COURSE GUIDE

CIT 301
STRUCTURED PROGRAMMING

Course Team

Dr. Moses E. Ekpenyong (Course Writer)
- University of Uyo

NATIONAL OPEN UNIVERSITY OF NIGERIA

National Open University of Nigeria
Headquarters
University Village
Plot 91, Cadastral Zone Nnamdi Azikiwe Expressway
Jabi, Abuja

Lagos Office
14/16 Ahmadu Bello Way
Victoria Island, Lagos

e-mail: centralinfo@nou.edu.ng
URL: www.nou.edu.ng

Printed 2022

ISBN:

Course Guide

Introduction

CIT 301: Structure Programming is a 3-credit unit course for students studying towards acquiring a Bachelor of Science in Computer Science and other related disciplines.

The course is divided into 8 modules and 20 study units. It will first provide an overview of programming languages and their types and explain the principles of abstraction and modularity. The elements of structured programming is then given before outlining the steps in program design and execution. An introduction to the C programming language follows with how to use and apply operators and control statements. Functions and arrays in C are then discussed, finally ending the course with a study of structures and pointers in C.

Course Objectives

It is important to note that each unit has specific objectives. Students should study them carefully before proceeding to subsequent units. Therefore, it may be useful to refer to these objectives in the course of your study of the unit to assess your progress. You should always look at the unit objectives after completing a unit. In this way, you can be sure that you have done what is required of you by the end of the unit.

However, below are overall objectives of this course. On completing this course, you should be able to:

- Define programs and classify programming languages.
- State the advantages and disadvantages of high and low–level languages.
- State at least six characteristics of a good program.
- List the various phases of program development.
- Explain the steps involved in problem definition and analysis.
- Describe functions and Procedures in program.
- Illustrate a typical function structure for sorting numerical arrays in C language.
- Define a class in Object–oriented programming.
- Explain the abstraction costs and benefits.
- Define modularity in programming.

- State the advantages of modular programming approach.
- Describe the real-life application of modularity concept.
- Explain the importance of header file, module implementation and main program in C programming language.
- Illustrate the structure of interface file in C language declarations.
- State the purpose of IMPORT and the EXTERN macros in C language program declaration.
- State the advantages and disadvantages of structured programming.
- Describe the examples of programming paradigms.
- State the various programming paradigms.
- Describe each of the named programming paradigms and programming language sassociated with it.
- Give a brief history of C Programming Language.
- Explain the taxonomy of C programming types.
- Explain the importance of studying C programming language.
- Describe the characteristics and uses of C Program.
- Illustrate the structure of a C Programming language.
- Explain the contents of the C program structure.
- Explain the processes involved in compilation and execution of C program.
- Describe the sample input/ output steps used in program compilation and execution.
- List the character set in C.
- Apply the character set in constructing variables and identifiers.
- Differentiate between a variable and a keyword.
- Explain what a data type is.
- Define a constant.
- State the rules for constructing integer constants.
- State the rules for constructing real constants.
- State the rules for constructing real constants expressed in exponential form.
- State the rules for constructing character constants.

- Format your input.
- Format your output.
- Differentiate between the output of integer float.
- Differentiate between the input of integer float.
- Define an operator.
- Use operators in expressions.
- Mention the various operators applicable to C programming.
- Describe each of the operators.
- State the three control structures inherent in C.
- State the generic syntax for the various structures.
- Use these structures to write a program code or a block of code.
- Describe each of the structure.
- Manually simulate a program code involving the structures.
- Differentiate between monolithic Vs modular Programming.
- State the disadvantages of monolithic Programming.
- State the advantages of modular Programming.
- Declare a function.
- Outline the various function categories.
- Differentiate between a user define functions vs standard function.
- Differentiate between call by value and call by reference.
- Describe an array.
- Differentiate between one-dimensional and a two-dimensional arrays.
- Initialize one-dimensional, two-dimensional and multi-dimensional arrays.
- State the syntax of array declaration.
- Define a string.
- Differentiate between a string and a character.
- Manipulate string.
- Mention some commonly used string input/output library functions.
- Read and write string.
- Declare a string variable.
- Understand C structures and pointers.

- Know how to define and use structures and pointers in C.

Working Through this Course

To complete this course, you are required to study all the units, the recommended textbooks, and other relevant materials. Each unit contains some self-assessment exercises and tutor marked assignments, and at some point, in this course, you are required to submit the tutor-marked assignments. There is also a final examination at the end of this course. Stated below are the components of this course and what you have to do.

Study Units

MODULE 1: PROGRAMING LANGUAGES
Unit 1:Computer Programming
Unit 2: Characteristics of a Good Program
Unit 3: Phases of Program Development (Programming)

MODULE 2: ABSTRACTION AND MODULARITY
Unit 1: Introduction to Abstraction
Unit 2: Modular Programming
Unit 3: Modular Interface

MODULE 3: ELEMENTS OF STRUCTURED PROGRAMMING
Unit 1: Overview of Structured Programming
Unit 2: Programming Language Paradigms

MODULE 4: Structured Programming with C
Unit 1: Overview of C
Unit 2: C Program Design
Unit 3: Executing a C Program

MODULE 5: Introduction to C Programming Language
Unit1: Element of C
Unit2: Data Type
Unit3: Variables, Statements, Expressions

MODULE 6: OPERATORS AND CONTROL STATEMENTS
Unit 1: Operators
Unit 2: Overview of Control Statements

MODULE 7: FUNCTIONS AND ARRAYS IN C PROGRAMMING LANGUAGE
Unit 1: Overview of Functions in C
Unit 2: Arrays
Unit 3: Fundamentals of Strings

MODULE 8: STRUCTURE AND POINTERS IN C
Unit 1: Structure and Pointers

References and Further Readings

Every study unit contain list of references and further readings. Do not hesitate to consult them if need be.

Presentation Schedule

The Presentation Schedule included in your course material gives you important dates for the completion of Tutor Marked Assignments and tutorial attendance. Remember, you are required to submit all your assignments by the due date. You should guard against falling behind in your work.

Assessment

There are two aspects to the assessment of this course. First, there are tutor marked assignments; and second, the written examination. Therefore, you are expected to take note of the facts, information and problem solving gathered during the course. The tutor

marked assignments must be submitted to your tutor for formal assessment, in accordance to the deadline given. The work submitted will count for 30% of your total course mark. At the end of the course, you will need to sit for a final written examination. This examination will account for 70% of your total score.

How to Get the Most from the Course

In distance learning, the study units replace the university lectures. This is one of the great advantages of distance learning; you can read and work through specially designed study materials at your own pace, and at a time and place that suits you best. Think of it as reading the lecture instead of listening to the lecturer. In the same way a lecturer might give you some reading to do, the study units tell you when to read, and which are your text materials or set books. You are provided exercises to do at appropriate points, just as a lecturer might give you an in-class exercise.

Each of the study units follows a common format. The first item is an introduction to the subject matter of the unit, and how a particular unit is integrated with the other units and the course as a whole. Next to this is a set of learning objectives. These objectives let you know what you should be able to do by the time you have completed the unit. These learning objectives are meant to guide your study. The moment a unit is finished, you must go back and check whether you have achieved the objectives. If you make this a habit, then you will significantly improve your chances of passing the course. The main body of the unit guides you through the required reading from other sources. This will usually be either from your set books or from a reading section. The following is a practical strategy for working through this course. If you run into any trouble, telephone your tutor. Remember that your tutor‘s job is to help you. When you need assistance, do not hesitate to call and ask your tutor to provide it.

In addition, do the following:

1. Read this Course Guide thoroughly, it is your first assignment.
2. Organize a Study Schedule. Design a ―Course Overview‖ to guide you through the Course. Note the time you are expected to spend on each unit and how the assignments relate to the units. Important information, e.g., details of your tutorials, and the date of the first day of the semester is available from the study

centre. You need to gather all the information into one place, such as your diary or a wall calendar. Decide on a method and write in your own dates and schedule of work for each unit.

3. Once you have created your own study schedule, do everything to stay faithful to it. The major reason students fail is that they get behind with their course work. If you get into difficulty with your schedule, please, let your tutor know before it is tool late for help.
4. Turn to Unit 1 and read the introduction and the objectives for the unit.
5. Assemble the study materials. You will need your set books and the unit you are studying at any point in time.
6. Work through the unit. As you work through it, you will know what sources to consult for further information.
7. Keep in touch with your study centre as up-to-date course information will be continuously available there.
8. Well before the relevant due dates (about 4 weeks before due dates), keep in mind that you will learn a lot by doing the assignments carefully. They have been designed to help you meet the objectives of the course and therefore will help you pass the examination. Submit all assignments not later than the due date.
9. Review the objectives for each study unit to confirm that you have achieved them. If you fee unsure about any of the objectives, review the study materials or consult your tutor.
10. When you are confident that you have achieved a unit's objectives, you can start on the next unit. Proceed unit by unit through the course and try to pace your study so that you keep yourself on schedule.
11. When you have submitted an assignment to your tutor for marking, do not wait for its return before starting on the next unit. Keep to your schedule. When the assignment is returned, pay particular attention to your tutor's comments, both on the tutor-marked assignment form and on the ordinary assignments.
12. After completing the last unit, review the course and prepare yourself for the final examination. Check that you have achieved the unit objectives (listed at the beginning of each unit) and the course objectives (listed in the Course Guide).
13. Finally, ensure that you practice on the personal computer as prescribed to gain the maximum proficiency required.

Facilitation

The dates, times and locations of these Tutorials will be made available to you, together with the name, telephone number and address of your Tutor. Each assignment will be marked by your tutor. Pay close attention to the comments your tutor might make on your assignments as these will help in your progress. Make sure that assignments reach your tutor on or before the due date.

Your tutorials are important; therefore, try not to skip any. It is an opportunity to meet your tutor and your fellow students. It is also an opportunity to get the help of your tutor and discuss any difficulties you might encounter when reading.

Course Information

Course Code: CIT 301

Course Title: STRUCTURED PROGRAMMING

Credit Unit: 3

Course Status:

Course Blurb:

Semester:

Course Duration: Required Hours for Study

Course Team

Course Developer:

Course Writer: Dr. Moses E. Ekpenyong
Department of Computer Science
University of Uyo
Uyo

Content Editor:

Instructional Designer:

Learning Technologists:

Copy Editor

CONTENTS **PAGE**

MODULE 1: PROGRAMING LANGUAGES……12
Unit 1: Computer Programming ……...12
Unit 2: Program Design ……17
MODULE 2: ABSTRACTION AND MODULARITY ……20
Unit 1: Introduction to Abstraction ……..20
Unit 2: Modular Programming ……..27
Unit 3: Modular Design ……31
MODULE 3: ELEMENTS OF STRUCTURED PROGRAMMING ……43
Unit 1: Overview of Structured Programming ……..43
Unit 2: Programming Language Paradigms ……..46
MODULE 4: Structured Programming with C ……49
Unit 1: Overview of C ……49
Unit 2: C Program Design ……53
Unit 3: Executing a C Program ……..57
MODULE 5: Introduction to C Programming Language ……60
Unit 1: Element of C ……..60
Unit 2: Data Type ……..63
Unit 3: Variables, Statements, Expressions …… 67
Unit 4: Formatted Input-Output ……76
MODULE 6: OPERATORS AND CONTROL STATEMENTS ……..80
Unit 1: Operators ……..80
Unit 2: Overview of Control Statements ……..87
MODULE 7: FUNCTIONS AND ARRAYS IN C PROGRAMMING LANGUAGE ……118
Unit 1: Overview of Functions in C ……118
Unit 2: Arrays ……..131
Unit 3: Fundamentals of Strings ……153
MODULE 8: STRUCTURE AND POINTERS IN C ……163
Unit 1: Structure and Pointers ……163

MODULE 1: PROGRAMING LANGUAGES

Module Introduction

This module has three (3) units

Unit 1: Computer Programming

Unit 2: Characteristics of a Good Program

Unit 3: Phases of Program Development (Programming)

UNIT 1: COMPUTER PROGRAMMING

CONTENTS

1.0 Introduction

2.0 Intended Learning Outcomes (ILOs)

3.0 Main Contents

- 3.1 Classification of Programming Languages
- 3.2 Low Level Language
- 3.3 High Level Language
- 3.4 Features of High-Level Language

4.0 Self Study Exercise

5.0 Further Reading

1.0 Introduction

A program is a finite set of sequenced instructions or commands given to a computer in order to carry out a particular task. To write a program for the computer to carry out these instructions, there must be a means of communication. Humans communicate via natural languages such as English, French, Chinese etc. Likewise, to communicate with the computer, we also use languages known as programming languages. Programming is the art of program writing using a particular programming language. It can also be said to be the process of writing a set of instructions in sequential manner using programming language to control the activity of a computer system. A computer programming is an

artificial language which is used in writing a set of instructions to control the activities of a computer system.

There are two main types of computer programming languages; they are: Low-level language and High-level language. Machine language makes fast and efficient use of the computer. It requires no translator to translate the code. It is directly understood by the computer. On the contrary, the person writing program in High-level language does not need to know anything about the computer in which the program will be run (Machine Independent), programs are portable and very easy to learn and write. Examples of High-level Languages are FORTRAN, COBOL, QBASIC, VISUAL BASIC, JAVA, ADA, and PASCAL etc.

2.0 Intended Learning Outcomes (ILOs)

By the end of the unit, you will be able to:

- define programs and classify programming languages
- state the advantages and disadvantages of high and low–level languages.

3.0 Main Contents

3.1 Classification of Programming Languages

There are two main types of computer programming languages; they are: Low-level language and High-level language.

3.2. Low level language

This type of language is closer to the machine compared with the human natural language. The two major examples are the Machine language and the Assembly language.

Machine Language: This is the only language computer understands. It is the native language of the computer. The computer directly executes a program written in machine language. These programs are coded using strings of 0's and 1's. It doesn't need a translator.

Advantages of Machine Language

- Machine language makes fast and efficient use of the computer.
- It requires no translator to translate the code. It is directly understood by the computer.

Disadvantages of Machine Language

- Very bulky.
- They require much time for writing and reading.
- They are prone to error which is difficult to be detected and corrected.
- Very difficult learn.
- Can only run on the computer it is designed for meaning that it is machine dependent

Assembly Language: Assembly Language uses MNEMONICS (symbols) to represent data and instructions. Such program eliminates problems associated with machine language. Computer cannot execute directly a program written in assembly language, it requires a translator called assembler. Assembler is a special program designed to translate a program written in assembly language to a machine language equivalent.

Advantages of Assembly Language

- It allows complex jobs to run in a simpler way.
- It is memory efficient, as it requires less memory.
- It is faster in speed, as its execution time is less.
- It is mainly hardware oriented.
- It requires less instruction to get the result.
- It is used for critical jobs.

Disadvantages of Assembly Language

- It is machine dependent; the programmer must be knowledgeable in both subject area and the operations of the machine.
- It is cumbersome though less cumbersome than that of machine language.
- Very expensive to develop
- It consumes time

3.3. High Level Language

A high-level language is a problem orientated programming language, whereas a low-level language is machine oriented. The source programs are written in human readable languages like English instead of mere symbols. In other words, a high-level language is a convenient and simple means of describing the information structures and sequences of actions required to perform a particular task.

Advantages of High-Level Language

- The person writing the program does not need to know anything about the computer in which the program will be run (Machine Independent)
- The programs are portable
- Very easy to learn and write

Disadvantages of High-Level Language

- It takes additional translation times to translate the source to machine code.
- High level programs are comparatively slower than low level programs.
- Compared to low level programs, they are generally less memory efficient.
- Cannot communicate directly with the hardware

3.4. Features of High-Level Language

- Machine independent
- Problem oriented
- Ability to clearly reflect the structure of program written in it.
- Readability
- Programs are portable.

Examples of High level Languages are FORTRAN, COBOL, QBASIC, VISUAL BASIC,JAVA , ADA, and PASCAL etc.

4.0 Self-study Exercise

- Classify programming languages and explain the different categories

5.0 Further Readings

Fundamentals of Structured Programming, Lubia Vinhas, 2016
Structured Programming with C++ by Kjell Backman, 2012

UNIT 2: PROGRAM DESIGN

CONTENTS

1.0 Introduction
2.0 Intended Learning Outcomes (ILOs)
3.0 Main Contents
    3.1 Characteristics of a Good Program
    3.2 Phases of Program Development (Programming)
4.0 Self Study Exercise
5.0 Further Reading

1.0 INTRODUCTION

The principles of data processing set the pace for obtaining the requirements of a good program. In data processing, three phases are critical: The input, processing and the output phases. Input constitutes what instruction and data goes into the system. Processing has to do with what logic or tools are required to manipulate the data. Hence, we expect certain characteristics from a good program or tool intended to process our data to yield informed output.

There are various phases in the development of computer programs. These phases must be strictly adhered to, to ensure a reliable, efficient program devoid of syntax and semantic errors.

2.0 INTENDED LEARNING OUTCOMES (ILOs)

By the end of the unit, you will be able to:

- state at least six characteristics of a good program
- list the various phases of program development.
- explain the steps involved in problem definition and analysis.

3.0 MAIN CONTENT

3.1 Characteristics of a Good Program

- Transferability- Must be able to work on any computer machine.
- Reliability- It can be relied upon to do what it is expected to do.
- Efficiency/cost saving- It must not cost more than its benefits and enables problem to be solved appropriately, quickly and efficiently.
- Simplicity- It should be as simple as possible to understand.
- Understandability/Readability- It must be readable and understandable by other programmers and end users.
- Flexibility/Adaptability / Maintainability- A good program must be flexible adaptable and maintainable in order to suit user's need. Modification must be possible and very easy.

3.2 Phases of Program Development (Programming)

The process of producing a computer program (software) may be divided into eight phases or stages:

1) Problem definition/Analysis
2) Selection or development of an algorithm
3) Designing the program
4) Coding the programming statements
5) Compiling/Compilation stage
6) Testing/Running and Debugging the program
7) Documentation.
8) Maintenance

1) *Problem Definition/Analysis Stage:* There is need to understand the problem that requires a solution. The need to determine the data to be processed, from of the data, volume of the data, what to be done to the data to produce the expected / required output.

2) *Selection or development of an algorithm:* An algorithm is the set of steps required to solve a problem written down in English language.
3) *Designing the program:* In order to minimize the amount of time to be spent in developing the software, the programmer makes use of flowchart. Flowchart is the pictorial representation of the algorithm developed in step 2 above. Pseudocode IPO chart (input processing output) and Hipo chart (Hierachical- input-processing and output) may be used in place of flowchart or to supplement flowchart.
4) *Coding the statement:* This involves writing the program statements. The programmer uses the program flow chart as a guide for coding the steps the computer will follow.
5) *Compiling:* There is need to translate the program from the source code to the machine or object code if it is not written in machine language. A computer program is fed into the computer first, then as the source program is entered, a translated equivalent (object program) is created and stored in the memory.
6) *Running, Testing and Debugging:* When the computer is activated to run a program, it may find it difficult to run it because many syntax errors might have been committed. Manuals are used to debug the errors. A program that is error free is tested using some test data. If the program works as intended, real required data are then loaded.
7) *Documentation:* This is the last stage in software development. This involves keeping written records that describe the program, explain its purposes, define the amount, types and sources of input data required to run it. List the Departments and people who use its output and trace the logic the program follows.

