = 2;
   if(.)    // syntax error
   printf("a is greater than 1");
  return 0;
  }
```

The following result:

```
main.c: In function 'main':
main.c:15:6: error: expected expression before '.' token
   if(.)
      ^
```

b). Run-time error:

Sometimes the errors exist during the execution time, even after the successful compilation, known as run-time errors. When the program is running, and it is not able to perform the operation is the main cause of the run-time error. The division by zero is the common example of the run-time error. These errors are very difficult to find, as the compiler does not point to these errors.

```
#include <stdio.h>
int main()
{
    int a=2;
    int b=2/0;
    printf("The value of b is : %d", b);
    return 0;
}
```

The following result:

```
main.c:14:12: warning: division by zero [-Wdiv-by-zero]
Floating point exception


...Program finished with exit code 136
Press ENTER to exit console.
```

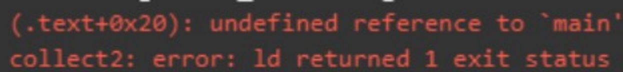
c). Linker error:

Linker errors are mainly generated when the executable file of the program is not created. This can be happened either due to the wrong function prototyping or usage of the wrong header file. For

example, the **main.c** file contains the **sub()** function whose declaration and definition is done in some other file such as **func.c**. During the compilation, the compiler finds the **sub()** function in **func.c** file, so it generates two object files, i.e., **main.o** and **func.o**. At the execution time, if the definition of **sub()** function is not found in the **func.o** file, then the linker error will be thrown. The most common linker error that occurs is that we use **Main()** instead of **main().**

```
#include <stdio.h>
int Main()
{
   int a=78;
   printf("The value of a is : %d", a);
   return 0;
}
```

The following result:



Logical error:

The logical error is an error that leads to an undesired output. These errors produce the incorrect output, but they are error-free, known as logical errors. These types of mistakes are mainly done by beginners. The occurrence of these errors mainly depends upon the logical thinking of the developer. If the programmers sound logically good, then there will be fewer chances of these errors.

```
#include <stdio.h>
int main()
{
  int sum=0; // variable initialization
  int k=1;
```

```
  for(int i=1;i<=10;i++); // logical error, as we put the semicolon a
fter loop
  {
     sum=sum+k;
     k++;
  }
printf("The  value of sum is %d", sum);
   return 0;
}
```

The following result:

```
The  value of sum is 1

...Program finished with exit code 0
Press ENTER to exit console.
```

Semantic error:

Semantic errors are the errors that occurred when the statements are not understandable by the compiler.

The following can be the cases for the semantic error:

Use of a un-initialized variable.

```
int i;
i=i+2;
```

Type compatibility

```
int b = "Error Handling";
```

Errors in expressions

```
int a, b, c;
a+b = c;
```

Array index out of bound

```
int a[10];
a[10] = 34;
```

```
      #include <stdio.h>
      int main()
```

```
{
int a,b,c;
a=2;
b=3;
c=1;
a+b=c;    // semantic error
return 0;
}
```

The following result:

```
main.c: In function 'main':
main.c:17:6: error: lvalue required as left operand of assignment
   a+b=c;
      ^
```

13.4 THE PERROR () FUNCTION

The **perror()** function is used to show the error description. It displays the string you pass to it, followed by a colon, a space, and then the textual representation of the current errno value.

Syntax

*void **perror**(const char* *str*);*

Parameters

str: It is a string containing a custom message to be printed before the error message itself.

Example

```
 // C implementation to see how perror() function is used to
 // print the error messages.
#include <errno.h>
#include <stdio.h>
#include <string.h>
 int main()
{
   FILE* fp;
   // If a file is opened which does not exist,
   // then it will be an error and corresponding
```

```
    // errno value will be set
    fp = fopen(" Error Handling.txt ", "r");
    // opening a file which does
    // not exist.
    printf("Value of errno: %d\n ", errno);
    perror("Message from perror");
    return 0;
}
```

The following result:

```
Value of errno: 2
Message from perror: No such file or directory
```

13.5 THE STERROR () FUNCTION

The **strerror()** function is also used to show the error description. This function returns a pointer to the textual representation of the current errno value.

Syntax