4.0 Self-study Exercise

- Explain any five characteristics of a good program.
- Explain the phases or stages of program (or software) development.
- Describe the content of program documentation.

5.0 Further Readings

Fundamentals of Structured Programming, Lubia Vinhas, 2016.
Structured Programming with C++ by Kjell Backman, 2012.

MODULE 2: ABSTRACTION AND MODULARITY

Module Introduction

This module has three (3) units

Unit 1: Introduction to Abstraction

Unit 2: Modular Programming

Unit 3: Modular Interface

UNIT 1: INTRODUCTION TO ABSTRACTION

CONTENTS

1.0 Introduction

2.0 Intended Learning Outcomes

3.0 Main Contents

- 3.1 Abstraction
- 3.2 Functions and Procedures
- 3.3 Classes
- 3.4 Abstraction Costs and Benefits

4.0 Self Study Exercises

5.0 Further Reading

1.0 Introduction

As programmers began to write instructions that were equivalent to a few bytes, the level of thinking in terms of what the computer was doing on a functional level raised the level of abstraction. Statements and structured code can be thought of as assembly language operations, at a higher level of abstraction. Statements are collected to form functions, procedures, subroutines, or methods. The abstraction of grouping code and its data structures is called object-oriented programming. However, the clump of code and data definitions is called a class in most programming languages. In many software maintenance projects, the cost of the additional performance of low levels of abstraction is far higher than the cost of the computer cycles that would be required to run the program.

2.0 Intended Learning Outcomes (ILOs)

By the end of the unit, you will be able to:

- describe functions and Procedures in program
- illustrate a typical function structure for sorting numerical arrays in C language.
- define a class in Object–oriented programming
- explain the abstraction costs and benefits.

3.0 MAIN CONTENT

3.1 Abstraction

The history of programming has experienced rising levels of granularity. Decades ago, programmers manipulated individual bits of codes. Then the assembly language was invented, and programmers began to write instructions that were equivalent to a few bytes. The advantage was clear: Instead of thinking in terms of essentially meaningless 1s and 0s, you could think in terms of what the computer was doing on a functional level—move this value to that memory location, multiply these two bytes together.

This is called raising the level of abstraction. Every time you raise the level of abstraction in a programming language, you get more program (as measured in terms of bits) for less work. The language at which you communicate with the computer can also be altered into something closer to the way we communicate in English.

Each unit of the level of abstraction has a contract or agreement: The language makes an exact promise of what the computer will do when the unit is executed. For the following assembly language instruction:

```
LD (BC),A
```

the language promises that it will move the value from the register named A into the place in memory pointed to by registers B and C. Obviously, this is only a very small piece of what you want the computer to do, such as word processing, video processing, etc. but it's a lot clearer and easier to use than its binary equivalent:

```
00000010
```

It may not seem any shorter or easier to remember LD (BC), A, but each of the letters here has an explicit and easily remembered meaning: LD is short for LOAD; A, B, and C refer to some registers, and (BC) refers to a way to do indirection into memory. 00000010 may be just seven 0s and a 1, but the order is both critical and hard to memorize. Swapping two of the bits to 00000100 means INC B (increment the B register), which is totally different.

3.2. Functions and Procedures

Statements and structured code can be thought of as assembly language operations, at a higher level of abstraction. The next level of abstraction is to group statements into operational units with contracts of their own. Statements are collected to form functions, procedures, subroutines, or methods, as they are called in various languages. The beauty about functions is that they limit even further the amount of code required to understand a piece of code.

A typical function structure for sorting a numerical array in C programming language, is given below:

```
/** Takes the array and returns a sorted version,
 * removing duplicates. It may return the same array, or it
 * may allocate a new one. If there are no duplicates, it'll
 * probably return the old array. If there are, it'll have to
 * create a new one. */
int[] sort(int[] array)
{
        ... the body ...
}
```

You can learn a lot about the function without even seeing the body. The name sort, and the fact that it takes an array of integers and returns a (possibly different) array of integers, tell you a lot about what the function is supposed to do. The rest of the contract is described in the comment, which talks about other things such as memory allocation. That’s even more important in C and C++ than in Java, where it’s often up to the contract to express who’s responsible for freeing memory allocated in the function.

The maintenance programmer's life is made simpler because the program is chopped up into these functional units. A common rule is that a function should be only as long as a screenful of code. That makes it possible to visualize, all at once, a complete, nameable, understandable unit of the program you're maintaining. That rule turns out to be a little bit too strict, though.

The names of functions and procedures are a critical part of the abstraction. It takes a chunk of code and allows you to refer to it later with a single word (or a short collection of words, strungTogetherLikeThis or _like_this). This strategy focuses every line in the function on achieving the goal with that name. Once the scope of your function grows beyond the name you've assigned to it, it's time to consider breaking the function into pieces with better names. If you find yourself writing code like this:

```
void sortNamesAndSendEmail()
{
        // Sort names
        ... Spend 100 lines sorting the names ...
        // Send email
  .     .. Spend 500 lines sending out emails ...
}
```

it's a good indicator that it's time to start breaking the function into pieces. In effect, you'll probably write two functions:

```
sortNames()
sendEmail()
```

which allows you to eliminate the verbose and weird function name sortNamesAndSendEmail.

3.3. Classes

Structured programming and functions neatly solve some of the problems of maintenance by limiting the amount of code you must look at in order to understand any given line. There's still one way that far-off pieces of code can affect a particular line of code, however.

The sort example given earlier sorts only integers, which is not a particularly interesting job. Why would you ever want to sort just a list of numbers? More likely, you want to be able to sort a list of objects of some kind, based on an integer key. Or, more generally, you'd like to be able to sort on any key, so long as you can reliably compare any two objects.

Even before object-oriented programming, there were ways to group chunks of data into functional units. In C, these units are called structs. However, structs don't have any reliable way to compare them. You need some level of abstraction a little higher than provided by structs that allows you to tell which of two structs should come first. The abstraction of grouping code and its data structures is called object-oriented programming. The clump of code and data definitions is called a class in most programming languages.

C++ is an object-oriented language. It provides a higher level of abstraction than C does. In general, higher levels of abstraction come at a performance penalty, and many people criticize C++ for its performance cost relative to C++.

Java aims for an even higher level of abstraction than C++ by abstracting away access to locations in memory. Though not the first language to do so (Lisp and Basic leap readily to mind, among general-purpose programming languages), it probably has the highest market penetration.

And that level of abstraction also costs performance most of the time. Not always, of course. An advantage to abstraction is that the intermediary translators are allowed to make any optimizations they want, so long as they don't violate the contracts. The larger the program, the harder it is to perform all the optimizations and still make the schedule.

The longer a language has been around, the more tricks the compiler writers learn for optimization. Increasingly, languages at higher levels of granularity perform faster than those at lower levels. There's no way you could write a large program for a Pentium processor and make it as efficient as the same program written in C; the pipeline stalls would suck up all of your performance gains (even if you knew what they were).

3.4. Abstraction Costs and Benefits

In many software maintenance projects, the cost of the additional performance of low levels of abstraction is far higher than the cost of the computer cycles that would be required to run the program.

As a maintenance programmer, your time is extremely expensive. The time of your users is even more expensive (since there are usually more of them than there are of you), so correctness of the program is key. If users lose work or time waiting for your software to be corrected, that easily represents lots of money.

Higher levels of abstraction lead to improved maintenance, simply because there's less code. The less code, the less you have to read to understand it. Certainly, there are limits to this, as 50 lines of clear codeis preferable to 10 lines of total obscurity. In general, however, by using higher levels of abstraction, improved maintainability is gained.

Of course, there's a downside to these higher levels of abstraction in terms of performance. The more flexible a program is, the harder it is to optimize. As a maintainer, you'll have to find the balance that works best. The old dictum of C.A.R. Hoare that "Premature optimization is the root of all evil" is particularly applicable to abstraction. Choose your levels appropriately and optimize those parts that can't be made to function at the level of abstraction you choose. The payoff is in programming time, both in development and maintenance, and that makes users happy.

4.0 Self Study Exercises

- What is Abstraction?
- Write a function to select the largest of 10 numbers

- Define a class to sort a list of n numbers

5.0 Further Readings

Fundamentals of Structured Programming, Lubia Vinhas, 2016.
Structured Programming with C++ by Kjell Backman, 2012.

UNIT 2: MODULAR PROGRAMMING

CONTENTS

1.0 Introduction
2.0 Intended Learning Outcomes
3.0 Main Contents
    3.1 Modularity
    3.2 Advantages of Using Modular Programming Approach
    3.3 Real-life Example of Modules
    3.4 Modular Programming in C
4.0 Self Study Exercises
5.0 Further Readings

1.0 Introduction

A module is basically a set of interrelated files that share their implementation details but hide it from the outside world. The main advantages of modular programming approach, includes ease of use, reusability, and ease of maintenance. Modularity is applicable in real-life such as electrical devices that can plug into any outlet/socket. Modularization is a method to organize large programs in smaller parts, i.e., the modules. Every module has a well-defined interface toward client modules that specifies how "services" provided by this module are made available. Moreover, every module has an implementation part that hides the code, and any other private implementation details the clients' module.

2.0 Intended Learning Outcomes (ILOs)

By the end of the unit, you will be able to:

- define modularity in programming
- state the advantages of modular programming approach
- describe the real-life application of modularity concept.

3.0 Main Content

3.1. Modularity

Modular programming is the process of subdividing a computer program into separate sub-programs. A module is a separate software component. It can be used in a variety of applications and functions with other components of the system.

Some programs may have thousands or millions of lines and to manage such programs it becomes quite difficult as there might be too many of syntax errors or logical errors present in the program, so to manage such programs,the concept of modular programming is essential. The modular programming concept permits that each sub-module contains something necessary to execute only one aspect of the desired functionality. Modular programming therefore places emphasis on breaking of large programs into small problems to increase the maintainability, readability of the code and to make the program handy for any changes in future.

3.2 Advantages of Using Modular Programming Approach

Ease of Use: This approach allows simplicity, as lines of program code can be accessed in the form of modules, rather than focusing on the entire thousands and millions of lines code. This allows ease in debugging the code and prone to less error.

Reusability: It allows the user to reuse the functionality with a different interface without typing the whole program again.

Ease of Maintenance: It helps in less collision at the time of working on modules, helping a team to work with proper collaboration while working on a large application.

3.3. Real-life Example of Modules

Let’s consider a familiar modular system. Consider the electrical devices (microwaves, electric kettles, washers, dryers, etc.) that can plug into any outlet/socket. None of these devices care if they are plugged into the electrical outlet in your house or your neighbor’s house or your office, etc. They are designed to do their specific task and functionality when they are plugged in and when the power is on, regardless the place they are in.

Application modules should follow the same philosophy. Regardless of the application and even regardless of what application they plugged into, they should do their specific task and only their specific task.

Also, in exactly the same way that an electrical device can easily be unplugged from the wall outlet, a code module should be designed in such a way that it can easily be decoupled and removed from your application.

Furthermore, as the removal of one electrical device has no impact on the functionality of other devices that are plugged into your electrical system, the removal of a code module or a series of code modules from your application should not have any effect on the functionality of the other parts of your application.

This decoupling should also have no effect on the application, other than perhaps just losing the specific functionality that was provided by that particular module or group of modules in the application.

3.4. Modular Programming in C

C is called a structured programming language because to solve a large problem, C programming language divides the problem into smaller modules called functions or procedures each of which handles a particular responsibility. The program which solves the entire problem is a collection of such functions.

A module is basically a set of interrelated files that share their implementation details but hide it from the outside world. Each function defined in C by default is globally accessible. This can be achieved by including the header file in which implementation of the function is defined.

Modularization is a method to organize large programs in smaller parts, i.e., the modules. Every module has a well-defined interface toward client modules that specifies how “services” provided by this module are made available. Moreover, every module has an implementation part that hides the code and any other private implementation detail the clients’ modules should not care of.

Modularization has several benefits, especially on large and complex programs:

- modules can be re-used in several projects;
- changing the implementation details of a modules does not require to modify the clients using them as far as the interface does not change;
- faster re-compilation, as only the modules that have been modified are actually re-compiled;
- self-documenting, as the interface specifies all that is required to know to use the module;
- easier debugging, as modules dependencies are clearly specified and every module can be tested separately.

Programming by modules using the C language means splitting every source code into an header file module1.h that specifies how that module talks to the clients, and a corresponding implementation source file module1.c where all the code and the details are hidden. The header contains only declarations of constants, types, global variables and function prototypes that client programs are allowed to see and to use. Every other private item internal to the module must stay inside the code file. We describe in detail the general structure of the interface and the implementation files.

4.0 Self-study Exercise

- State the benefits of modularization on large and complex programs

5.0 Further Readings

Fundamentals of Structured Programming, Lubia Vinhas, 2016
Structured Programming with C++ by Kjell Backman, 2012

UNIT 3: MODULE DESIGN

CONTENTS

1.0 Introduction
2.0 Intended Learning Outcomes
3.0 Main Contents
    3.1 Module Interface
    3.2 Header file: Constant declarations
    3.3 Header file: Type declarations
    3.4 Header file: Global variables
    3.5 Header file: Function prototypes
    3.6 Module implementation
    3.7 Main program
4.0 Self Study Exercise
5.0 Further Readings

1.0 Introduction

Every interface file should start with a brief description of its purpose, author, copyright statement, version and how to check for further updates. The purpose of the module1_IMPORT and the EXTERN macros in C language program declaration is to allow the definition file to be included by client modules and the implementation of the module, so that global public variables can be declared only once, and the compiler can check if the function prototypes do really match their implementation. In the main program declaration, program_name and its source file is program_name.c are necessary. The source module does not require an header file, and it contains only public function, main(), that does not need a prototype. The main source includes and initializes all the required modules, and finally terminates them once the program is finished.

2.0 Intended Learning Outcomes (ILOs)

By the end of the unit, you will be able to:

- Explain the importance of header file, module implementation and main program in C programming language.
- Illustrate the structure of interface file in C language declarations.

- State the purpose of IMPORT and the EXTERN macros in C language program declaration

3.0 Main Contents

3.1 Module interface

Every interface file should start with a brief description of its purpose, author, copyright statement, version and how to check for further updates. All these information are simply C comments.

Proper C declarations must be enclosed between C preprocessor directives that prevent the same declarations from being parsed twice in the same compilation run. Here is the skeleton of our module1.h interface file

```
/**
 * Skeleton example of a C module. Illustrates the general structure of a
 * module's interface.
  */

#ifndef module1_H
#define module1_H

/*
 * System headers required by the following declarations
 * (the implementation will import its specific dependencies):
 */
#include <stdlib.h>
#include <math.h>

/*
 * Application specific headers required by the following declarations
 * (the implementation will import its specific dependencies):
 */
#include "module2.h"
```

```
#include "module3.h"

/* Set EXTERN macro: */
#ifdef module1_IMPORT
   #define EXTERN
#else
   #define EXTERN extern
#endif

/* Constants declarations here. */

/* Types declarations here. */

/* Global variables declarations here. */

/* Function prototypes here. */

#undef module1_IMPORT
#undef EXTERN
#endif
```

As a general rule, to prevent collisions in the global space of names, every public identifier must start with the name of the module, then an underscore, and then the actual name of the item.

The purpose of the module1_IMPORT and the EXTERN macros is to allow the definition file to be included by client modules AND the implementation of the module, so that global public variables can be declared only once, and the compiler can check if the function prototypes do really match their implementation.

And here is the trick. The implementation file module1.c will define the macro module1_IMPORT just before including its own header file; in this way the EXTERN macro is left empty and all the public variables and public functions will result properly

defined: variables will be allocated by the compiler in the text section of the generated object file; function prototypes will be checked against their implementation.

Client modules, instead, do not define the module1_IMPORT macro, then the compiler will see only external variables and external functions the linker will have to resolve.

3.2. Header file: Constant declarations

Constants can either be simple macros or enumerative values. Enumeratives are more suited to define also a new type and are discussed below along the type declarations. Usually, constants are simple int or double numbers, but also float and literal strings are allowed.

```
/* module1.h -- Constants declarations */

#define module1_MAX_BUF_LEN (4*1024)

#define module1_RED_MASK0xff0000
#define module1_GREEN_MASK0x00ff00
#define module1_BLUE_MASK0x0000ff

#define module1_ERROR_FLAG    (1<<0)
#define module1_WARNING_FLAG  (1<<1)
#define module1_NOTICE_FLAG   (1<<2)
```

3.3. Header file: Type declarations

This section of the header file contains enumerative declarations, data structure declarations, explicit type declarations and opaque type declarations. Enumeratives are suitable to declare several constants. struct declarations are suitable to declare data structures whose internal details are exposed to client modules.

To enforce the encapsulation of the implementation details, an opaque data type can be declared instead of an explicit data type. Opaque data types are types whose internal

details are hidden to the client modules; their actual internal structure is fully declared only in the implementation module, so that client modules cannot access their internal details. This opaque declaration follows this general pattern for the .h and the .c files respectively:

Opaque data type	
module.h	**module.c**

`typedef struct module_Type module_Type;`	`struct module_Type {`
	`  int field1;`
	`  int field2;`
	`  ...`
	`};`

Note that two identifiers are defined: one module_Type is an opaque struct, and the other module_Type is a type derived from this struct type. There is no conflict between these two types because they belong to two different symbol tables inside the C compiler.

The drawback of the opaque types is that clients' modules cannot dynamically allocate opaque data structures, nor they can declare arrays or struct fields of such types because their size is known only inside their own implementation; only pointers to such opaque types are allowed:

Clients can't do this:	...but can use pointers:

module1_Type elems[100]; /* ERR */
 	
struct AnotherType {
int field1;
int field2;
module1_Type field3; /* ERR */
};
	module1_Type *elems[100]; /* ok */
	
	struct AnotherType {
	int field1;
	int field2;
	module1_Type *field3; /*
	ok */
	};

Since client modules can deal only with pointers to opaque types, the implementation must then provide every allocation and initialization routine that may be required, whose typical name follows the scheme module_type_alloc() and module_type_free() respectively.

```
/* module1.h -- Types declarations */

        enummodule1_Direction {
        module1_NORTH,
        module1_EAST,
        module1_SOUTH,
        module1_WEST
};

/**
 * Explicit type declaration example.
 */
typedef struct module1_Node
{
        struct module1_Node *left, *right;
```

```
        char * key;
} module1_Node;


/**
 * Alternative opaque declaration of the node above.
 */
typedef struct module1_Nodemodule1_Node;
```

3.4. Header file: Global variables

It is a good rule to avoid public global variables. But if you really need them, here is the recipe to deal with their declaration and initialization. The module1_IMPORT macro is required in order to allocate the variable in the "text" section of the code module. Without this macro every client module would allocate its own copy if the variable, which is not what we expect.

```
/* module1.h -- Global variables declarations */

EXTERN int module1_counter
#ifdef module1_IMPORT
   = -1
#endif
;

EXTERN module1_Node *module1_root;
```

The preprocessor code protects the initial value from being evaluated by client modules, so that the variables are allocated in the code module and here initialized. Client modules will only see an external variable of some type.

Note that global variables are always initialized to zero and pointers are set to NULL, which typically is just the initial safe value programs expect, so there is need to assign an explicit initial value.

3.5. Header file: Function prototypes

All the functions that need to be accessible from client modules must be declared with a prototype. Remember that functions without arguments must have a dummy void formal argument, otherwise the compiler would complain with a quite misleading error message telling the prototype is missing when it looks to be right there!

```
/* module1.h -- Function prototypes */

/**
 * Initializes this module. Should be called in main() once for all.
 */
EXTERN void module1_initialization(void);

/**
 * Releases internal data structures. Should be called in main()
 * before ending the program.
 */
EXTERN void module1_termination(void);

/**
 * Add a node to the root three.
 * @param key Value to add to the tree.
 * @return Allocated node.
 */
EXTERN module1_Node * module1_add(char * key);

/**
 * Releases node from memory.
```

```
 * @param n Node to release.
 */
EXTERN void module1_free(module1_Node * n);
```

3.6. Module implementation

The implementation module module1.c should include the required headers, then it should define the module1_IMPORT macro before including its own header file. By including its own header, the compiler grabs all the constants, types and variables it requires. Moreover, by including its own header file the code file allocates and initialize the global variables declared in the header. Another useful effect of including the header is that prototypes are checked against the actual functions, so that for example if you forgot some argument in the prototype, or if you changed the code missing to update the header, then the compiler will detect the mismatch with a proper error message.

Macros, constants and types declared inside a code file cannot be exported, as them are implicitly always “private”. Global variables for internal use must have the static keyword in order to make them “private”.

Remember also to declare as static all the functions that are private to the code module. The static keyword tells the compiler that these functions are not available for linking, and then them will not be visible anymore once the code file has been compiled in its own module1.o object file.

Since all the private items are no exported, there is no need to prepend the module name module1_ to their name, as them cannot collide with external items. Private items are still available to the debugger, anyway.

```
/* module1.c -- See module1.h for copyright and info */

/* Import system headers and application specific headers: */
#include <malloc.h>
#include <string.h>
#include "module4.h"
```

```
#include "module5.h"

/* Including my own header for checking by compiler: */
#define module1_IMPORT
#include "module1.h"

/* Private macros and constants: */

/* Private types: */

/* Actual declaration of the private opaque struct: */
struct module1_Node
{
        struct module1_Node *left, *right;
        char * key;
};

/* Private global variables: */
static module1_Node * spare_nodes = NULL;
static int allocated_total_size = 0;

/* Private functions: */
static module1_Node * alloc_Node(void){ ... }
static void free_Node(module1_Node * p){ ... }

/* Implementation of the public functions: */
void module1_initialization(void){ ... }
void module1_termination(void){ ... }
module1_Node * module1_add(char * key){ ... }
void module1_free(void){ ... }
```

Note that public functions are left by last, since usually them need some private function; moreover, since public functions already have their prototype, public functions can be

called everywhere in the code above them. The code file should never need to declare function prototypes, the only exception being recursive functions.

3.7. Main program

The name of our project will be program_name and its source file is program_name.c. This source is the only one that does not require an header file, and it contains the only public function, main(), that does not need a prototype. The main source includes and initializes all the required modules, and finally terminates them once the program is finished. The general structure of the main program source file is as follows:

```
/**
 * Our sample program.
 * @copyright information
 * @license as you wish
 * @author: your name
 * @version: date of coding
 * @file
 */

/* Include standard headers: */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

/* Include modules header we directly invoke here: */
#include "module1.h"
#include "module2.h"

int main(int argc, char **argv)
{
        /* Initialize modules: */
        module1_initialization();
        module2_initialization();
```

```
        /* Perform our job. */

        /* Properly terminate the modules, if required: */
        module2_termination();
        module1_termination();

        return 0;
}
```

4.0 Self-study Exercise

Explain header file in C programming language in terms of Constant, Type declarations, global variables and function prototypes.

5.0 Further Readings

Fundamentals of Structured Programming, Lubia Vinhas, 2016
Structured Programming with C++ by Kjell Backman, 2012

MODULE 3: ELEMENTS OF STRUCTURED PROGRAMMING

Unit 1: Overview of Structured Programming
Unit 2: Programming Language Paradigms

UNIT 1: OVERVIEW OF STRUCTURED PROGRAMMING

CONTENTS

1.0 Introduction
2.0 Intended Learning Outcomes (ILOs)
3.0 Main Contents
    3.1 Structured Programming Concept
    3.2 Advantages of Structured Programming
    3.3 Disadvantages of Structured Programming
4.0 Self Study Exercises
5.0 Further Readings

1.0 Introduction

In structured programming design, programs are broken into different functions these functions are also known as modules, subprogram, subroutines and procedures. Structured programming minimizes the chances of the function affecting another. It allows for clearer programs code. It made global variables to disappear and replaced by the local variables.

2.0 Intended Learning Outcomes (ILOs)

By the end of the unit, you will be able to:

- state the advantages and disadvantages of structured programming
- describe the examples of programming paradigms.

3.0 Main Contents

3.1 Structured programming Concept

In structured programming design, programs are broken into different functions these functions are also known as modules, subprogram, subroutines and procedures.

Each function is design to do a specific task with its own data and logic. Information can be passed from one function to another function through parameters. A function can have local data that cannot be accessed outside the function's scope. The result of this process is that all the other different functions are synthesized in another function. This function is known as main function. Many of the high-level languages support structured programming.

Structured programming minimizes the chances of the function affecting another. It allows for clearer programs code. It made global variables to disappear and replaced by the local variables. Due to this change one can save the memory allocation space occupied by the global variable. Its organization helps in the easy understanding of programming logic. So that one can easily understand the logic behind the programs. It also helps the newcomers of any industrial technology company to understand the programs created by their senior workers of the industry. It also made

The languages that support Structured programming approach are:

- C
- C++
- Java
- C#
- Pascal

3.2 Advantages of Structured programming

- It is user friendly and easy to understand.
- Similar to English vocabulary of words and symbols.
- It is easier to learn.
- They require less time to write.

- They are easier to maintain.
- These are mainly problem oriented rather than machine based.
- Program written in a higher-level language can be translated into many machine languages and therefore can run on any computer for which there exists an appropriate translator.
- It is independent of machine on which it is used, i.e., programs developed in high level languages can be run on any computer.

3.3. Disadvantages of Structured Programming

- A high-level language has to be translated into the machine language by translator and thus a price in computer time is paid.
- The object code generated by a translator might be inefficient compared to an equivalent assembly language program.
- Data type are proceeds in many functions in a structured program. When changes occur in those data types, the corresponding change must be made to every location that acts on those data types within the program. This is really a very time-consuming task if the program is very large.
- Let us consider the case of software development in which several programmers work as a team on an application. In a structured program, each programmer is assigned to build a specific set of functions and data types. Since different programmers handle separate functions that have mutually shared data type, other programmers in the team must reflect the changes in data types done by the programmer in data type handled. Otherwise, it requires rewriting several functions.

4.0 Self-study Exercise

Describe how structured programming can lead to programming efficiency.

5.0 Further Readings

Fundamentals of Structured Programming, Lubia Vinhas, 2016
Structured Programming with C++ by Kjell Backman, 2012

UNIT 2: PROGRAMMING PARADIGMS

CONTENTS

1.0 Introduction
2.0 Intended Learning Outcomes (ILOs)
3.0 Main Contents
    3.1 Imperative Paradigm
    3.2 Functional Paradigm
    3.3 Logical Paradigm
    3.4 Object Oriented Paradigm
    3.5 Other Paradigms
4.0 Self Study Exercise
5.0 Further Readings

1.0 Introduction

A programming paradigm, or programming model, is an approach to programming a computer based on a mathematical theory or a coherent set of principles. Examples of these paradigms are imperative, functional, logical, object-oriented paradigms and others.

2.0 Intended Learning Outcomes (ILOs)

By the end of the unit, you will be able to:

- state the various programming paradigms
- describe each of the named programming paradigms and programming languages associated with it.

3.0 Main Content

3.1 Programming paradigms

Solving a programming problem requires choosing the right concepts. All but the smallest toy problems require different sets of concepts for different parts of the program. A programming paradigm, or programming model, is an approach to programming a computer based on a mathematical theory or a coherent set of principles. It is a way of conceptualizing what it means to perform computation and how tasks to be carried out on the computer should be structured and organized. Programming languages are used to

realize programming paradigms. Examples of programming paradigms: imperative, functional, logical, object-oriented. Most popular languages are imperative and use structured programming techniques. Structured programming techniques involve giving the code you write structures, these often involve writing code in blocks such as sequence (code executed line by line), selection (branching statements such as if..then..else, or case) and repetition (iterative statements such as for, while, repeat, loop).

3.2 Imperative paradigm

This paradigm is based on the ideas of a Von Neummann architecture. A command has a measurable effect on the program and the order of commands is important. First do this and next do that. Its main characteristics are incremental change of the program state (variables) as a function of time; execution of commands in an order governed by control structures; and the use of procedures, abstractions of one or more actions, which can be called as a single command. Examples: Fortran, Algol, Basic, C, Pascal.

3.3 Functional paradigm

This paradigm is based on mathematics and theory of functions. The values produced are non-mutable and plays a minor role compared to imperative program. All computations are done by applying functions with no side effects. Functions are firsts class citizens. Evaluate an expression and use the resulting value for something. Example: Haskell, Clojure Check for Functional paradigm

3.4 Logical paradigm

The logic paradigm fits well when applied in problem domains that deals with the extraction of knowledge from basic facts and relations. Is based on axioms, inference rules, and queries. Program execution becomes a systematic search in a set of facts, making use of a set of inference rules. Answer a question via search for a solution. Examples: Prolog and List.

3.5 Object-oriented paradigm

Data as well as operations are encapsulated in objects. Information hiding is used to protect internal properties of an object. Objects interact by means of message passing. In most object-oriented languages objects are grouped in classes and classes are organized

in inheritance hierarchies. Send messages between objects to simulate the temporal evolution of a set of real-world phenomena. Examples: C++, Java.

3.6 Other Paradigms

Other paradigms include Visual paradigm, Constraint based paradigm, Aspect oriented paradigm and Event-oriented paradigm.

4.0 Self-study Exercise

- List the examples of object-oriented programming language
- Differentiate with examples, the difference between functional and object-oriented programming paradigms.

5.0 Further Readings

Fundamentals of Structured Programming, Lubia Vinhas, 2016.
Structured Programming with C++ by Kjell Backman, 2012.

MODULE 4: STRUCTURED PROGRAMMING WITH C

Module Introduction

Unit 1: Overview of C

Unit 2: C Program Design

Unit 3: Executing a C Program

UNIT 1: OVERVIEW OF C

CONTENTS

1.0 Introduction

2.0 Intended Learning Outcomes (ILOs)

3.0 Main Contents

- 3.1 Brief History of C
- 3.2 Taxonomy of C Types
- 3.3 Why Study C?
- 3.4 Why is C Popular?
- 3.5 Characteristics of C program
- 3.6 Uses of C

4.0 Self Study Exercises

5.0 Further Readings

1.0 INTRODUCTION

C programming language is a structure-oriented programming language, developed at Bell Laboratories in 1972 by Dennis Ritchie. It is defined with C Types taxonomy. C language is popular because it is reliable, simple and easy to use among others. It is also characterized with supports for loose typing and extensive use of function calls. The C programming language is used for developing system applications that forms a major portion of operating systems such as Windows, UNIX and Linux.

2.0 Intended Learning Outcomes (ILOs)

By the end of the unit, you will be able to:

- give a brief history of C Programming Language
- explain the taxonomy of C programming types
- explain the importance of studying C programming language
- describe the characteristics and uses of C Program.

3.0 Main Content

3.1 Brief History of C

- The C programming language is a structure-oriented programming language, developed at Bell Laboratories in 1972 by Dennis Ritchie.
- C programming language features were derived from an earlier language called "B" (Basic Combined Programming Language – BCPL)
- C language was invented for implementing UNIX operating system.
- In 1978, Dennis Ritchie and Brian Kernighan published the first edition "The C Programming Language" and is commonly known as K&RC.
- In 1983, the American National Standards Institute (ANSI) established a committee to provide a modern, comprehensive definition of C. The resulting definition, the ANSI standard, or "ANSI C" was completed late 1988.
- Many of C's ideas & principles were derived from the earlier language B, thereby naming this new language "C".

3.2 Taxonomy of C Types

- Scalar types
- Arithmetic types
- Integral types: char, short, int, long
- Floating-point types: float, double, long double
- Pointer types
- Aggregate types
- Array types
- Structure types

- Union types
- Function types
- Void types

3.3 Why is C Popular?

- It is reliable, simple and easy to use.
- C is a small, block-structured programming language.
- C is a portable language, which means that C programs written on one system can be run on other systems with little or no modification.
- C has one of the largest assortments of operators, such as those used for calculations and data comparisons.
- Although the programmer has more freedom with data storage, the languages do not check data type accuracy for the programmer.

3.4 Why Study C?

- By the early 1980s, C was already a dominant language in the minicomputer world of Unix systems. Since then, it has spread to personal computers (microcomputers) and to mainframes.
- Many software houses use C as the preferred language for producing word processing programs, spreadsheets, compilers, and other products.
- C is an extremely flexible language—particularly if it is to be used to write operating systems.
- Unlike most other languages that have only four or five levels of precedence, C has 15.

3.5 Characteristics of a C Program

- Middle level language.
- Small size – has only 32 keywords
- Extensive use of function calls- enables the end user to add their own functions to the C library.
- Supports loose typing – a character can be treated as an integer & vice versa.

- Structured language
- Low level (Bit Wise) programming readily available
- Pointer implementation - extensive use of pointers for memory, array, structures and functions.
- It has high-level constructs.
- It can handle low-level activities.
- It produces efficient programs.
- It can be compiled on a variety of computers

3.6 Uses of C

- The C programming language is used for developing system applications that forms a major portion of operating systems such as Windows, UNIX and Linux. Below are some examples of C being used:
- Database systems
- Graphics packages
- Word processors
- Spreadsheets
- Operating system development
- Compilers and Assemblers
- Network drivers and Interpreters

4.0 Self-study Exercise

- State the uses of C Programming language
- Explain the reasons C language is popular.
- List the application packages that C language is considered useful tools for developments.

5.0 Further Readings

Fundamentals of Structured Programming, Lubia Vinhas, 2016
Structured Programming with C++ by Kjell Backman, 2012

UNIT 2: C PROGRAM DESIGN

CONTENTS

1.0 Introduction
2.0 Intended Learning Outcomes
3.0 Main Contents
    3.1 C Program Structure
    3.2 Files Used in A C Program
4.0 Self Study Exercise
5.0 Further Readings

1.0 Introduction

The structure of a C program is a protocol (rules) to the programmer, which he has to follow while writing a C program. A number of files are used in a C Program. Examples of these files are: source, object, header and executable files.

2.0 Intended Learning Outcomes (ILOs)

By the end of the unit, you will be able to:

- illustrate the structure of a C Programming language
- explain the contents of the C program structure.

3.0 Main Contents

3.1 C Program Structure

The structure of a C program is a protocol (rules) to the programmer, which he has to follow while writing a C program. The general basic structure of C program is shown in the code below. Based on this structure, we can write a C program. Example:

```
/* This program accepts a number and displays it to the user*/
    #include <stdio.h>
    void main(void)
    { int number;
        printf( "Please enter a number: " );
```

```
        scanf( "%d", &number );
        printf( "You entered %d", number );
        return 0;
}
```

Explanation:

- #include: The part of the compiler which actually gets your program from the source file is called the preprocessor.
- *#include <stdio.h>:*#include is a pre-processor directive. It is not really part of our program, but instead it is an instruction to the compiler to make it do something. It tells the C compiler to include the contents of a file (in this case the system file called stdio.h).
- The compiler knows it is a system file, and therefore must be looked for in a special place, by the fact that the filename is enclosed in <> characters
  <stdio.h>: stdio.h is the name of the standard library definition file for all STanDard Input and Output functions.
  The program will almost certainly want to send information to the screen and read things from the keyboard, and stdio.h is the name of the file in which the functions that we want to use are defined.
  The function we want to use is called printf. The actual code of printf will be tied in later by the linker.
  The ".h" portion of the filename is the language extension, which denotes an include file.
- *void:*This literally means that this means nothing. In this case, it is referring to the function whose name follows. Void tells C compiler that a given entity has no meaning and produces no error.
- *main:*In this example, the only function in the program is called main. A C program is typically made up of large number of functions. Each of these is given a name by the programmer and they refer to each other as the program runs. C regards the name main as a special case and will run this function first i.e. the program execution starts from main.
- (void): This is a pair of brackets enclosing the keyword void.

It tells the compiler that the function main has no parameters.

A parameter to a function gives the function something to work on.

- { (Brace): This is a brace (or curly bracket). As the name implies, braces come in packs of two - for every open brace there must be a matching close one. Braces allow us to group pieces of program together, often called a block.A block can contain the declaration of variable used within it, followed by a sequence of program statements.

  In this case the braces enclose the working parts of the function main.
- *; (semicolon)*: The semicolon marks the end of the list of variable names, and also the end of that declaration statement. All statements in C programs are separated by ";" (semicolon) characters. The ";" character is actually very important. It tells the compiler where a given statement ends.
- If the compiler does not find one of these characters where it expects to see one, then it will produce an error.
- *scanf:* In other programming languages, the printing and reading functions are a part of the language. In C this is not the case; instead they are defined as standard functions which are part of the language specification, but are not a part of the language itself.
- The standard input/output library contains a number of functions for formatted data transfer; the two we are going to use are scanf (scan formatted) and printf (print formatted).
- *printf:* The printf function is the opposite of scanf.
  - It takes text and values from within the program and sends it out onto the screen.
  - Just like scanf, it is common to all versions of C and just like scanf, it is described in the system file stdio.h.
  - The first parameter to a printf is the format string, which contains text, value descriptions and formatting instructions.

3.2 Files Used in a C Program

- Source File- This file contains the source code of the program. The file extension of any c file is .c. The file contains C source code that defines the main function & maybe other functions.
- Header File- A header file is a file with extension .h which contains the C function declarations and macro definitions and to be shared between several source files.
- Object File- An object file is a file containing object code, with an extension .o, meaning relocatable format machine code that is usually not directly executable. Object files are produced by an assembler, compiler, or other language translator, and used as input to the linker, which in turn typically generates an executable or library by combining parts of object files.
- Executable File- The binary executable file is generated by the linker. The linker links the various object files to produce a binary file that can be directly executed.

4.0 Self-study Exercise

- Explain all the reserved words used in the description of a C program structure.
- Describe the types files used in a C Program.