```
char *strerror(int errnum);
```

Parameters

errnum: It is the error number (errno).

Example:

```
// C implementation to see how strerror() function is used
// to print the error messages.
#include <errno.h>
#include <stdio.h>
#include <string.h>
 int main()
{
    FILE* fp;
   // If a file is opened which does not exist,
```

```
  // then it will be an error and corresponding
 // errno value will be set
    fp = fopen(" Error Handling.txt ", "r");
 // opening a file which does
 // not exist.
    printf("Value of errno: %d\n", errno);
    printf("The error message is : %s\n", strerror(errno));
    return 0;
}
```

The following result:

```
Value of errno: 2
The error message is: No such file or directory
```

13.6 USES OF FERROR () FUNCTION

The **ferror()** function is used to check whether an error occurred during a file operation.

Syntax

```
int ferror(FILE *stream);
```

Parameters

stream: It is the pointer that points to the FILE for which we want to check the error.

Return Value

It returns a non-zero value if an error occurred, otherwise it returns 0.

Example

```
// C program to demonstrate the ferror() function
#include <stdio.h>
  int main()
```

```
{
   // Open the file in read mode
   FILE* file = fopen("nonexistent_file.txt", "r");
   if (file == NULL)
{
      // Print an error message
      // if file opening fails
      perror("Error opening file");
      // Return with non-zero exit status to
      // indicate an error
      return 1;
   }
    int c;
   // Process the character
   // Add your code here to perform operations on each
   // character read from the file
   while ((c = fgetc(file)) != EOF) {
   }
    if (ferror(file)) {
      // Print an error message if an error occurred
      // during file reading
      printf(
         "An error occurred while reading the file.\n");
   }
   else {
      // Print success message if file reading completed
      // without errors
      printf("File read successfully.\n");
   }
   // Close the file
   fclose(file);
   // Return with zero exit status to indicate successful
```

```
    // execution
    return 0;
}
```

The following result:

```
Error opening file: No such file or directory
```

13.7 THE USES OF CLEARER () FUNCTION

The **clearerr()** function is used to clear both end-of-file and error indicators for a file stream.

Syntax

```
void clearerr(FILE *stream);
```

Parameters

stream: It is the pointer that points to the FILE for which we want to check the error.

Example

```
#include <stdio.h>
  int main()
{
    FILE* file = fopen("file.txt", "r");
    // Open the file in read mode
    if (file == NULL)
{
    // Print an error message
    // if file opening fails
        perror("Error opening file");
    // Return with non-zero exit status to
    // indicate an error
        return 1;
```

```
    }
    // Perform file operations
    // Add your code here to perform operations on the
    // opened file
     if (ferror(file)) {
    // Print an error message
    // if an error occurred
    // during file operations
        printf("An error occurred while performing file "
               "Operations.\n");
    }
    // Clear the error indicators for the
    // file stream
    clearerr(file);
  // Continue with file operations
  // Add your code here to continue with further
  // operations on the file
  // Close the file
     fclose(file);
     return 0;
}
```

13.8 DIVIDE BY ZERO ERRORS FUNCTION

A common pitfall made by C programmers is not checking if a divisor is zero before a division command. Division by zero leads to undefined behavior, there is no C language construct that can do anything about it. Your best bet is to not divide by zero in the first place, by checking the denominator.

Example:

```
// C program to check and rectify
// divide by zero condition
#include <stdio.h>
#include <stdlib.h>
  void function(int);
  int main()
{
    int x = 0;
    function(x);
    return 0;
}
  void function(int x)
{
    float fx;
    if (x == 0) {
    printf("Division by Zero is not allowed");
    fprintf(stderr, "Division by zero! Exiting...\n");
    exit(EXIT_FAILURE);
    }
    else
{
    fx = 10 / x;
    printf("f(x) is: %.5f", fx);
}
}
```

The following result:

Division by zero! Exiting ….

13.9 PROGRAM EXIT STATUS

Exit status is the value returned by the program after its execution is completed which tells the status of the execution of the program.

The C standard specifies two constants: **EXIT_SUCCESS** and **EXIT_FAILURE**, that may be passed to exit() to indicate successful or unsuccessful termination, respectively. *These are macros defined in <stdlib.h> header file.*

Example:

```
// C implementation which shows the
// use of EXIT_SUCCESS and EXIT_FAILURE.
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
 int main()
{
  FILE* fp;
  fp = fopen("filedoesnotexist.txt", "rb");
  if (fp == NULL)
{
 printf("Value of errno: %d\n", errno);
 printf("Error opening the file: %s\n",
 strerror(errno));
 perror("Error printed by perror");
 exit(EXIT_FAILURE);
 printf("I will not be printed\n");
  }
 else
```

```
{
  fclose(fp);
   exit(EXIT_SUCCESS);
   printf("I will not be printed\n");
   }
   return 0;
   }
```

The following result:

```
Value of errno: 2
Error opening the file: No such file or directory
Error printed by perror: No such file or directory
```

13.9 CONCLUSION

This chapter showed you how to perform error handling in C. Function Split tells you to split your functions into smaller parts to make error handling of these parts easier. A Guard Clause for your functions checks pre-conditions of your function and returns immediately if they are not met. This leaves fewer error-handling obligations for the rest of that function. Instead of returning from the function, you could also abort the program, adhering to the Samurai Principle. When it comes to more complex error handling a ”particularly in combination with acquiring and releasing resources a ”you have several options. Goto Error Handling makes it possible to jump forward in your function to an error-handling section. Instead of jumping, Cleanup Record stores the info, which resources require cleanup, and performs it by the end of the function. A method of resource acquisition that is closer to object-oriented programming is Object-Based Error Handling, which uses separate initialization and cleanup functions similar to the concept of constructors and destructors.

With these error-handling patterns in your repertoire, you now have the skill to write small programs that handle error situations in a way that ensures the code stays maintainable.

UNIT 14 STANDARDS I/O OPERATIONS

Methods of working with Files

14.0 Introduction

14.1 Objectives

14.2 Introduction to Input and Output

14.3 Standard Input/Output System

14.3.1 A File using the Function fopen()

14.3.2 A File using the Function fclose()

14.4 Character Input and Output in Files

14.4.1 The getchar() Function

14.4.2 The putchar() Function

14.4.3 Printf(format, values)

14.4.4 Printf Substitution Types

14.4.5 Printf Substitution Modifier

14.5 String Input/Output Functions

14.6 Block Input/Output Functions

14.7 Sequential Vs Random Access Files

14.8 Positioning the File Pointer

14.9 The Unbuffered I/O

14.10 Conclusion

14.11 Unit-based Questions /Answers

14.0 INTRODUCTION

This unit discusses the examples we have seen so far in the previous units that deal with standard input and output. When data is stored using variables, the data is lost when the program exits unless something is done to save it. This unit discusses methods of working with files, and a data structure to store data. C views file simply as a sequential stream of bytes. Each file ends either with an end-of-file marker or at a specified byte number recorded in a

system maintained, administrative data structure. C supports two types of files called binary files and text files. The difference between these two files is in terms of storage. In text files, everything is stored in terms of text i.e. even if we store an integer 54; it will be stored as a 3-byte string - “54\0”. In a text file, certain character translations may occur. For example, a newline(\n) character may be converted to a carriage return, linefeed pair. This is what Turbo C does. Therefore, there may not be to one relationship between the characters that are read or written and those in the external device. A binary file contains data that was written in the same format used to store internally in main memory.

The fact that a numeric value is in a standard length makes binary files easier to handle. No special string-to-numeric conversions is necessary. The disk I/O in C is accomplished through the use of library functions. The ANSI standard, which is followed by TURBO C, defines one complete set of I/O functions. But since originally C was written for the UNIX operating system, the UNIX standard defines a second system of routines that handles I/O operations. The first method, defined by both standards, is called a buffered file system. The second is the unbuffered file system.

This unit will discuss the buffered file functions of C language.

14.1 OBJECTIVES

After completing this unit, you will be able to:

- define the concept of file pointer and file storage in C;
- create text and binary files in C;

- read and write from text and binary files;
- deal with large set of Data such as a File of Records; and
- perform operations on files such as count the number of words in a file, search a word in a file, compare two files etc.

14.2 INTRODUCTION TO INPUT AND OUTPUT

There are no built-in input or output statements in the C language. This means that all input and output must be done by calling functions, at least some of which must be written in a language other than C. There is, however, a standard library of functions that allow I/O to be done in a relatively uniform manner for all C implementations without the need to know how it is being done in any particular case.

This standard library is known as the "stdio" library and can be considered to be an extension to the C language itself. To be able to use the functions within this library, it is necessary to insert the following preprocessor statement at the start of the program source code:

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

The following functions are part of the "stdio" library

getchar for reading single characters from the standard input, usually the keyboard.

gets for reading a whole line of characters from the standard input, usually the keyboard.

putchar for writing single characters to the standard output,

usually the screen.

printf for writing more complex, formatted output to the standard output.

Functions getchar, putchar, and printf are used extensively in the program and many other input and output.

14.3 STANDARD INPUT/OUTPUT SYSTEM

A sequential stream of bytes ending with an end-of-file marker is what is called a file. When the file is opened, the stream is associated with the file. By default, three files and their streams are automatically opened when program execution begins - the standard input, standard output, and the standard error. Streams provide communication channels between files and programs. For example, the standard input stream enables a program to read data from the keyboard, and the standard output stream enables to write data on the screen. Opening a file returns a pointer to a FILE structure (defined in <stdio.h>) that contains information, such as size, current file pointer position, type of file etc., to perform operations on the file. This structure also contains an integer called a file descriptor, which is an index into the table maintained by the operating system, namely, the open file table. Each element of this table contains a block called file control block (FCB) used by the operating system to administer a particular file.

The standard input, standard output and the standard error are manipulated using file pointers stdin, stdout and stderr. The set of functions which we are now going to discuss come under the category of buffered file system. This file system is referred to as

buffered because, the routines maintain all the disk buffers required for reading / writing automatically. To access any file, we need to declare a pointer to FILE structure and then associate it with the particular file.

It is declared as follows:

FILE *fp;

14.3.1 A File using the Function fopen()

Once a file pointer variables has been declared, the next step is to open a file. The fopen() function opens a stream for use and links a file with that stream. This function returns a file pointer, described in the previous section.
The syntax is:

FILE *fopen(char *filename,*mode);

where mode is a string, containing the desired open status. The filename must be a string of characters that provide a valid file name for the operating system and may include a path specification.

Example:

```
#include <stdio.h>
main ()
{
FILE *fp;
if ((fp=fopen(“file1.dat”, “r”))==NULL)
{
printf(“FILE DOES NOT EXIST\n”);
```

```
exit(0);
}
}
```

14.3.2 A File using the Function fclose()

When the processing of the file is finished, the file should be closed using the fclose() function,

The syntax is:

int fclose(FILE *fptr);

This function flushes any unwritten data for stream, discards any unread buffered input, frees any automatically allocated buffer, and then closes the stream. The return value is 0 if the file is closed successfully or a constant EOF, an end-of file marker, if an error occurred. This constant is also defined in. If the function fclose() is not called explicitly, the operating system normally will close the file when the program execution terminates.

Example:

```
#include <stdio.h>
main ()
{
FILE *fp;
if ((fp=fopen("file1.dat", "r"))==NULL)
{
printf("FILE DOES NOT EXIST\n");
exit(0);
}
……………..
……………..
……………..
…………….
```

```
/* close the file */
fclose(fp);
}
```

14.4 CHARACTER INPUT AND OUTPUT IN FILES

ANSI C provides a set of functions for reading and writing character by character or one byte at a time. These functions are defined in the standard library. They are:

- The getchar() Function
- The putchar() Function
- The Printf() Function
- The Scanf() Function
- Printf Substitution Types
- Printf Substitution Modifier

14.4.1 The getchar() Function

This will get a character from the keyboard, waiting as long as required for a key to be pressed. When the character is found, it's corresponding bit pattern (usually the ASCII value) is assigned to abc.

This function is:

abc = getchar();

getc( ) is used to read a character from a file and putc( ) is used to write a character to a file.

Their syntax is:

int putc(int ch, FILE *stream);

```
int getc(FILE *stream);
```

If the function encounters an error, such as an end of file marker, it returns the value EOF (this type of error is unusual from a keyboard!). EOF is a constant defined in the stdio.h header file described in section C

EOF is equivalent to -1 on most systems.

It is possible to use getchar without assigning the resulting value to any variable, such as in the statement:

```
getchar();
```

14.4.2 The putchar() Function

putchar outputs a character to the screen as follows:

```
putchar(character_expression);
```

This will output to the screen whatever character has a bit pattern that corresponds to the value of the specified parameter in the ( ).

Example:

```
int fred = 65;
putchar('x'); /* outputs character 'x' */
putchar(fred); /* outputs character 'A' */
```

1. putchar('x'); is not the same as putchar(x);
The former outputs the character 'x'.
The latter looks at the variable called x and prints the character with a bit pattern that corresponds to the value stored in x.
2. 'x' is not the same as "x".
"x" is a string as used in printf.
3. putchar(2); will not put a 2 onto the screen.
The bit pattern of the number 2 does not correspond to the ASCII value of a character that is "printable".

i.e. Nothing will appear on the screen.

Their syntax is:

```
int fgetc(FILE *stream);
int fputc(int c, FILE *stream);
```

14.4.3 The Printf() Function

printf outputs a string of characters to the

```
eg. printf("\nHello World");
```

1. A string is a collection of one or more characters with a hidden zero byte at the end.
2. A string with one character in it is not the same as a single character. Double quotes, " ", enclose a string, single quotes ' ' enclose a character.
3. The \n in the above example makes sure the output is on a new line . . . . . . otherwise, it would have continued where it previously left off.
4. If the string contains a %d then the output is modified by substituting the value of the next parameter in place of the %d.

For each further %d another parameter value is substituted.

```
eg. printf("Add %d and %d to get %d.",a,b,a+b);
```

An expression can be used for a printf parameter as in a+b .

14.4.4 The Scanf() Function

The int scanf(const char *format, ...) function reads the input from the standard input stream stdin and scans that input according to the format provided.

The format can be a simple constant string, but you can specify

%s, %d, %c, %f, etc., to print or read strings, integer, character, or float, respectively. There are many other formatting options available which can be used based on requirements.

Example:

```
#include <stdio.h>
int main( )
{
char str[100];
int i;
printf( "Enter a value :");
scanf("%s %d", str, &i);
printf( "\nYou entered: %s %d ", str, i);
return 0;
}
```

14.4.5 Printf Substitution Types

Wherever a % is found in the printf output string, the next character will be

%d will substitute the decimal value of the next parameter.

Other % character substitutions cause the next parameter to be interpreted in different forms.

%u output as an unsigned decimal number

%o output as an octal number %x output as a hexadecimal number

%c output as a character %s output as a string

%f output as a real number with 6 decimal places

%e output as a real number in scientific notation

%g output as a real number in ordinary or scientific notation, whichever takes the least space

%% output a '%' character

14.4.6 Printf Substitution Modifier

The normal %d output will substitute the minimum number of digits with a leading - if it is negative.

This may not be neat or convenient if, say, a table of figures is to be output, but, a field width modifier can be used to specify how many character positions the substituted value will take up.

eg. ***printf("Height is %5d metres",size);***

There will always be five characters substituted for the %5d, if the value is between -999 and 9999 then the output will be padded with spaces on the left.
Any print type can have a field width modifier.

eg. ***printf("Gender:%5c\n", sex);***

A further modifier can alter the number of decimal places output for real numbers printed with a %f, %e or %g, in the form:

printf("Average is %10.5f\n", xyz);

Where: 10 gives the total width of the field including digits, the decimal point and possible - sign.
.5 gives the number of digits after the decimal point.

14.5 STRING INPUT/OUTPUT FUNCTIONS

The file, then each time we will need to call the character input function, instead, C provides some string input/output functions with the help of which we can read/write a set of characters at one time.

These are defined in the standard library:

- fgets( )
- fputs( )

These functions are used to read and write strings.
The syntax is:

int fputs(char *str, FILE *stream);
char *fgets(char *str, int num, FILE *stream);

The integer parameter in fgets( ) is used to indicate that at most num-1 characters are to be read, terminating at end-of-file or end-of-line. The end-of-line character will be placed in the string str before the string terminator, if it is read. If end-of-file is encountered as the first character, EOF is returned, otherwise str is returned. The fputs( ) function returns a non-negative number or EOF if unsuccessful.

Example:

```
/*Program to read a file and count the number of lines in the file */
#include <stdio.h>
#include <conio.h>
#include <process.h>
void main()
{
FILE *fp;
int cnt=0;
char str[80];
/* open a file in read mode */
if ((fp=fopen("lines.dat","r"))== NULL)
```

```
{
printf("File does not exist\n");
exit(0);
}
/* read the file till the end of the file is encountered */
while(!(feof(fp)))
{ fgets(str,80,fp);        /*reads at most 80 characters in str */
cnt++;                     /* increment the counter after reading a line
*/
}
}
/* print the number of lines */
printf("The number of lines in the file is :%d\n",cnt);
fclose(fp);
}
```

14.6 BLOCK INPUT/OUTPUT FUNCTIONS

Block Input / Output functions read/write a block from to a file. A block can be a record, a set of records or an array. These functions are also defined in standard library.

- fread( )
- fwrite( )

These two functions allow reading and writing of blocks of data.
The syntax is:

```
int fread(void *buf, int num_bytes, int count, FILE *fp);
int fwrite(void *buf, int num_bytes, int count, FILE *fp);
```

In case of fread(), buf is the pointer to a memory area that receives

the data from the file, and in fwrite(), it is the pointer to the information to be written to the file. These functions are quite helpful in the case of binary files. Generally, these functions are used to read or write an array of records from or to a file.

```
/* Program to illustrate the fread() and fwrite() functions*/
#include <stdio.h>
#include <conio.h>
#include <process.h>
#include <string.h>
void main()
{
struct stud
{
char name[30];
int age;
int roll_no;
}
s[30],st;
int i;
FILE *fp;
/*opening the file in write mode*/
if((fp=fopen("sud.dat","w"))== NULL)
{
printf("Error while creating a file\n");
exit(0);
}
/* reading an array of students */
for(i=0;i<30:i++)
scanf("%s %d %d",s[i].name,s[i].age,s[i].roll_no);
/* writing to a file*/
fwrite(s,sizeof(struct stud),30,fp);
```

```
fclose(fp);
/* opening a file in read mode */
fp=fopen("stud.dat","r");
/* reading from a file and writing on the screen */
while(!feof(fp))
{
fread(&st,sizeof(struct stud),1,fp);
fprintf("%s %d %d",st.name,st.age,st.roll_no);
}
fclose(fp);
}
```

Example:

Give the output of the following code fragment:

```
#include <stdio.h>
#include <process.h>
#include <conio.h>
main()
{
FILE * fp1, * fp2;
double a,b,c;
fp1=fopen("file1", "w");
fp2=fopen("file2", "w");
fprintf(fp1,"1 5.34 –4E02");
fprintf(fp2,"-2\n1.245\n3.234e02\n");
fclose(fp1);
fclose(fp2);
fp1=fopen("file1", "r");
fp2=fopen("file2","r");
fscanf(fp1,"%lf %lf %lf",&a,&b,&c);
printf("%10lf %10lf %10lf",a,b,c);
fscanf(fp2,"%lf %lf %lf",&a,&b,&c);
printf("%10.1e %10lf %10lf",a,b,c);
```

```
fclose(fp1);
fclose(fp2);
}
```

14.7 SEQUENTIAL Vs RANDOM ACCESS FILES

C supports two type of files – text and binary files, also two types of file systems – buffered and unbuffered file system. It can also differentiate in terms of the type of file access as Sequential access files and random access files. Sequential access files allow reading the data from the file in sequential manner which means that data can only be read in sequence.

Example: We have considered till now in this unit are performing sequential access. Random access files allow reading data from any location in the file. To achieve this purpose, C defines a set of functions to manipulate the position of the file pointer.

14.8 POINTIONING THE FILE POINTER

To random access files, C requires a function with the help of which the file pointer can be positioned at any random location in the file. Such a function defined in the standard library is:

The function fseek( ) is used to set the file position. Its prototype is:

int fseek(FILE *fp, long offset, int pos);

The first argument is the pointer to a file. The second argument is the number of bytes to move the file pointer, counting from zero.

This argument can be positive, negative or zero depending on the desired movement. The third parameter is a flag indicating from where in the file to compute the offset.

It can have three values:

SEEK_SET(or value 0)	the beginning of the file,
SEEK_CUR(or value 1)	the current position and
SEEK_END(or value 2)	the end of the file

These three constants are defined in <sstdio.h>. If successful fseek( ) returns zero. Another function, rewind() is used to reset the file position to the beginning of the file.

Its prototype is:

```
void rewind(FILE *fp);
fseek(fp,0,SEEK_SET);
```

Another function ftell() is used to tell the position of the file pointer.

Its prototype is:

```
long ftell(FILE *fp);
```

It returns 1 on error and the position of the file pointer if successful.

14.9 THE UNBUFFERED I/O

The buffered I/O system uses buffered input and output, that is, the operating system handles the details of data retrieval and storage, the system stores data temporarily (buffers it) in order to optimize file system access. The buffered I/O functions are handled directly as system calls without buffering by the operating system. That is why they are also known as low-level functions. This is referred to

as an unbuffered I/O system because the programmer must provide and maintain all disk buffers; the routines do not do it automatically.

The low-level functions are defined in the header file <io.h>.

These functions do not use a file pointer of type FILE to access a particular file, but they use the file descriptors, as explained earlier, of type integer. They are also called handles.

a) Opening and closing of files

The function used to open a file is open( ). Its prototype is:

int open(char *filename, int mode, int access);

Here, mode indicates one of the following macros in <fcntl.h>.

Mode:

O_RDONLY	Read only
O_WRONLY	Write only
O_RDWR	Read / Write

The access parameter is used in a UNIX environment for providing access to particular users and is just included here for compatibility and can be set to zero. open() function returns –1 on failure.

```
int fd;
if ((fd=open(filename,mode,0)) == -1)
{
printf("cannot open file\n");
exit(1);
}
```

If the file does not exist, open() the function will not create it. The function create() is used, which will create new files and rewrite

old ones. The prototype is:

```
int creat(char *filename, int access);
```

It returns a file descriptor; if successful else it returns –1. It is not an error to create an already existing file, the function will just truncate its length to zero. The access parameter is used to provide permissions to the users in the UNIX environment. The function close() is used to close a file.

The prototype is:

```
int close(int fd);
```

It returns zero if successful and –1 if not.

b). Reading, Writing and Positioning in File

The functions read() and write() are used to read from and write to a file. The prototypes is:

```
int read(int fd, void *buf, int size);
int write(int fd, void *buf, int size);
```

The first parameter is the file descriptor returned by open(), the second parameter holds the data which must be typecast to the format needed by the program, the third parameter indicates the number of bytes to transferred. The return value tells how many bytes are actually transferred. If this value is –1, then an error must have occurred.

```
/* Program to copy one file to another file to illustrate the functions*/
# include <stdio.h>
# include <io.h>
#include <process.h>
typedef char arr[80];
```

```
typedef char name[30];
main()
{
arr buf;
name fname, sname;
int fd1,fd2,size;
/* check for the command line arguments */
if (argc!=3)
{
printf("Invalid number of arguments\n");
exit(0);
}
if ((fd1=open(argv[1],O_RDONLY))0)
{
printf("Error in opening file %s \n",argv[1]);
exit(0);
}
if ((fd2=creat(argv[2],0))<0)
printf("Error in opening file %s \n",argv[2]);
exit(0);
}
open(argv[2],O_WRONLY);
size=read(fd1,buf,80);                    /* read till end of file */
while (size>0)
{
write(fd2,buf,80);
size=read(fd1,buf,80);
}
close(fd1);
close(fd2);
}
```

14.10 CONCLUSION

In this unit, we have learnt about files and how C handles them. We have discussed the buffered as well as unbuffered file systems. The available functions in the standard library have been discussed. This unit provided you with an ample set of programs to start with. We have also tried to differentiate between sequential access as well as random access files.

The file pointers assigned to standard input, standard output and standard error are stdin, stdout, and stderr, respectively. The unit clearly explains the different file types of modes oof opening the file. As seen there are several functions available to read/write from the file. The usage of a particular function depends on the application. After reading this unit one must be able to handle large data bases in the form of files.