5.0 Further Readings

Fundamentals of Structured Programming, Lubia Vinhas, 2016
Structured Programming with C++ by Kjell Backman, 2012

UNIT 3: EXECUTING A C PROGRAM

CONTENTS

1.0 Introduction
2.0 Intended Learning Outcomes
3.0 Main Contents
    3.1 Compilation and Execution of a C Program
    3.2 Commonly used Programs for execution on Linux System
    3.3. Pictorial Diagram of C Compilation and Execution
4.0 Self Study Exercise
5.0 Further Readings

1.0 Introduction

Program compilation and execution processes are divided into several steps, namely: preprocessing, compilation, assembly, linking and loading. In each of these input and output are defined during the compilation and execution process depending on the operating systems e.g., Linux.

2.0 Intended Learning Outcomes (ILOs)

By the end of the unit, you will be able to:

- explain the processes involved in compilation and execution of C program
- describe the sample input/ output steps used in program compilation and execution.

3.0 Main Contents

3.1 Compilation and Execution of a C Program

- The compilation and execution process of C can be divided into several steps:
- Preprocessing - Using a Preprocessor program to convert C source code in expanded source code. "#includes" and "#defines" statements will be processed and replaced source codes in this step.

- Compilation - Using a Compiler program to convert C expanded source to assembly source code.
- Assembly - Using an Assembler program to convert assembly source code to object code.
- Linking - Using a Linker program to convert object code to executable code. Multiple units of object codes are linked to together in this step.
- Loading - Using a Loader program to load the executable code into CPU for execution.

Sample I/O steps

Here is a simple table showing input and output of each step in the compilation and execution process:

Input	Program	Output
source code	> Preprocessor	> expanded source code
expanded source code	> Compiler	> assembly source code
assembly code	> Assembler	> object code
object code	> Linker	> executable code
executable code	> Loader	> execution

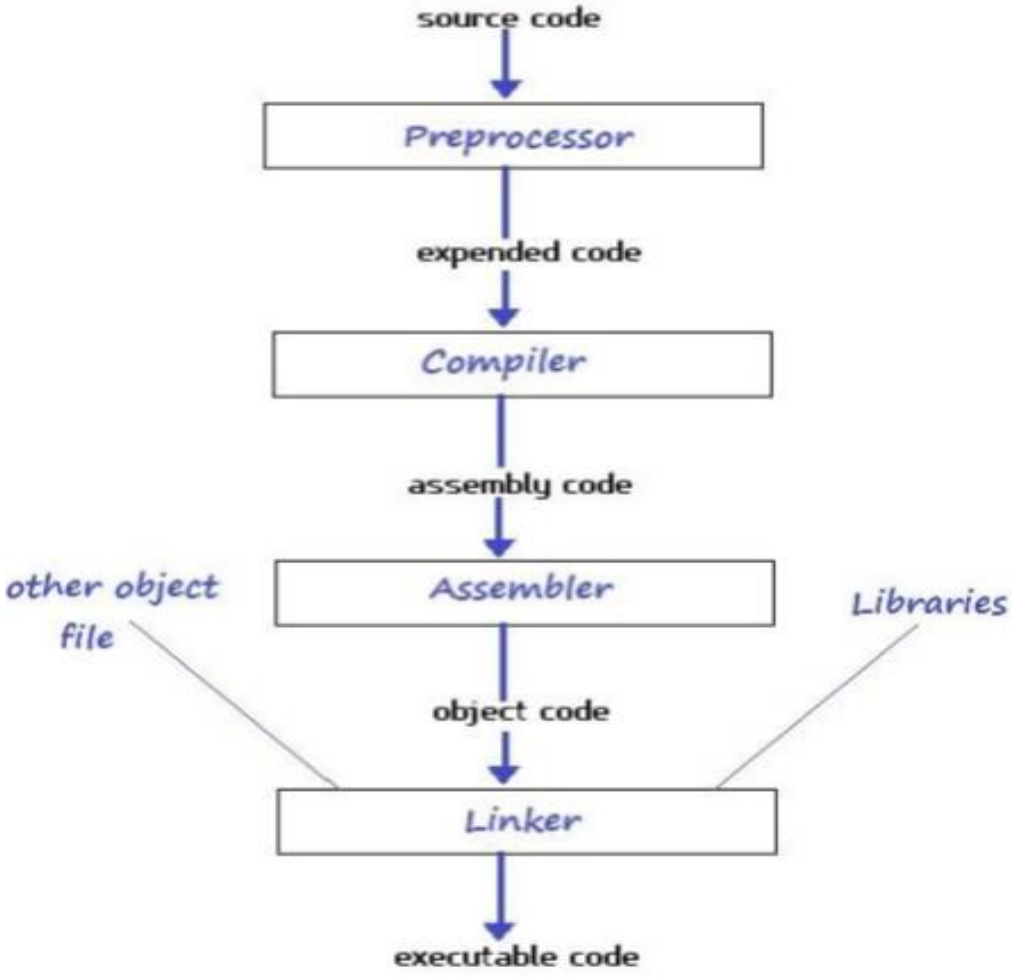
3.2 Commonly used Programs for execution on Linux System

Below are examples of commonly used programs for different compilation and execution steps on a Linux system:

- "cpphello.c -o hello.i" - Preprocessor preprocessing hello.c and saving output to hello.i.
- "cc1hello.i -o hello.s" - Compiler compiling hello.i and saving output to hello.s.
- "as hello.s -o hello.o" - Assembler assembling hello.s and saving output to hello.o.
- "ldhello.o -o hello" - Linker linking hello.o and saving output to hello.
- "load hello" - Loader loading hello and running hello.

3.3 Pictorial Diagram of C Compilation and Execution

A pictorial diagram showing the compilation and execution of a C program is shown following.



4.0 Self-study Exercise

- Illustrate the compilation and execution of a C program with diagram only.
- State the examples of commonly used programs for different compilation and execution steps on a Linux system

5.0 Further Readings

Fundamentals of Structured Programming, Lubia Vinhas, 2016

Structured Programming with C++ by Kjell Backman, 2012

MODULE 5: INTRODUCTION TO C PROGRAMMING LANGUAGE

This module consists of three units
Unit1: Element of C
Unit2: Data Type
Unit3: Variables, Statements, Expressions

Contents

1.0 Introduction
2.0 Intended Learning Outcomes (ILOs)
3.0 Main Content
3.1 Character Set
3.2 Keywords
3.3 Identifier

1.0 Introduction

Every language has some basic elements and grammatical rules. Before starting with programming, we should be acquainted with the basic elements that build the language.

2.0 Intended Learning Outcomes (ILOs)

By the end of the unit, you will be able to:

- list the character set in C
- apply the character set in constructing variables and identifiers
- differentiate between a variable and a keyword.

3.0 Main Content

Elements of C

Every language has some basic elements and grammatical rules. Before starting with programming, we should be acquainted with the basic elements that build the language.

3.1 Character Set

Communicating with a computer involves speaking the language the computer understands. In C, various characters have been given to communicate.

Character set in C consists of:

Types	Character Set
Lower case	a-z
Upper case	A-Z
Digits	0-9
Special Character	!@#$%^&*
White space	Tab or new lines or space

3.2 Keywords

Keywords are the words whose meaning has already been explained to the C compiler. The keywords cannot be used as variable names because if we do so we are trying to assign a new meaning to the keyword, which is not allowed by the computer. There are only 32 keywords available in C. Below figure gives a list of these keywords for your ready reference.

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

3.3 Identifier:

In the programming language C, an identifier is a combination of alphanumeric characters, the first being a letter of the alphabet or an underline, and the remaining being any letter of the alphabet, any numeric digit, or the underline.

Two rules must be kept in mind when naming identifiers.

- The case of alphabetic characters is significant. Using "INDEX" for a variable is not the same as using "index" and neither of them is the same as using "InDeX" for a variable. All three refer to different variables.
- As C is defined, up to 32 significant characters can be used and will be considered significant by most compilers. If more than 32 are used, they will be ignored by the compiler.

UNIT 2: DATA TYPE

CONTENTS

1.0 Introduction
2.0 Intended Learning Outcomes (ILOs)
3.0 Main Content
3.1 Data Types
3.2 Constants
3.3 Rules for Constructing Integer Constants
3.4 Rules for Constructing Real Constants
3.5 Rules for constructing real constants expressed in exponential form
3.6 Rules for Constructing Character Constants

1.0 Introduction

A data type defines a set of values and the operations that can be defined on those values. Data types are especially important in C programming language All operations are type checked by the compiler for type compatibility. Illegal operations will not be compiled. Thus, strong type checking helps prevent errors and enhances reliability.

2.0 Intended Learning Outcomes (ILOs)

By the end of the unit, you will be able to:

- Explain what a data type is
- Define a constant
- State the rules for constructing integer constants
- State the rules for constructing real constants
- State the rules for constructing real constants expressed in exponential form
- State the rules for constructing character constants

1.0 Main Content

3.1 Data Types

In the C programming language, data types refer to a domain of allowed values & the operations that can be performed on those values. The type of a variable determines how

much space it occupies in storage and how the bit pattern stored is interpreted. There are 4 fundamental data types in C, which are- char, int, float &, double. Char is used to store any single character; int is used to store any integer value, float is used to store any single precision floating point number & double is used to store any double precision floating point number. We can use 2 qualifiers with these basic types to get more types.

There are 2 types of qualifiers

- Sign qualifier- signed & unsigned
- Size qualifier- short & long

The data types in C can be classified as follows:

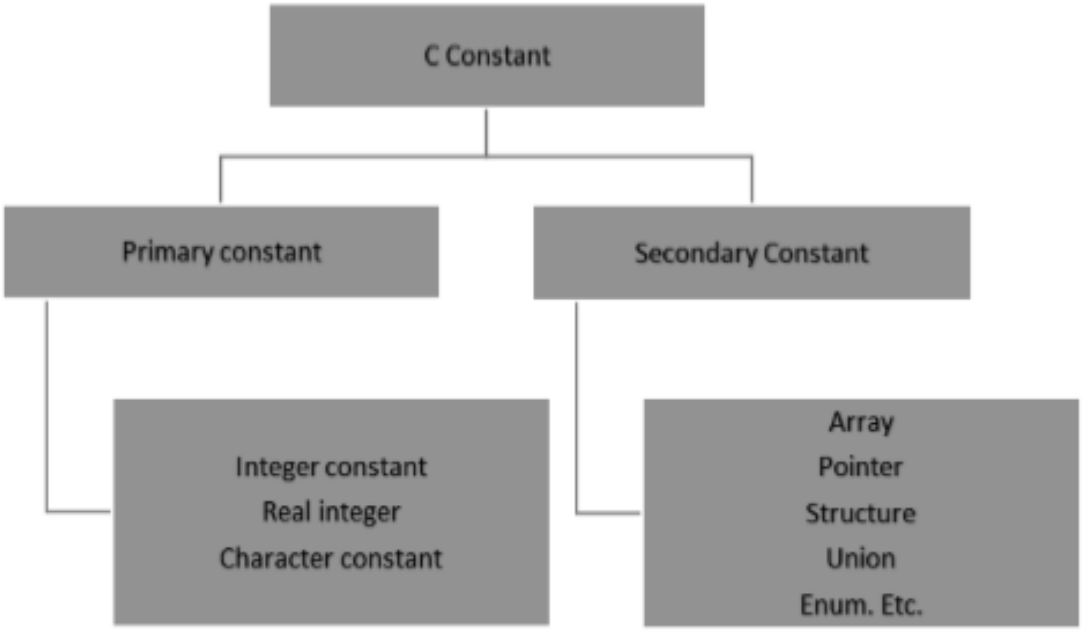
Type	Storage size	Value range
char	1 byte	-128 to 127
unsigned char	1 byte	0 to 255
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

Type	Storage size	Value range	Precision
float	4 bytes	1.2E-38 to 3.4E+38	6 decimal places
double	8 bytes	2.3E-308 to 1.7E+308	15 decimal places
long double	10 bytes	3.4E-4932 to 1.1E+4932	19 decimal places

3.2 Constants

A constant is an entity that doesn't change whereas a variable is an entity that may change. C constants can be divided into two major categories:

- Primary Constants
- Secondary Constants



Here our only focus is on primary constant. For constructing these different types of constants certain rules have been laid down.

3.3 Rules for Constructing Integer Constants

- An integer constant must have at least one digit.
  - It must not have a decimal point.
  - It can be either positive or negative.
  - If no sign precedes an integer constant it is assumed to be positive.
  - No commas or blanks are allowed within an integer constant.
  - The allowable range for integer constants is -32768to 32767.
  - Eg.: 426, +782,-8000, -7605

3.4 Rules for Constructing Real Constants

- Real constants are often called Floating Point constants.
- The real constants could be written in two forms—Fractional form and Exponential form.
- Rules for constructing real constants expressed in fractional form:
    - A real constant must have at least one digit.
    - It must have a decimal point.
    - It could be either positive or negative.
    - Default sign is positive.
    - No commas or blanks are allowed within a real constant.
    - Ex. +325.34, 426.0, -32.76, -48.5792

3.5 Rules for constructing real constants expressed in exponential form

- The mantissa part and the exponential part should be separated by a letter e. b) The mantissa part may have a positive or negative sign.
- Default sign of mantissa part is positive.
- The exponent must have at least one digit, which must be a positive or negative integer. Default sign is positive.
- Range of real constants expressed in exponential form is -3.4e38 to 3.4e38.
- Ex. +3.2e-5, 4.1e8, -0.2e+3, -3.2e-5

3.6 Rules for Constructing Character Constants

- A character constant is a single alphabet, a single digit or a single special symbol enclosed within single inverted commas.
- The maximum length of a character constant can be 1 character.
- Ex.: ‘M’, ‘6’, ‘+’

UNIT 3: VARIABLES, STATEMENTS, EXPRESSIONS

CONTENTS

1.0 Introduction

2.0 Intended Learning Outcomes (ILOs)

3.0 Main Content

3.1 Variables and Variable Declaration

3.2 Initialization of Variables

3.3 Expressions

3.4 Statements

3.5 Compound Statements (Blocks)

3.6 Input-Output in C

3.7 Input-Output of integers in C

3.8 Input-Output of floats in C

3.9 Input-Output of characters and ASCII code

3.10 ASCII code

1.0 Introduction

A variable is the name given to a memory location that allows values to be stored in those locations. When declaring variables, the type of value or data to be stored is also indicated. A statement is an executable instruction given to the computer to execute. An expression is a combination of operands, operators and constant for the purpose of evaluation

2.0 Intended Learning Outcomes (ILOs)

By the end of the unit, you will be able to:

- explain what a data type is
- define a constant
- state the rules for constructing integer constants
- state the rules for constructing real constants
- state the rules for constructing real constants expressed in exponential form
- state the rules for constructing character constants

3.0 Main Content

3.1 Variables and Variable Declaration

Variables are names that are used to store values. It can take different values but one at a time. A data type is associated with each variable & it decides what values the variable can take. When you decide your program needs another variable, you simply declare (or define) a new variable and C makes sure you get it. You declare all C variables at the top of whatever blocks of code need them. Variable declaration requires that you inform C of the variable's name and data type.

Syntax: datatype variablename;

Eg:

```
int page_no;
char grade;
float salary;
long y;
```

There are two places where you can declare a variable:

- After the opening brace of a block of code (usually at the top of a function)
- Before a function name (such as before main() in the program)

Consider various examples:

Suppose you had to keep track of a person's first, middle, and last initials. Because an initial is obviously a character, it would be prudent to declare three character variables to hold the three initials. In C, you could do that with the following statement:

1.
```
main()
{
    char first, middle, last;
    // Rest of program follows
}
```

2.
```
main()
{
    char first;
    char middle;
```

```
        char last;
        // Rest of program follows
}
```

3.2 Initialization of Variables

When a variable is declared, it contains undefined value commonly known as garbage value. If we want, we can assign some initial value to the variables during the declaration itself. This is called initialization of the variable.

Eg.,

```
int pageno=10;
char grade='A';
float salary= 20000.50;
```

3.3 Expressions

An expression consists of a combination of operators, operands, variables & function calls. An expression can be arithmetic, logical or relational. Here are some expressions:

```
a+b – arithmetic operation
a>b- relational operation
a == b – logical operation
func (a,b) – function call
4+21
a*(b + c/d)/20
q = 5*2
x = ++q % 3
q > 3
```

As you can see, the operands can be constants, variables, or combinations of the two. Some expressions are combinations of smaller expressions, called subexpressions. For example, c/d is a subexpression of the sixth example.

An important property of C is that every C expression has a value. To find the value, you perform the operations in the order dictated by operator precedence.

3.4 Statements

Statements are the primary building blocks of a program. A program is a series of statements with some necessary punctuation. A statement is a complete instruction to the computer. In C, statements are indicated by a semicolon at the end. Therefore

```
legs = 4
```

is just an expression (which could be part of a larger expression), but,

```
legs = 4;
```

is a statement. What makes a complete instruction? First, C considers any expression to be a statement if you append a semicolon. (These are called expression statements.) Therefore, C won't object to lines such as the following:

```
8;
3 + 4;
```

However, these statements do nothing for your program and can't really be considered sensible statements. More typically, statements change values and call functions:

```
x = 25;
++x;
y = sqrt(x);
```

Although a statement (or, at least, a sensible statement) is a complete instruction, not all complete instructions are statements. Consider the following statement:

```
x = 6 + (y = 5);
```

In it, the subexpression y = 5 is a complete instruction, but it is only part of the statement. Because a complete instruction is not necessarily a statement, a semicolon is needed to identify instructions that truly are statements.

3.5 Compound Statements (Blocks)

A compound statement is two or more statements grouped together by enclosing them in braces; it is also called a block. The following while statement contains an example:

```
while (years < 100)
    {
       wisdom = wisdom * 1.05;
    printf("%d %d\n", years, wisdom);
       years = years + 1;
```

}

If any variable is declared inside the block, then it can be declared only at the beginning of the block. The variables that are declared inside a block can be used only within the block.

3.6 Input-Output in C

When we are saying Input that means we feed some data into program. This can be given in the form of file or from command line. C programming language provides a set of built-in functions to read given input and feed it to the program as per requirement.

When we are saying Output that means to display some data on screen, printer or in any file. C programming language provides a set of built-in functions to output the data on the computer screen. Functions printf() and scanf() are the most commonly used to display out and take input respectively.

Let us consider an example:

```
#include <stdio.h>      //This is needed to run printf() function.
 int main()
{ printf("C Programming");  //displays the content inside quotation
   return 0;
}
```

Output:

C Programming

Explanation:

- Every program starts from main() function.
- printf() is a library function to display output which only works if #include<stdio.h>is included at the beginning.
- Here, stdio.h is a header file (standard input output header file) and #include is command to paste the code from the header file when necessary. When compiler

encounters printf()function and doesn't find stdio.h header file, compiler shows error.

- return 0; indicates the successful execution of the program.

3.7 Input-Output of integers in C

```
#include<stdio.h>
int main()
{
        int c=5;
        printf("Number=%d",c);
        return 0;
}
```

Output:

```
Number=5
```

Inside quotation of printf() there, is a conversion format string "%d" (for integer). If this conversion format string matches with remaining argument, i.e, c in this case, value of c is displayed.

```
#include<stdio.h>
int main()
{
        int c;
        printf("Enter a number\n");
        scanf("%d",&c);
        printf("Number=%d",c);
        return 0;
}
```

Output:

```
Enter a number
4
Number=4
```

The scanf() function is used to take input from user. In this program, the user is asked an input and value is stored in variable c. Note the '&' sign before c. &c denotes the address of c and value is stored in that address.

3.8 Input-Output of floats in C

```
#include <stdio.h>
int main()
{
        float a;
        printf("Enter value: ");
        scanf("%f",&a);
        printf("Value=%f",a);    //%f is used for floats instead of %d
        return 0;
}
```

Output

```
Enter value:
23.45
Value=23.450000
```

Conversion format string "%f" is used for floats to take input and to display floating value of a variable.

3.9 Input-Output of characters and ASCII code

```
#include <stdio.h>
int main()
{
        char var1;
        printf("Enter character: ");
        scanf("%c",&var1);
        printf("You entered %c.",var1);
        return 0;
```

```
}
```

Output

```
Enter character:
g
You entered g.
```

Conversion format string "%c" is used in case of characters.

3.10 ASCII code

When character is typed in the above program, the character itself is not recorded a numeric value (ASCII value) is stored. And when we displayed that value by using "%c", that character is displayed.

```
#include <stdio.h>
int main()
{
        char var1;
        printf("Enter character: ");
        scanf("%c",&var1);
        printf("You entered %c.\n",var1);
        /* \n prints the next line(performs work of enter). */
        printf("ASCII value of %d",var1);
        return 0;
}
```

Output:

```
Enter character:
g
103
```

When, 'g' is entered, ASCII value 103 is stored instead of g.

You can display character if you know ASCII code only. This is shown by following example.

```
#include <stdio.h>
int main()
{
        int var1=69;
        printf("Character of ASCII value 69: %c",var1);
        return 0;
}
```

Output

```
Character of ASCII value 69:
E
```

The ASCII value of 'A' is 65, 'B' is 66 and so on to 'Z' is 90. Similarly, ASCII value of 'a' is 97, 'b' is 98 and so on to 'z' is 122.

UNIT 4: FORMATTED INPUT-OUTPUT

CONTENTS

1.0 Introduction
2.0 Intended Learning Outcomes (ILOs)
3.0 Main Content
    3.1 Formatted Input-Output
    3.2 Variations in Output for integer and floats
    3.3 Variations in Input for integer and floats
4.0 Self Study Exercise
5.0 Further Reading

1.0 Introduction

Normally printf() method display output on the screen in an unpleasant and undesirable manner. It therefore implies that the programmer must format the output to suit his requirements. For example, he must specify how many places of decimal is required, the space between two outputs, etc.

2.0 Intended Learning Outcomes (ILOs)

By the end of the unit, you will be able to:

- format your input
- format your output
- differentiate between the output of integer float
- differentiate between the input of integer float.

3.0 Main Content

3.1 Formatted Input-Output

Data can be entered & displayed in a particular format. Through format specifications, better presentation of results can be obtained.

3.2 Variations in Output for integer and floats

```
#include<stdio.h>
int main()
	{
		printf("Case 1:%6d\n",9876);
		 /*  Prints the number right justified within 6 columns  */
		printf("Case 2:%3d\n",9876);
		/* Prints the number to be right justified to 3 columns but, there are 4 digits
		so number is not right justified  */
		printf("Case 3:%.2f\n",987.6543);
		/* Prints the number rounded to two decimal places */
		printf("Case 4:%.f\n",987.6543);
		/* Prints the number rounded to 0 decimal place, i.e, rounded to integer */
		printf("Case 5:%e\n",987.6543);
		/* Prints the number in exponential notation (scientific notation) */
		return 0;
	}
```

Output

```
Case 1:  9876
Case 2:9876
Case 3:987.65
Case 4:988
Case 5:9.876543e+002
```

3.3 Variations in Input for integer and floats

```
#include <stdio.h>
int main()
{
	int a,b;
	float c,d;
	printf("Enter two intgers: ");
```

```
        /*Two integers can be taken from user at once as below*/
        scanf("%d%d",&a,&b);
        printf("Enter intger and floating point numbers: ");
        /*Integer and floating point number can be taken at once from user as
        below*/
        scanf("%d%f",&a,&c);
        return 0;
}
```

Similarly, any number of inputs can be taken at once from user.

3.4 Self Study Exercise

1. To print out a and b given below, which of the following printf() statement will you use?

```
#include<stdio.h>
float a=3.14;
double b=3.14;
```

A. printf("%f %lf", a, b);
B. printf("%Lf %f", a, b);
C. printf("%Lf %Lf", a, b);
D. printf("%f %Lf", a, b);

2. To scan a and b given below, which of the following scanf() statement will you use?

```
#include<stdio.h>
float a;
double b;
```

A. scanf("%f %f", &a, &b);
B. scanf("%Lf %Lf", &a, &b);
C. scanf("%f %Lf", &a, &b);
D. scanf("%f %lf", &a, &b);

3. For a typical program, the input is taken using.

A. scanf

B. Files

C. Command-line

D. None of the mentioned

4. What is the output of this C code?

```
#include <stdio.h>
int main()
{
        int i = 10, j = 2;
        printf("%d\n", printf("%d %d ", i, j));  }
```

A. Compile time error

B. 10 2 4

C. 10 2 2

D. 10 2 5

5. What is the output of this C code?

```
#include <stdio.h>
int main()
{
        int i = 10, j = 3;
        printf("%d %d %d", i, j);  }
```

A. Compile time error

B. 10 3

C. 10 3 some garbage value

D. Undefined behavior

5.0 Further Readings

Fundamentals of Structured Programming, Lubia Vinhas, 2016

Structured Programming with C++ by Kjell Backman, 2012

MODULE 6: OPERATORS AND CONTROL STATEMENTS

This module is divided into two units

Unit 1: Operators

Unit 2: Overview of Control Statements

UNIT 1: OPERATORS

CONTENTS

1.0 Introduction
2.0 Intended Learning Outcomes (ILOs)
3.0 Main Content
    3.1 Arithmetic Operators
    3.2 Relational Operators
    3.3 Logical Operators
    3.4 Bitwise Operators
    3.5 Assignment Operators
    3.6 Increment and decrement operators
    3.7 Conditional operators
    3.8 Misc Operators
    3.9 Operators Precedence in C
4.0 Self Study Exercise
5.0 Further Readings

3.0 Introduction

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C language is rich in built-in operators and provides the following types of operators:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators

- Increment and decrement operators
- Conditional operators
- Misc Operators
- Operators Precedence in C

2.0 Intended Learning Outcomes (ILOs)

After studying this unit, you should be able to

- define an operator
- use operators in expressions
- mention the various operators applicable to C programming
- describe each of the operators.

3.0 MAIN CONTENT

3.1 Arithmetic operators

These are used to perform mathematical calculations like addition, subtraction, multiplication, division and modulus.

Following table shows all the arithmetic operators supported by C language. Assume variable A holds 10 and variable B holds 20 then:

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A – B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2

%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increments operator increases integer value by one	A++ will give 11
--	Decrements operator decreases integer value by one	A–will give 9

3.2 Relational Operators:

These operators are used to compare the value of two variables.

Following table shows all the relational operators supported by C language. Assume variable A holds 10 and variable B holds 20, then:

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

3.3 Logical Operators:

These operators are used to perform logical operations on the given two variables.

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 nonzero, then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

3.4 Bitwise Operators

Bitwise operator works on bits and performs bit-by-bit operation. Bitwise operators are used in bit level programming. These operators can operate upon int and char but not on float and double.

Showbits( ) function can be used to display the binary representation of any integer or character value.

Bit wise operators in C language are; & (bitwise AND), | (bitwise OR), ~ (bitwise OR), ^ (XOR), << (left shift) and >> (right shift).

The truth tables for &, |, and ^ are as follows:

p	*q*	p & q	p | q	p ^ q
0	*0*	0	0	0
0	*1*	0	1	1
1	*1*	1	1	0
1	*0*	0	1	1

The Bitwise operators supported by C language are explained in the following table. Assume variable A holds 60 (00111100) and variable B holds 13 (00001101), then:

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12, which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61, which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49, which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61, which is 1100 0011 in 2's complement form.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

3.5 Assignment Operators:

In C programs, values for the variables are assigned using assignment operators.

There are following assignment operators supported by C language:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A

-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C – A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

3.6 Increment and Decrement Operators

In C, ++ and – are called increment and decrement operators respectively. Both of these operators are unary operators, i.e, used on single operand. ++ adds 1 to operand and – subtracts 1 to operand respectively. For example:

```
Let a=5 and b=10
a++;  //a becomes 6
a--; //a becomes 5
++a; //a becomes 6
--a; //a becomes 5
```

When i++ is used as prefix(like: ++var), ++var will increment the value of var and then return it but, if ++ is used as postfix(like: var++), operator will return the value of operand first and then only increment it. This can be demonstrated by an example:

```
#include <stdio.h>
int main()
{
        int c=2,d=2;
        printf("%d\n",c++); //this statement displays 2 then, only c incremented by 1 to 3.
        Printf("%d",++c);     //this statement increments 1 to c then, only c is displayed.
        Return 0;
}
```

Output

2

4

3.7 Conditional Operators (?)

Conditional operators are used in decision making in C programming, i.e., executes different statements according to test condition whether it is either true or false.

Syntax of conditional operators:

conditional_expression?expression1:expression2

If the test condition is true (that is, if its value is non-zero), expression1 is returned and if false expression2 is returned.

Let us understand this with the help of a few examples:

```
int x, y;
scanf ( "%d", &x ) ;
y = ( x> 5 ? 3 : 4 ) ;
```

This statement will store 3 in y if x is greater than 5, otherwise it will store 4 in y.

The equivalent if statement will be,

```
if ( x> 5 )
```

```
        y = 3;
else
        y = 4;
```

3.8 Misc Operators

There are few other operators supported by c language.

Operator	Description	Example
sizeof()	It is a unary operator which is used in finding the size of data type, constant, arrays, structure etc.	sizeof(a), where a is integer, will return 4.
&	Returns the address of a variable.	&a; will give actual address of the variable.
*	Pointer to a variable.	*a; will pointer to a variable.

3.9 Operators Precedence in C

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator.

For example x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

UNIT 2: OVERVIEW OF CONTROL STATEMENTS

CONTENTS

1.0 Introduction

2.0 Intended Learning Outcomes (ILOs)

3.0 Main Content

3.1 Selection Statements

3.1.1 if Statement

3.1.2 else-if Statement

3.1.3 Nested if-else

3.1.4 switch case

3.2 Iterative Statements

3.2.1 while statement

3.2.2 do-while Loop

3.2.3 for Loop

3.2.4 Nesting of Loops

3.3 Jump Statements

3.3.1 The break statement

3.3.2 The continue Statement

3.3.3 The goto statement

4.0 Self-Assessment Exercise(s)

4.0 Introduction

C programming language has basically three control structures which make C qualify as a structured programming language. These structures include sequence, selection and repetition structure. Normally programming are executed sequentially but the last two structures i.e. selection and repetition allow the sequence to be broken. That is the sequence of execution is transferred a different line or block of code. These structures will be examined in more details in this module.

2.0 Intended Learning Outcomes (ILOs)

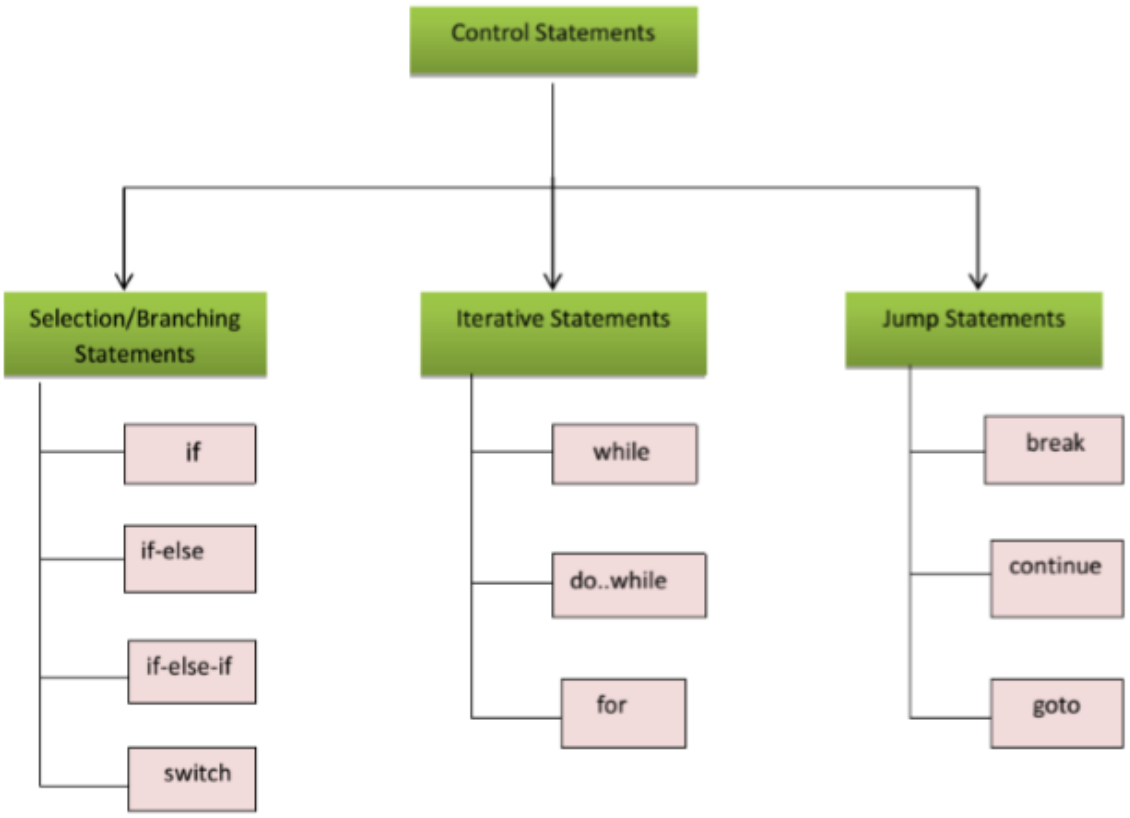
After studying this unit, you should be able to

- State the three control structures inherent in C

- State the generic syntax for the various structures
- Use these structures to write a program code or a block of code
- Describe each of the structure
- Manually simulate a program code involving the structures

3.0 Main Content

In C, programs are executed sequentially in the order of which they appear. This condition does not hold true always. Sometimes a situation may arise where we need to execute a certain part of the program. Also, it may happen that we may want to execute the same part more than once. Control statements enable us to specify the order in which the various instructions in the program are to be executed. They define how the control is transferred to other parts of the program. Control statements are classified in the following ways:



3.1 Selection Statements

The selection statements are also known as Branching or Decision Control Statements. Sometimes we come across situations where we have to make a decision. E.g., If the weather is sunny, I will go out and play, else I will be at home. Here my course of action is governed by the kind of weather. If it's sunny, I can go out and play, else I have to stay indoors. I choose an option out of 2 alternate options. Likewise, we can find ourselves in situations where we have to select among several alternatives. We have decision control statements to implement this logic in computer programming.

The programmer therefore specifies one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

3.1.1 if Statement

The keyword if tells the compiler that what follows is a decision control instruction. The if statement allows us to put some decision -making into our programs. A flowchart illustrating the general form of the if statement is shown below:

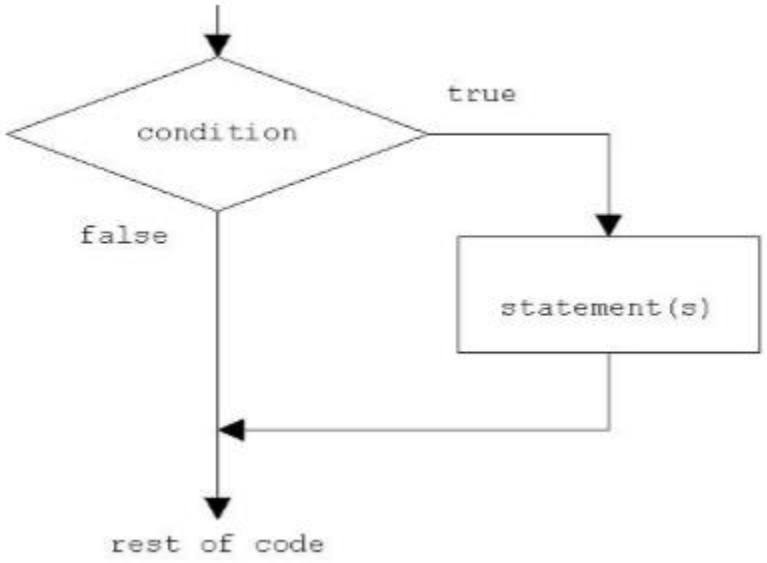


Fig 2: if statement construct

Syntax of if statement:

```
if (condition )
{
        Statement 1;
        Statement 2;
        ……..
        ……..
        ……..
        Statement n;
}
//Rest of the code
```

If the condition is true(nonzero), the statement will be executed. If the condition is false(0), the statement will not be executed. For example, suppose we are writing a billing program.

```
if (total_purchase>=1000)
        printf("You are gifted a pen drive.\n");
```

Multiple statements may be grouped by putting them inside curly braces {}. For example:

```
if (total_purchase>=1000)
{
        gift_count++;
        printf("You are gifted a pen drive.\n");
}
```

For readability, the statements enclosed in {} are usually indented. This allows the programmer to quickly tell which statements are to be conditionally executed. As we will see later, mistakes in indentation can result in programs that are misleading and hard to read.

Programs:

1. Write a program to print a message if negative no is entered.

```
#include<stdio.h>
int main()
{
  int no;
  printf("Enter a no : ");
  scanf("%d", &no);
   if(no<0)
        {
        printf("no entered is negative");
        no = -no;
        }
  printf("value of no is %d \n",no);
  return 0;
}
```

Output:

Enter a no: 6

value of no is 6

Output:

Enter a no: -2

value of no is 2

2. Write a program to perform division of 2 nos

```
#include<stdio.h>
int main()
{
  int a,b;
  float c;
  printf("Enter 2 nos : ");
  scanf("%d %d", &a, &b);
  if(b == 0)
```

```
    {
printf("Division is not possible");
    }
  c = a/b;
  printf("quotient is %f \n",c);
  return 0;
}
```

Output:
Enter 2 nos:
6 2
quotient is 3
Output: Enter 2 nos:
6 0
Division is not possible

3.1.2 if-else Statement

The if statement by itself will execute a single statement, or a group of statements, when the expression following if evaluates to true. By using else we execute another group of statements if the expression evaluates to false.

```
if (a > b)
  {
        z = a;
        printf("value of z is :%d",z);
  }
else
  {
         z = b;
        printf("value of z is :%d",z);
  }
```

The group of statements after the if is called an 'if block'. Similarly, the statements after the else form the 'else block'.

Programs:

3. Write a program to check whether the given no is even or odd

```
#include<stdio.h>
int main()
{
        int n;
        printf("Enter an integer\n");
        scanf("%d",&n);
        if ( n%2 == 0 )
        printf("Even\n");
else
        printf("Odd\n");
        return 0;
}
```

Output: Enter an integer 3

Odd

Output: Enter an integer 4

Even

4. Write a program to check whether a given year is leap year or not

```
#include <stdio.h>
int main()
  {
        int year;
        printf("Enter a year to check if it is a leap year\n");
        scanf("%d", &year);
        if ( (year%4 == 0) && (( year%100 != 0) || ( year%400 == 0 ))
        printf("%d is a leap year.\n", year);
 else
        printf("%d is not a leap year.\n", year);
        return 0;
  }
```

Output: Enter a year to check if it is a leap year 1996
1996 is a leap year
Output: Enter a year to check if it is a leap year 2015
2015 is not a leap year

else-if Statement

This sequence of if statements is the most general way of writing a multi−way decision. The expressions are evaluated in order; if an expression is true, the statement associated with it is executed, and this terminates the whole chain. As always, the code for each statement is either a single statement, or a group of them in braces.

```
if (expression)
        statement
else if (expression)
        statement
else if (expression)
        statement
else if (expression)
        statement
else
        statement
```

The last else part handles the ``none of the above'' or default case where none of the other conditions is satisfied. Sometimes there is no explicit action for the default; in that case the trailing can be omitted, or it may be used for error checking to catch an “impossible” condition.

Program

6. The above program can be used as an e.g., here.

```
#include <stdio.h>
int main()
{
  int m=40,n=20;
  if (m>n)
```

```
        {
        printf("m is greater than n");
        }
else if(m<n)
        {
        printf("m is less than n");
        }
else
        {
        printf("m is equal to n");
         }
}
```

Output: m is greater than n

3.1.3 Nested if-else

An entire if-else construct can be written within either the body of the if statement or the body of an else statement. This is called ‘nesting’ of ifs. This is shown in the following structure.

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

The second if construct is nested in the first if statement. If the condition in the first if statement is true, then the condition in the second if statement is checked. If it is false, then the else statement is executed.

Program:

5. Write a program to check for the relation between 2 nos

```
#include <stdio.h>
int main()
{
  int m=40, n=20;
  if ((m >0 )&& (n>0))
        {
              printf("nos are positive");
              if (m>n)
        {
        printf("m is greater than n");
        }
        else
        {
        printf("m is less than n");
        }
        }
        else
        {
        printf("nos are negative");
        }
        return 0;
}
```

Output

40 is greater than 20

3.1.4 switch case

This structure helps to make a decision from the number of choices. The switch statement is a multi−way decision that tests whether an expression matches one of a number of constant integer values, and branches accordingly.

```
switch( integer expression)
```

```
{
case constant 1 :     do this;
case constant 2 :   do this ;
case constant 3 :   do this ;
default :   do this ;   }
```

The integer expression following the keyword switch is any C expression that will yield an integer value. It could be an integer constant like 1, 2 or 3, or an expression that evaluates to an integer. If a case matches the expression value, execution starts at that case. All case expressions must be different. The case labelled default is executed if none of the other cases are satisfied. A default is optional; if it isn't there and if none of the cases match, no action at all takes place. Cases and the default clause can occur in any order.

Consider the following program:

```
main( )
{
int i = 2;
switch ( i )
  {
        case 1:
        printf ( "I am in case 1 \n" ) ;
        case 2:
        printf ( "I am in case 2 \n" ) ;
        case 3:
        printf ( "I am in case 3 \n" ) ;
        default :
        printf ( "I am in default \n" ) ;
   }
}
```

The output of this program would be:

```
I am in case 2
I am in case 3
```

I am in default

Here the program prints case 2 and 3 and the default case. If you want that only case 2 should get executed, it is up to you to get out of the switch then and there by using a break statement.

```
main( )
{
int i = 2 ; switch ( i )
  {
        case 1:
        printf ( "I am in case 1 \n" ) ;
        break ;
        case 2:
        printf ( "I am in case 2 \n" ) ;
        break ;
        case 3:
        printf ( "I am in case 3 \n" ) ;
        break ;
        default:
        printf ( "I am in default \n" ) ;
  }
}
```

The output of this program would be:

I am in case 2

Program

7. Write a program to enter a grade and check its corresponding remarks.

```
#include <stdio.h>
int main ()
{
char grade;
printf(“Enter the grade”);
```

```
scanf(“%c”, &grade);
switch(grade)
  {
        case 'A' :  printf("Outstanding!\n" );
        break;
        case 'B' :
        printf("Excellent!\n" );
        break;
        case 'C' : printf("Well done\n" );
        break;
        case 'D' : printf("You passed\n" );
        break;
        case 'F' : printf("Better try again\n" );
        break;
        default :  printf("Invalid grade\n" );
  }
printf("Your grade is  %c\n", grade );
return 0;
}
```

Output

```
Enter the grade
B
Excellent
Your grade is B
```

3.2 Iterative Statements

3.2.1 while statement

The while statement is used when the program needs to perform repetitive tasks. The general form of a while statement is:

```
while ( condition)
        statement ;
```

The program will repeatedly execute the statement inside the while until the condition becomes false(0). (If the condition is initially false, the statement will not be executed.)

Consider the following program:

```
main( )
{
int p, t, count;
float r, si;
count = 1;
while ( count<= 3 )
  {
        printf ( "\nEnter values of p, t and r " ) ;
        scanf("%d %d %f", &p, &t, &r ) ;
        si=p * t * r / 100;
        printf ( "Simple interest = N. %f", si ) ;
        count = count+1;
  }
}
```

Some outputs of this program:

Enter values of p, t and r 1000000 5 13.5
Simple Interest = N. 675000.000000
Enter values of p, t and r 2000000 5 13.5
Simple Interest = N. 1350000.000000
Enter values of p, t and r 3500000 5 13.5
Simple Interest = N. 612000.000000

The program executes all statements after the while 3 times. These statements form what is called the 'body' of the while loop. The parentheses after the while contain a condition. As long as this condition remains true all statements within the body of the while loop keep getting executed repeatedly.

Consider the following program;

```
/* This program checks whether a given number is a palindrome or not */
#include <stdio.h>
int main()
{
int n, reverse = 0, temp;
printf("Enter a number to check if it is a palindrome or not\n");
scanf("%d",&n);
temp = n;
while( temp != 0 )
  {
        reverse = reverse * 10;
        reverse = reverse +temp%10;
        temp = temp/10;
  }
        if ( n == reverse )
        printf("%d is a palindrome number.\n", n);
        else
        printf("%d is not a palindrome number.\n", n);
         return 0;
}
```

Output:

```
Enter a number to check if it is a palindrome or not
12321
12321 is a palindrome
Enter a number to check if it is a palindrome or not
12000
12000 is not a palindrome
```

3.2.2 do-while Loop

The body of the do-while executes at least once. The do-while structure is similar to the while loop except the relational test occurs at the bottom (rather than top) of the loop. This ensures that the body of the loop executes at least once. The do-while tests for a

positive relational test; that is, as long as the test is True, the body of the loop continues to execute.

The format of the do-while is

```
do
{
        block of one or more C statements;
}  while (test expression)
```

The test expression must be enclosed within parentheses, just as it does with a while statement.

Consider the following program

```
// C program to add all the numbers entered by a user until user enters 0.
#include <stdio.h>
int main()
{
int sum=0,num;
do
/* Codes inside the body of do...while loops are at least executed once. */
{
        printf("Enter a number\n");
        scanf("%d",&num);
        sum+=num;
} while(num!=0);
printf("sum=%d",sum);
return 0;
}
```

Output:

```
Enter a number
3
Enter a number
-2
```

Enter a number
0
sum=1

Consider the following program:

```
#include <stdio.h>main()
{
        int i = 10;
        do
{
        printf("Hello %d\n", i );
                i = i -1;
}while ( i> 0 );  }
```

Output

Hello 10
Hello 9
Hello 8
Hello 7
Hello 6
Hello 5
Hello 4
Hello 3
Hello 2
Hello 1

Program

8. Program to count the no of digits in a number

```
#include <stdio.h>
int main()
{
        int n,count=0;
        printf("Enter an integer: ");
```

```
        scanf("%d", &n);
        do
        {
                n/=10;
                /* n=n/10 */
                count++;
        } while(n!=0);
        printf("Number of digits: %d",count);
}
```

Output

```
Enter an integer: 34523
Number of digits: 5
```

3.2.3 for Loop

The for is the most popular looping instruction. The general form of for statement is shown below:

```
for ( initialize counter ; test counter ; Updating counter )
        {
                do this;
                and this;
                and this;
        }
```

The for keyword allows us to specify three things about a loop in a single line:

- Setting a loop counter to an initial value.
- Testing the loop counter to determine whether its value has reached the number of repetitions desired.
- Updating the value of loop counter either increment or decrement.

Consider the following program

```
        int main(void)
        {
```

```
        int num;
        printf("    n   n cubed\n");
        for (num = 1; num <= 6; num++)
        printf("%5d %5d\n", num, num*num*num);
        return 0;
}
```

The program prints the integers 1 through 6 and their cubes.

```
 n   n cubed
1     1
2     8
3     27
4     64
5    125
6     216
```

The first line of the for loop tells us immediately all the information about the loop parameters: the starting value of num, the final value of num, and the amount that num increases on each looping

Grammatically, the three components of a for loop are expressions. Any of the three parts can be omitted, although the semicolons must remain.

Consider the following program:

```
main( )
{
int i ;
for ( i = 1 ; i<= 10 ; )
  {
        printf ( "%d\n", i ) ;
        i = i + 1 ;
  }
}
```

Here, the increment is done within the body of the for loop and not in the for statement. Note that in spite of this the semicolon after the condition is necessary.

Programs:

9. Program to print the sum of the first N natural numbers.

```
#include <stdio.h>
int main()
{
int n, i, sum=0;
printf("Enter the limit: ");
scanf("%d", &n);
for(i=1;i<=n;i++)
  {
        sum = sum +i;
  }
printf("Sum of N natural numbers is: %d",sum);
}
```

Output

Enter the limit: 5 Sum of N natural numbers is 15.

10. Program to find the reverse of a number

```
#include<stdio.h>
int main()
{
int num,r,reverse=0;
printf("Enter any number: ");
scanf("%d",&num);
for(num!=0;num=num/10)
{
        r=num%10;
        reverse=reverse*10+r;
}
```

```
printf("Reversed of number: %d",reverse);
return 0;
}
```

Output:

Enter any number: 123

Reversed of number: 321

3.2.4 Nesting of Loops

C programming language allows using one loop inside another loop. Following section shows few examples to illustrate the concept.

Syntax:

The syntax for a nested for loop statement in C is as follows:

```
for ( init; condition; increment )
{
 for ( init; condition; increment)
   {
        statement(s);
    }
statement(s);
}
```

The syntax for a nested while loop statement in C programming language is as follows:

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

The syntax for a nested do...while loop statement in C programming language is as follows:

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

A final note on loop nesting is that you can put any type of loop inside of any other type of loop. For example, a for loop can be inside a while loop or vice versa.

Programs: 11. program using a nested for loop to find the prime numbers from 2 to 20:

```
#include <stdio.h>
int main ()
{
/* local variable definition */
int i, j;
for(i=2; i<20; i++)
{
        for(j=2; j <= (i/j); j++)
                if(!(i%j))
        break;
// if factor found, not prime
if(j > (i/j))
        printf("%d is prime\n", i);
}
return 0;  }
```

Output

2 is prime

3 is prime

5 is prime

7 is prime

11 is prime

13 is prime

17 is prime

19 is prime

Programs: 12. Using for loops reproduce the star triangle below

```
    *
   ***
  *****
 *******
*********
```

```
#include <stdio.h>
int main()
{
int row, c, n,I, temp;
printf("Enter the number of rows in pyramid of stars you wish to see ");
scanf("%d",&n);
temp = n;
   for ( row = 1 ; row <= n ; row++ )
  {
    for ( i= 1 ; i< temp ; i++ )
    {
printf(" ");
        temp--;
         for ( c = 1 ; c <= 2*row - 1 ; c++ )
          {
printf("*");
```

```
printf("\n");
            }
       }
  }
 return 0;
}
```

13. Write a program to print series from 10 to 1 using nested loops.

```
#include<stdio.h>
void main ()
{  int a;
a=10;
   for (k=1;k=10;k++)
   {
     while (a>=1)
     {
printf ("%d",a);  a--;
      } printf("\n");
a= 10;
    }
}
```

Output:

```
10 9 8 7 5 4 3 2 1
10 9 8 7 5 4 3 2 1
10 9 8 7 5 4 3 2 1
10 9 8 7 5 4 3 2 1
10 9 8 7 5 4 3 2 1
10 9 8 7 5 4 3 2 1
10 9 8 7 5 4 3 2 1
10 9 8 7 5 4 3 2 1
10 9 8 7 5 4 3 2 1
```

10 9 8 7 5 4 3 2 1

3.3 Jump Statements

3.3.1 The break Statement

The break statement provides an early exit from for, while, and do, just as from switch. A break causes the innermost enclosing loop or switch to be exited immediately. When break is encountered inside any loop, control automatically passes to the first statement after the loop.

Consider the following example;

```
main( )
{
int i = 1 , j = 1 ;
while ( i++ <= 100 )
   {
  while ( j++<= 200 )
        {
           if ( j == 150 )
        break ;
        else
        printf ( "%d %d\n", i, j );
        }
   }
}
```

In this program when j equals 150, break takes the control outside the inner while only, since it is placed inside the inner while.

3.3.2 The continue Statement

The continue statement is related to break, but less often used; it causes the next iteration of the enclosing for, while, or do loop to begin. In the while and do, this means that the test part is executed immediately; in the for, control passes to the increment step. The continue statement applies only to loops, not to switch.

Consider the following program:

```
main( )
{
int i, j ;
for ( i = 1 ; i<= 2 ; i++ )
   {
      for ( j = 1 ; j <= 2 ; j++ )
      {
        if ( i == j)
continue ;
printf ( "\n%d %d\n", i, j ) ;
      }
   }
}
```

The output of the above program would be...

```
1 2
2 1
```

Note that when the value of I equals that of j, the continue statement takes the control to the for loop (inner) by passing rest of the statements pending execution in the for loop (inner).

3.3.3 The goto statement

Kernighan and Ritchie refer to the goto statement as "infinitely abusable" and suggest that it "be used sparingly, if at all. The goto statement causes your program to jump to a different location, rather than execute the next statement in sequence.

The format of the goto statement is;

```
goto statement label;
```

Consider the following program fragment

```
if (size > 12)
        goto a;
        goto b;
a: cost = cost * 1.05;
flag = 2;
b: bill = cost * flag;
```

Here, if the if conditions satisfies the program jumps to block labelled as a: if not then it jumps to block labelled as b:.

4.0 Self Study Exercises

1. Draw a flowchart and write a for and while loop program to produce the following output: 2 5 8 11 14
2. Draw a flowchart and write a for and do loop program to generate this output: -50 0 50 100 150
3. Develop an algorithm that behaves as follows:
   Prompt the user for an integer, one hundred times. Each time through the loop, show the highest number so far, the lowest number so far, and the average of all input so far.
4. Write an algorithm that behaves as follows:
   Prompt the user for an integer, one thousand times. Then show the user the average of those numbers, the highest number, and the lowest number.
5. Draw a flowchart and write a switch statement algorithm with no more than five cases that behaves as follows.
   Enter a positive integer, or -1 to stop:
   5
   5 is less than 20
   Enter a positive integer, or -1 to stop:
   35
   35 is less than 40

Enter a positive integer, or -1 to stop:
55
55 is less than 60
Enter a positive integer, or -1 to stop:
61
61 is less than 100
Enter a positive integer, or -1 to stop:
85
85 is less than 100
Enter a positive integer, or -1 to stop:
106
106 is equal to or more than 100
Enter a positive integer, or -1 to stop:
-1

6. Write an algorithm that prompts the user for a value, reads the value into a variable, then prints the number out and stops.
7. Write an algorithm that will print “green” for input integers 0-3; “red” for 4-7; “yellow” for 8-11; and “purple” for any other integer. You may not use more than four different cases. Assume the input is already in the declared integer variable “input”.
8. Write a program (WAP) to input the 3 sides of a triangle and print its corresponding type.
9. WAP to input the name of salesman and total sales made by him. Calculate and print the commission earned.

TOTAL SALES	RATE OF COMMISSION
1-1000	3 %
1001-4000	8 %
6001-6000	12 %
6001 and above	15 %

10. WAP to calculate the wages of a labor.

TIME	WAGE

First 10 hrs.	N 60
Next 6 hrs.	N 15
Next 4 hrs.	N 18
Above 10 hrs.	N 25

11. WAP to calculate the area of a triangle, circle, square or rectangle based on the user's choice.
12. WAP that will compute:
    - Vol of a cube
    - Vol of a cuboid
    - Vol of a cylinder
    - Vol of sphere

13. WAP to print the following series
    - S = 1 + 1/2 + 1/3 … 1/10
    - P = (1*2) + (2 *3) + (3* 4)+…+(8 *9) +(9 *10)
    - Q = ½ + ¾ +5/6 +….+13/14
    - S = 2/5 + 5/9 + 8/13….n
    - S = $x + x^2 + x^3 + x^4$.....+ $x^9 + x^{10}$
    - P = x + x3/3 + x5/5 + x7/7……n terms
    - S = (13*1) + (12*2)+…+(1 *13)
    - S = 1 + 1/(2^2) + 1/ (3^3) + 1/(4^4) + 1/(5^5)
    - S = 1/1! + 1/2! + 1/3! +…+1/n!
    - S = 1 + 1/3! + 1/5!+…+n terms
    - S = 1 + (1+2) +(1+2+3) + (1+2+3+4)+…+(1+2+3+…+20)
    - S= $x + x^2/2! + x^3/3! + x^4/4!$+…+$x^{10}/10!$
    - P = x/2! + x^2/3! +…+x^9/10!
    - S = 1-2 + 3-4+…+ 9-10
    - S = 1 -22 + 32 – 42+…+92 - 102
    - S = 1/(1 + 2) + 3/(3 + 5) + …+ 15/(15 + 16)
    - S = 1 +x2/2! – x4/4! + x6/6!...n
    - S = 1 + (1 + 2) + (1+2+3) + … + (1+2+3+4…..20)

- $S = 1 + x + x^2/2 + x^3/3 + \ldots + x^n/n$
- $S = (1 * 3)/ (2 * 4 * 5) + (2 * 4) /(3 * 5 * 6) +(3 * 5)/ (4 * 6 * 7) + \ldots +(n * (n+2))/ ((n+1) * (n+3) * (n+4))$

5.0 Further Readings

- Fundamentals of Structured Programming, Lubia Vinhas, 2016
- Structured Programming with C++ by Kjell Backman, 2012

MODULE 7: FUNCTIONS AND ARRAYS IN C PROGRAMMING LANGUAGE

Module Introduction

This module is divided into three (3) units

Unit 1: Overview of Functions in C

Unit 2: Arrays

Unit 3: Fundamentals of Strings

UNIT 1: OVERVIEW OF FUNCTIONS IN C

CONTENTS

1.0 Introduction

2.0 Intended Learning Outcomes (ILOs)

3.0 Main Content

3.1 Monolithic Vs Modular Programming:

3.2 Function

- 3.2.1 Function Declaration OR Function Prototype:
- 3.2.2 Function Definition:

3.5 User Define Functions Vs Standard Function:

- 3.5.1 User Define Function:
- 3.5.2 Standard Function:

3.6 Function Categories

- 3.6.1 Function with no arguments and no return values:
- 3.6.2 Function with no arguments and a return value:
- 3.6.3 Function with arguments and return value:

3.7 Actual Arguments and Formal Arguments

- 3.7.1 Actual Arguments
- 3.7.2 Formal Arguments
- 3.7.3 Basic difference between formal and local argument
- 3.7.4 Parameter Passing Techniques

4.0 Self-study Exercise

5.0 Further Readings

1.0 Introduction

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing Different programming languages name them differently, for example, functions, methods, sub-routines, procedures, etc. function interface is a declaration of a function that specifies the function's name and type signature (arity, data types of parameters, and return type), but omits the function body.

2.0 Intended Learning Outcomes (ILOs)

By the end of the unit, you will be able to:

- differentiate between monolithic vs modular programming
- state the disadvantages of monolithic programming
- state the advantages of modular programming
- declare a function
- outline the various function categories
- differentiate between a user define functions vs standard function.
- differentiate between call by value and call by reference.

3.0 Main Content

3.1 Monolithic Vs Modular Programming:

- Monolithic Programming indicates the program which contains a single function for the large program.
- Modular programming helps the programmer to divide the whole program into different modules and each module is separately developed and tested. Then the linker will link all these modules to form the complete program.
- On the other hand, monolithic programming will not divide the program and it is a single thread of execution. When the program size increases it leads inconvenience and difficult to maintain.

Disadvantages of monolithic programming:

- Difficult to check error on large programs.
- Difficult to maintain. 3. Code can be specific to a particular problem. i.e., it cannot be reused.

Advantage of modular programming

- Modular programs are easier to code and debug.
- Reduces the programming size.
- Code can be reused in other programs.
- Problem can be isolated to specific module so easier to find the error and correct it.

3.2 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.

3.2.1 Function Declaration OR Function Prototype

1. It is also known as function prototype .
2. It inform the computer about the three things
    a) Name of the function
    b) Number and type of arguments received by the function.
    c) Type of value return by the function

    Syntax :
    return_typefunction_name (type1arg1 , type2arg2);
    OR
return_typefunction_name (type1type2);

3. Calling function need information about called function .If called function is place before calling function then the declaration is not needed.

3.2.2 Function Definition

1. It consists of code description and code of a function .

It consists of two parts

a) Function header

b) Function coding

Function definition tells what are the I/O function and what is going to do.

Syntax:

```
return_typefunction_name (type1arg1 , type2arg2)
  {
     local variable;
statements ;
     return (expression);
  }
```

2. Function definition can be placed any where in the program but generally placed after the main function .

3. Local variable declared inside the function is local to that function. It cannot be used anywhere in the program and its existence is only within the function.

4. Function definition cannot be nested.

5. Return type denote the type of value that function will return and return type is optional if omitted it is assumed to be integer by default.

3.3 User Define Functions Vs Standard Function:

3.3.1 User Define Function:

A function that is declare, calling and define by the user is called user define function. Every user define function has three parts as:

1. Prototype or Declaration
2. Calling
3. Definition

3.3.2 Standard Function:

The C standard library is a standardized collection of header files and library routines used to implement common operations, such as input/output and character string handling. Unlike other languages (such as COBOL, FORTRAN, and PL/I) C does not

include built in keywords for these tasks, so nearly all C programs rely on the standard library to function.

3.4 Function Categories

There are four main categories of the functions these are as follows:

1. Function with no arguments and no return values.
2. Function with no arguments and a return value.
3. Function with arguments and no return values.
4. Function with arguments and return values.

3.4.1 Function with no arguments and no return values:

syntax:

```
void funct (void);
main ( )
{
        funct (  );
}
void funct ( void );
{
}
```

NOTE: There is no communication between calling and called function. Functions are executed independently, they read data and print result in same block.

Example:

```
void  link (void) ;
int  main ()
{  link ();
}
void link ( void );
{
        printf (“ link the file “)
```

```
}
```

3.4.2 Function with no arguments and a return value:

This type of functions has no arguments but a return value

example:

```
 int msg (void) ;
int main ( )
{
        int s = msg (  );
        printf( “summation = %d” , s);
}
int msg ( void )
{
        int a, b, sum ;
        sum = a+b ;
        return (sum) ;
}
```

NOTE: Here called function is independent, it read the value from the keyboard, initialize and return a value .Both calling and called function are partly communicated with each other.

Function with arguments and no return values: Here functions have arguments so, calling function send data to called function but called function does no return value. such functions are partly dependent on calling function and result obtained is utilized by called function .

Example:

```
void  msg ( int , int );
int main ( )
{
        int a,b;
        a= 2;
        b=3;
```

```
        msg( a, b);
}
void msg ( int a , int b)
{
        int s ;
        sum = a+b;
        printf (“sum = %d” , s ) ;
}
```

3.4.3 Function with arguments and return value:

Here calling function of arguments that passed to the called function and called function return value to calling function.

example:

```
int  msg ( int , int ) ;
int main ( )
{
        int a, b;
        a= 2;
        b=3;
        int s = msg (a, b);
        printf (“sum = %d” , s ) ;
}
int msg( int a , int b)
{
        int sum ;
        sum =a+b ;
        return (sum);
}
```

3.5 Actual Arguments and Formal Arguments

3.5.1 Actual Arguments:

1. Arguments which are mentioned in the function call are known as calling function.

2. These are the values which are actual arguments called to the function.
It can be written as constant , function expression on any function call which return a value .
ex: funct (6,9) , funct (a,b)

3.7.2 Formal Arguments:

1. Arguments which are mentioned in function definition are called dummy or formal argument.
2. These arguments are used to just hold the value that is sent by calling function.
3. Formal arguments are like other local variables of the function which are created when function call starts and destroyed when end function.

3.7.3 Basic difference between formal and local argument are:

a) Formal arguments are declared within the ( ) where as local variables are declared at beginning.
b) Formal arguments are automatically initialized when a value of actual argument is passed.
c) Where other local variables are assigned variable through the statement inside the function body.

Note: Order, number and type of actual argument in the function call should be matched with the order, number and type of formal arguments in the function definition.

3.7.4 Parameter Passing Techniques:
1. call by value
2. call by reference

Call by value:
Here value of actual arguments is passed to the formal arguments and operation is done in the formal argument.
Since formal arguments are photo copy of actual argument, any change of the formal arguments does not affect the actual arguments

Changes made to the formal argument t are local to block of called function, so when control back to calling function changes made vanish.

Example:

```
void swap (int a , int b)   /* called function */
        {
                int t;
                t = a;
                a=b;
                b = t;
        }
        main( )
        {
                int k = 50, m= 25;
                swap( k, m) ;    / * calling function */
                print (k, m);     / * calling function */
        }
```

Output:

50, 25

Explanation:

int k= 50, m=25 ;

Means first two memory space are created k and m , store the values 50 and 25 respectively.

swap (k,m);

When this function is calling the control goes to the called function.

void swap (int a , int b),

k and m values are assigned to the ‘a’ and ‘b’.

then a= 50 and b= 25 ,

After that control enters into the function a temporary memory space ‘t’ is created when int t is executed.

t=a; Means the value of a is assigned to the t , then t= 50.

a=b; Here value of b is assigned to the a , then a= 25;
b=t; Again t value is assigned to the b , then b= 50;
after this control again enters into the main function and execute the print function print (k,m). it returns the value 50 , 25.
NOTE: Whatever change made in called function does not affects the values in the calling function.

Call by reference:
Here instead of passing value, address or reference are passed. Function operators or address rather than values .Here formal arguments are the pointers to the actual arguments.

Example:

```
#include<stdio.h>
void add(int *n);
int main()
{
        int num=2;
        printf("\n The value of num before calling the function=%d", num);
        add(&num);
        printf("\n The value of num after calling the function = %d", num);
        return 0;
}
void add(int *n)
{
          *n=*n+10;
        printf("\n The value of num in the called function = %d", n);
}
```

Output:
The value of num before calling the function=2
The value of num in the called function=20 The
value of num after calling the function=20

NOTE:
In call by address mechanism whatever change made in called function affect the values in calling function.

EXAMPLES:
1: Write a function to return larger number between two numbers:

```
int fun(int p, int q)
{
        int large;
        if(p>q)
        {
                large = p;
        }
        else
        {
                large = q;
        }
return large;
}
```

2: Write a program using function to find factorial of a number.

```
#include <stdio.h>
int factorial (int n)
{
        int i, p;
        p = 1;
        for (i=n; i>1; i=i-1)
        {
                p = p * i;
        }
return (p);
}
```

```
void main()
{
        int a, result;
        printf ("Enter an integer number: ");
        scanf ("%d", &a);
        result = factorial (a);
        printf ("The factorial of %d is %d.\n", a, result);
}
```

4.0 Self-study Exercise:

1. What do you mean by function?
2. Why function is used in a program?
3. What do you mean by call by value and call by address?
4. What is the difference between actual arguments and formal arguments?
5. How many types of functions are available in C?
6. How many arguments can be used in a function?
7. Add two numbers using a) with argument with return type b) with argument without return type c) without argument without return type d) without argument with return type
8. Write a program using function to calculate the factorial of a number entered through the keyboard.
9. Write a function power(n,m), to calculate the value of n raised to m.
10. A year is entered by the user through keyboard. Write a function to determine whether the year is a leap year or not.
11. Write a function that inputs a number and prints the multiplication table of that number.
12. Write a program to obtain prime factors of a number. For Example: prime factors of 24 are 2,2,2 and 3 whereas prime factors of 35 are 5 and 7.
13. Write a function which receives a float and a int from main(), finds the product of these two and returns the product which is printed through main().
14. Write a function that receives % integers and returns the sum, average and standard deviation of these numbers. Call this function from main() and print the result in main().
15. Write a function that receives marks obtained by a student in 3 subjects and returns

the average and percentage of these marks. Call this function from main() and print the result in main().

16. Write function to calculate the sum of digits of a number entered by the user.

17. Write a program using function to calculate binary equivalent of a number.

18. Write a C function to evaluate the series sin(x) = x - (x3/3!) + (x5/5!) - (x7/7!) +…… to five significant digit.

19. If three sides of a triangle are p, q and r respectively, then area of triangle is given by area = (S(S-p)(S-q)(S-r))1/2, where S = (p+q+r)/2. Write a program using function to find the area of a triangle using the above formula.

20. Write a function to find GCD and LCM of two numbers.

21. Write a function that takes a number as input and returns product of digits of that number.

22. Write a single function to print both amicable pairs and perfect numbers.(Two different numbers are said to be amicable if the sum of proper divisors of each is equal to the other. 284 and 220 are amicable numbers.)

23. Write a function to find whether a character is alphanumeriLecture 10: Control Statements

5.0 Further Readings

Fundamentals of Structured Programming, Lubia Vinhas, 2016

Structured Programming with C++ by Kjell Backman, 2012

UNIT 2: ARRAYS

CONTENTS

1.0 Introduction
2.0 Intended Learning Outcomes (ILOs)
3.0 Main Content
    3.1 Arrays
        3.1.1 One Dimensional Array
        3.1.2 Declaration One Dimensional Array
        3.1.3 Initialization of One Dimensional Array
        3.1.4 Array Processing
        3.1.5 Two Dimensional Arrays
        3.1.6 Declaration of Two Dimensional Arrays
        3.1.7 Initialization of one Dimensional Array
        3.1.8 Multidimensional Array
        3.1.9 Arrays Using Functions
        3.1.10 Passing Whole 1-D array to a Function
4.0 Self-study Exercise
5.0 Further Readings

1.0 Introduction

In C programming language an array is a collection of similar data items stored at contiguous memory locations and elements can be accessed randomly using indices of an array. They can be used to store collection of primitive data types such as int, float, double, char, etc. of any particular type. Note the items in an array must be of the same data type. In computer science, array programming refers to solutions which allow the application of operations to an entire set of values at once. Such solutions are commonly used in scientific and engineering settings.

2.0 Intended Learning Outcomes (ILOs)

By the end of the unit, you will be able to:

- describe an array
- differentiate between one-dimensional and a two-dimensional arrays

- initialize one-dimensional, two-dimensional and multi-dimensional arrays
- state the syntax of array declaration.

3.0 Main Content

3.1 Arrays

A data structure is the way data is stored in the machine and the functions used to access that data. An easy way to think of a data structure is a collection of related data items. An array is a data structure that is a collection of variables of one type that are accessed through a common name. Each element of an array is given a number by which we can access that element which is called an index. It solves the problem of storing a large number of values and manipulating them.

Previously we use variables to store the values. To use the variables, we have to declare the variable and initialize the variable i.e., assign the value to the variable. Suppose there are 1000 variables are present, so it is a tedious process to declare and initialize each and every variable and also to handle 1000 variables. To overcome this situation, we use the concept of array. In an Array values of same type are stored. An array is a group of memory locations related by the fact that they all have the same name and same type. To refer to a particular location or element in the array we specify the name to the array and position number of particular element in the array.

3.1.1 One Dimensional Array

3.1.2 Declaration one Dimensional Array

Before using the array in the program it must be declared

Syntax:

data_typearray_name[size];

data_type represents the type of elements present in the array.

array_name represents the name of the array.

Size represents the number of elements that can be stored in the array.

Example:

```
int age[100];
```

float sal[15];
char grade[20];

Here age is an integer type array, which can store 100 elements of integer type. The array sal is floating type array of size 15, can hold float values. Grade is a character type array which holds 20 characters.

3.1.3 Initialization of one Dimensional Array

We can explicitly initialize arrays at the time of declaration.

Syntax:
data_typearray_name[size]={value1, value2,……..valueN};

Value1, value2, valueN are the constant values known as initializers, which are assigned to the array elements one after another.

Example:
int marks[5]={10,2,0,23,4};
The values of the array elements after this initialization are:
marks[0]=10, marks[1]=2, marks[2]=0, marks[3]=23, marks[4]=4

NOTE:
1. In 1-D arrays it is optional to specify the size of the array. If size is omitted during initialization then the compiler assumes the size of array equal to the number of initializers.

Example:
int marks[]={10,2,0,23,4};
Here the size of array marks is initialized to 5.
2. We can't copy the elements of one array to another array by simply assigning it.

Example:
int a[5]={9,8,7,6,5};
int b[5];

b=a; //not valid

we have to copy all the elements by using for loop.

```
for(a=i; i<5; i++)
    b[i]=a[i];
```

3.1.4 Array Processing

For processing arrays we mostly use for loop. The total no. of passes is equal to the no. of elements present in the array and in each pass one element is processed.

Example:

```
#include<stdio.h>
main()
{
        int a[3], i;
        for(i=0;i<=2;i++)          //Reading the array values
        {
                printf("enter the elements");
                scanf("%d",&a[i]);
        }
        for(i=0;i<=2;i++)                  //display the array values
        {
                printf("%d",a[i]);
                printf("\n");
        }
}
```

This program reads and displays 3 elements of integer type.

Example:1 C Program to increment every element of the array by one and print incremented array.

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

```
{
        int i;
        int array[4] = {10, 20, 30, 40};
        for (i = 0; i< 4; i++)
                arr[i]++;
                for (i = 0; i< 4; i++)
                        printf("%d\t", array[i]);
        }
```

Example: 2

C Program to print the alternate elements in an array

```
#include <stdio.h>
void main()
{
        int array[10];
        int i, j, temp;
        printf("enter the element of an array \n");
        for (i = 0; i< 10; i++)
        scanf("%d", &array[i]);
        printf("Alternate elements of a given array \n");
        for (i = 0; i< 10; i += 2)
        printf( "%d\n", array[i]) ;
}
```

Example-3

C program to accept N numbers and arrange them in an ascending order

```
#include <stdio.h>
void main()
{
int i, j, a, n, number[30];
printf("Enter the value of N \n");
scanf("%d", &n);
printf("Enter the numbers \n");
```

```
for (i = 0; i< n; ++i)
scanf("%d", &number[i]); for (i = 0; i< n; ++i)
{
for (j = i + 1; j < n; ++j)
        {
         if (number[i] > number[j])
                {
                a =number[i];
                number[i] = number[j];
                number[j] = a;
                }
        }
}
printf("The numbers arranged in ascending order are given below \n");
for (i = 0; i< n; ++i)
printf("%d\n", number[i]);
}
```

3.1.4 Two Dimensional Arrays

Arrays that we have considered up to now are one dimensional array, a single line of elements. Often data come naturally in the form of a table, e.g. spreadsheet, which need a two-dimensional array.

3.1.5 Declaration Two Dimensional Array

The syntax is same as for 1-D array but here 2 subscripts are used.

Syntax:
data_typearray_name[rowsize][columnsize];

Rowsize specifies the no.of rows Columnsize specifies the no.of columns.

Example:
int a[4][5];

This is a 2-D array of 4 rows and 5 columns. Here the first element of the array is a[0][0] and last element of the array is a[3][4] and total no.of elements is 4*5=20.

	col 0	col 1	col 2	col 3	col 4
row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]
row 3	a[3][0]	a[3][1]	a[3][2]	a[3][3]	a[3][4]

3.1.6 Initializationofone Dimensional Array

2-D arrays can be initialized in a way similar to 1-D arrays.

Example:

int m[4][3]={1,2,3,4,5,6,7,8,9,10,11,12};

The values are assigned as follows:

m[0][0]:1 m[0][1]:2 m[0][2]:3
m[1][0]:4 m[1][1]:5 m[3][2]:6

m[2][0]:7 m[2][1]:8 m[3][2]:9

m[3][0]:10 m[3][1]:11 m[3][2]:12

int m[4][3]={{11},{12,13},{14,15,16},{17}};

The values are assigned as:

m[0][0]:1 1 m[0][1]:0 m[0][2]:0

m[1][0]:12 m[1][1]:13 m[3][2]:0

m[2][0]:14 m[2][1]:15 m[3][2]:16

m[3][0]:17 m[3][1]:0 m[3][2]:0

Note:

In 2-D arrays it is optional to specify the first dimension but the second dimension should always be present.

Example:

```
int m[][3]={
                {1,10},
                {2,20,200},
                {3},
                {4,40,400}   };
```

Here the first dimension is taken 4 since there are 4 roes in the initialization list. A 2-D array is known as matrix.

Processing:

For processing of 2-D arrays we need two nested for loops. The outer loop indicates the rows and the inner loop indicates the columns.

Example: int a[4][5];

a) Reading values in a

```
for(i=0;i<4;i++)
for(j=0;j<5;j++)
scanf("%d",&a[i][j]);
```

b) Displaying values of a

```
for(i=0;i<4;i++)
for(j=0;j<5;j++)
printf("%d",a[i][j]);
```

Example 1: Write a C program to find sum of two matrices

```
#include <stdio.h>
#include<conio.h>
void  main()
{
float a[2][2], b[2][2], c[2][2];
int i,j;
```

```
clrscr();
printf("Enter the elements of 1st matrix\n");
/* Reading two dimensional Array with the help of two for loop. If there is an array of 'n' dimension, 'n' numbers of loops are needed for inserting data to array.*/
        for(i=0;i<2;I++)
        for(j=0;j<2;j++)
        {
                scanf("%f",&a[i][j]);
        }
        printf("Enter the elements of 2nd matrix\n");
        for(i=0;i<2;i++)
                for(j=0;j<2;j++)
                {
                        scanf("%f",&b[i][j]);
                }
        /* accessing corresponding elements of two arrays. */
        for(i=0;i<2;i++)
                for(j=0;j<2;j++)
                {
                        c[i][j]=a[i][j]+b[i][j];
                        /* Sum of corresponding elements of two arrays. */
                }
        /* To display matrix sum in order. */
        printf("\nSum Of Matrix:");
        for(i=0;i<2;++i)
        {
                for(j=0;j<2;++j)
                        printf("%f", c[i][j]);
                        printf("\n");
                }
        getch();
        }
```

Example 2: Program for multiplication of two matrices

```
#include<stdio.h>
#include<conio.h>
int main()
{
        int i,j,k;
        int row1,col1,row2,col2,row3,col3;
        int mat1[5][5], mat2[5][5], mat3[5][5];
        clrscr();
        printf("\n enter the number of rows in the first matrix:");
        scanf("%d", &row1);
        printf("\n enter the number of columns in the first matrix:");
        scanf("%d", &col1);
        printf("\n enter the number of rows in the second matrix:");
        scanf("%d", &row2);
        printf("\n enter the number of columns in the second matrix:");
        scanf("%d", &col2);
        if(col1 != row2)
        {
                printf("\n The number of columns in the first matrix must be equal to
                the number of rows   in the second matrix ");
                getch();
                exit();
        }
row3= row1;
col3= col3;
printf("\n Enter the elements of the first matrix");
for(i=0;i<row1;i++)
{
        for(j=0;j<col1;j++)
        scanf("%d",&mat1[i][j]);
}
printf("\n Enter the elements of the second matrix");
```

```
for(i=0;i<row2;i++)
{
        for(j=0;j<col2;j++)
        scanf("%d",&mat2[i][j]);
}
 for(i=0;i<row3;i++)
{
        for(j=0;j<col3;j++)
{
mat3[i][j]=0;
for(k=0;k<col3;k++)
mat3[i][j] +=mat1[i][k]*mat2[k][j];
}
}
printf("\n The elements of the product matrix are"):
for(i=0;i<row3;i++)
{
        printf("\n");
        for(j=0;j<col3;j++)
        printf("\t %d", mat3[i][j]);
}  return 0;
}
```

Output:

```
Enter the number of rows in the first matrix: 2
Enter the number of columns in the first matrix: 2
Enter the number of rows in the second matrix: 2
Enter the number of columns in the second matrix: 2
Enter the elements of the first matrix
1 2 3 4
Enter the elements of the second matrix
5 6 7 8
The elements of the product matrix are
```

19 22
43 50

Example 3: Program to find transpose of a matrix.

```
#include <stdio.h>
int main()
{
        int a[10][10], trans[10][10], r, c, i, j;
        printf("Enter rows and column of matrix: ");
        scanf("%d %d", &r, &c);
        printf("\nEnter elements of matrix:\n");
        for(i=0; i<r; i++)
        for(j=0; j<c; j++)
        {
                printf("Enter elements a%d%d: ",i+1,j+1);          scanf("%d",
                &a[i][j]);      }
                /*  Displaying the matrix a[][] */
                printf("\n Entered Matrix: \n");
        for(i=0; i<r; i++)
                for(j=0; j<c; j++)
                {
                        printf("%d  ",a[i][j]);
                        if(j==c-1)
                                printf("\n\n");
        }
    /* Finding transpose of matrix a[][] and storing it in array trans[][]. */
    for(i=0; i<r;i++)
        for(j=0; j<c; j++)
        {
                trans[j][i]=a[i][j];
        }
    /* Displaying the array trans[][]. */
    printf("\nTranspose of Matrix:\n");
```

```
for(i=0; i<c;i++)      for(j=0; j<r;j++)
{
        printf("%d  ",trans[i][j]);
        if(j==r-1)
        printf("\n\n");
}
return 0;
}
```

Output

```
Enter the rows and columns of matrix:
2   3
Enter the elements of matrix:
Enter elements a11: 1
Enter elements a12: 2
Enter elements a13: 9
Enter elements a21: 0
Enter elements a22: 4
Enter elements a23: 7
Entered matrix:  1 2 9  0 4 7
ranspose of matrix:  1 0  2 4  9 7
```

3.1.8 Multidimensional Array

More than 2-dimensional arrays are treated as multidimensional arrays.

Example:

int a[2][3][4];

Here a represents two 2-dimensional arrays and each of these 2-d arrays contains 3 rows and 4 columns.

The individual elements are:

a[0][0][0], a[0][0][1],a[0][0][2],a[0][1][0]……a[0][3][2]

a[1][0][0],a[1][0][1],a[1][0][2],a[1][1][0]……..a[1][3][2]

the total no. of elements in the above array is 2*3*4=24.

Initialization:

```
int a[2][4][3]={  {
 {1,2,3},
{4,5},
{6,7,8},
{9}
},
{
{10,11},
{12,13,14},
{15,16},
{17,18,19}
}
}
```

The values of elements after this initialization are as:

a[0][0][0]:1	a[0][0][1]:2	a[0][0][2]:3
a[0][1][0]:4	a[0][1][1]:5	a[0][1][2]:0
a[0][2][0]:6	a[0][2][1]:7	a[0][2][2]:8
a[0][3][0]:9	a[0][3][1]:0	a[0][3][2]:0
a[1][0][0]:10	a[1][0][1]:11	a[1][0][2]:0
a[1][1][0]:12	a[1][1][1]:13	a[1][1][2]:14
a[1][2][0]:15	a[1][2][1]:16	a[1][2][2]:0
a[1][3][0]:17	a[1][3][1]:18	a[1][3][2]:19

Note: The rule of initialization of multidimensional arrays is that last subscript varies most frequently and the first subscript varies least rapidly.

Example:

```
#include<stdio.h>
main()
{
        int d[5];
```

```
        int i;
        for(i=0;i<5;i++)
        {
                d[i]=i;
}
for(i=0;i<5;i++)
{
        printf("value in array %d\n",a[i]);
}
}
```

Pictorial representation of d will look like

d[0]	d[1]	d[2]	d[3]	d[4]
0	1	2	3	4

3.1.9 Arrays Using Functions

1-d arrays using functions

Passing individual array elements to a function

We can pass individual array elements as arguments to a function like other simple variables.

Example:

```
#include<stdio.h>
void check(int);
void main()
{
        int a[10],i;
        clrscr();
        printf("\n enter the array elements:");
        for(i=0;i<10;i++)
```

```
        {
        scanf("%d",&a[i]);
        check(a[i]);
    }
    void check(int num)
    {
        if(num%2==0)
        printf("%d is even\n",num);
        else
        printf("%d is odd\n",num);
    }
```

Output: enter the array elements:
1 2 3 4 5 6 7 8 9 10
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
10 is even

Example:
C program to pass a single element of an array to function

```
    #include <stdio.h>
    void display(int a)
    {
        printf("%d",a);
    }
    int main()
```

```
{
        int c[]={2,3,4};
        display(c[2]);  //Passing array element c[2] only.
        return 0;  }
```

Output

2 3 4

3.1.10 Passing whole 1-D array to a function

We can pass whole array as an actual argument to a function the corresponding formal arguments should be declared as an array variable of the same type.

Example 1:

```
#include<stdio.h>
main()
{
        int i, a[6]={1,2,3,4,5,6};
        func(a);
        printf("contents of array:");
        for(i=0;i<6;i++)  printf("%d",a[i]);
        printf("\n");
}
func(int val[])
{
        int sum=0,i;
        for(i=0;i<6;i++)
        {
                val[i]=val[i]*val[i]; sum+=val[i];
        }
        printf("the sum of squares:%d", sum);
}
```

Output

contents of array: 1 2 3 4 5 6

the sum of squares: 91

Example2:

Write a C program to pass an array containing age of person to a function. This function should find average age and display the average age in main function.

```
#include <stdio.h>
float average(float a[]);
int main()
{
        float avg, c[]={23.4, 55, 22.6, 3, 40.5, 18};
        avg=average(c);   /* Only name of array is passed as argument. */
        printf("Average age=%.2f",avg);
        return 0;
}
float average(float a[])
{
        int i;
        float avg, sum=0.0;
        for(i=0;i<6;++i)
        {
        sum+=a[i];
        }
        avg =(sum/6);
return avg;
}
```

Output

Average age= 27.08

Further Examples:

1. Write a program to find the largest of n numbers and its location in an array.

```
#include <stdio.h>
 #include<conio.h>
void main()
{
        int array[100], maximum, size, c, location = 1;
        clrscr();
        printf("Enter the number of elements in array\n");
        scanf("%d", &size);
        printf("Enter %d integers\n", size);
        for (c = 0; c < size; c++)
                scanf("%d", &array[c]);  maximum = array[0];
                for (c = 1; c < size; c++)
                {
                if (array[c] > maximum)
                {
                        maximum  = array[c];
                        location = c+1;
                }
}
printf("Maximum element is present at location %d and it's value is %d.\n", location, maximum);
getch();
}
```

Output:

```
Enter the number of elements in array  5
Enter 5 integers
2
4
7
9
```

1

Maximum element is present at location 4 and it's value is 9

2. Write a program to enter n number of digits. Form a number using these digits.

```
# include<stdio.h>
#include<conio.h>
#include<math.h>
void  main()
{
        int number=0,digit[10], numofdigits,i;
        clrscr(); printf("\n Enter the number of digits:");
        scanf("%d", &numofdigits);
for(i=0;i<numofdigits;i++)
{
        printf("\n Enter the %d th digit:", i);
        scanf("%d",&digit[i]);
}
i=0;
while(i<numofdigits)
{
        number= number + digit[i]* pow(10,i)
        i++;
}
printf("\n The number is : %d",number);
getch();  }
```

Output:

Enter the number of digits:

3

Enter the 0th digit:

5

Enter the 1th digit:

```
4
Enter the 2th digit:
3
The number is: 543
```

3. Matrix addition:

```
#include <stdio.h>  #include<conio.h>
void main()
{
        int m, n, c, d, first[10][10], second[10][10], sum[10][10];
        clrscr();
        printf("Enter the number of rows and columns of matrix\n");
        scanf("%d%d", &m, &n);
        printf("Enter the elements of first matrix\n");
        for ( c = 0 ; c < m ; c++ )
                for ( d = 0 ; d < n ; d++ )
                scanf("%d", &first[c][d]);
        printf("Enter the elements of second matrix\n");
        for ( c = 0 ; c < m ; c++ )
                for ( d = 0 ; d < n ; d++ )
                        scanf("%d", &second[c][d]);
        for ( c = 0 ; c < m ; c++ )
                for ( d = 0 ; d < n ; d++ )
                        sum[c][d] = first[c][d] + second[c][d];
        printf("Sum of entered matrices:-\n");
        for ( c = 0 ; c < m ; c++ )
        {
                for ( d = 0 ; d < n ; d++ )
                        printf("%d\t", sum[c][d]);
        printf("\n");
        }
getch();
}
```

Output:

Enter the number of rows and columns of matrix

2 2

Enter the elements of first matrix

1 2 3 4

Enter the elements of second matrix

5 6 2 1

Sum of entered matrices:- 6 8 5 5

4.0 Self-study Exercise

1. Compute sum of elements of an array in a program?
2. Write a program for histogram printing using an array?
3. Write a program for dice-rolling using an array instead of switch?
4. Sorting an array with bubble sort?
5. Write a program for binary search using an array?
6. Write a program to interchange the largest and the smallest number in the array.
7. Write a program to fill a square matrix with value 0 on the diagonals, 1 on the upper right triangle, and -1 on the lower left triangle.
8. Write a program to read and display a 2x2x2 array.
9. Write a program to calculate the number of duplicate entries in the array.
10. Given an array of integers, calculate the sum, mean, variance and standard deviation of the numbers in the array.
11. Write a program that reads a matrix and displays the sum of the elements above the main diagonal.
12. Write a program to calculate XA + YB where A and B are matrices and X=2, and Y=

5.0 Further Readings

Fundamentals of Structured Programming, Lubia Vinhas, 2016

Structured Programming with C++ by Kjell Backman, 2012

UNIT 3: FUNDAMENTALS OF STRINGS

CONTENTS

1.0 Introduction

2.0 Intended Learning Outcomes (ILOs)

3.0 Main Content

3.1 String

3.2 Reading strings

3.3 Writing string

3.4 Common Functions in String

4.0 Self Study Exercise

5.0 Further Readings

1.0 Introduction

A string is defined as a contiguous sequence of code units terminated by the first zero code unit (often called the NUL code unit).What this means is that a string cannot contain the zero code unit, as the first one seen marks the end of the string. The *length* of a string is the number of code units before the zero code unit. The memory occupied by a string is always one more code unit than the length, as space is needed to store the zero terminator. Generally, the term *string* means a string where the code unit is of type char which is exactly 8 bits on all modern machines. Stringliterals ("text" in the C source code) are converted to arrays during compilation.

2.0 Intended Learning Outcomes (ILOs)

By the end of the unit, you will be able to:

- define a string
- differentiate between a string and a character
- manipulate string
- mention some commonly used string input/output library functions
- read and write string
- declare a string variable.

3.0 Main Content

3.1 String

A string is a series of characters treated as a single unit. A string may include letters, digits and various special characters such as +, -, *, / and $. String literals or string constants in C are written in double quotation marks as follows:

"1000 Main Street" (a street address)

"(080)329-7082" (a telephone number)

"Kalamazoo, New York" (a city)

In C language strings are stored in array of char type along with null terminating character '\0' at the end.

When sizing the string array we need to add plus one to the actual size of the string to make space for the null terminating character, '\0'.

Syntax:

char fname[4];

The above statement declares a string called fname that can take up to 3 characters. It can be indexed just as a regular array as well.

fname[]={'t','w','o'};

character	T	w	o	\0
ASCII code	116	119	41	0

Generalized syntax is:-

char str[size];

when we declare the string in this way, we can store size-1 characters in the array because the last character would be the null character.

For example,

char mesg[10]; can store maximum of 9 characters.

If we want to print a string from a variable, such as four name string above we can do this.

e.g., printf(“First name:%s”,fname);

We can insert more than one variable. Conversion specification %s is used to insert a string and then go to each %s in our string, we are printing.

A string is an array of characters. Hence it can be indexed like an array.

char ourstr[6] = “EED”;
– ourstr[0] is ‘E’
– ourstr[1] is ‘E’
– ourstr[2] is ‘D’
– ourstr[3] is ‘\0’
– ourstr[4] is ‘\0’
– ourstr[5] is ‘\0’

‘E’	‘E’	‘D’	\0	“\0 “	“\0 “
ourstr[0]	ourstr[1]	ourstr[2]	ourstr[3]	ourstr[4]	ourstr[5]

3.2 Reading strings:

If we declare a string by writing
char str[100];
then str can be read from the user by using three ways;
1. Using scanf() function
2. Using gets() function
3. Using getchar(), getch(), or getche() function repeatedly

The string can be read using scanf() by writing
scanf(“%s”,str);

Although the syntax of scanf() function is well known and easy to use, the main pitfall with this function is that it terminates as soon as it finds a blank space. For example, if the user enters Hello World, then str will contain only Hello. This is because the moment a blank space is encountered, the string is terminated by the scanf() function.

Example:

```
char str[10];
printf("Enter a string\n");
scanf("%s",str);
```

The next method of reading a string a string is by using gets() function. The string can be read by writing

gets(str);

gets() is a function that overcomes the drawbacks of scanf(). The gets() function takes the starting address of the string which will hold the input. The string inputted using gets() is automatically terminated with a null character.

Example:

```
char str[10];
printf("Enter a string\n");
gets(str);
```

The string can also be read by calling the getchar() repeatedly to read a sequence of single characters (unless a terminating character is encountered) and simultaneously storing it in a character array as follows:

```
int i=0;
char str[10],ch;
getchar(ch);
while(ch!='\0')
{
        str[i]=ch;      // store the read character in str
        i++;
        getch(ch);    // get another character
```

```
}
str[i]='\0';        // terminate str with null character
```

3.3 Writing string

The string can be displayed on screen using three ways:

1. Using printf() function
2. Using puts() function
3. Using putchar() function repeatedly

The string can be displayed using pintf() by writing

printf("%s",str);

We can use width and precision specification along with %s. The width specifies the minimum output field width and the precision specifies the maximum number of characters to be displayed.

Example:

printf("%5.3s",str);

This statement would print only the first three characters in a total field of five charaters; also these three characters are right justified in the allocated width.

The next method of writing a string is by using the puts() function. The string can be displayed by writing:

```
puts(str);
```

It terminates the line with a newline character ('\n'). It returns an EOF(-1) if an error occurs and returns a positive number on success.

Finally the string can be written by calling the putchar( ) function repeatedly to print a sequence of single characters.

```
int i=0;
char str[10];
while(str[i]!='\0')
{
        putchar(str[i]);     // print the character on the screen
```

```
        i++;
    }
```

Example:

Read and display a string

```
#include<stdio.h>
#include<conio.h>
void main()
{
        char str[20];
        clrscr();
        printf("\n Enter a string:\n");
        gets(str);
        scanf("The string is:\n");
        puts(str);
getch();  }
```

Output:

```
Enter a string:
vssut burla
The string is:
vssut burla
```

3.4 Common Functions in String

Method Description char strcpy(s1, s2) Copy string char strcat(s1, s2) Append string int strcmp(s1, s2) Compare 2 strings int strlen(s) Return string length char strchr(s, int c) Find a character in string char strstr(s1, s2) Find string s2 in string s1

Type	Method	Description
char	strcpy(s1, s2)	Copy string
char	char strcat(s1, s2)	Append string

int	strcmp(s1, s2)	Compare 2 strings
int	strlen(s)	Return string length
char	strchr(s, int c)	Find a character in string
char	strstr(s1, s2)	Find string s2 in string s1

strcpy():

It is used to copy one string to another string. The content of the second string is copied to the content of the first string.

Syntax:

strcpy (string 1, string 2);

Example:

char mystr[10];

mystr = “Hello”; // Error! Illegal !!! Because we are assigning the value to mystr which is not possible in case of an string. We can only use "=" at declarations of C-String.

strcpy(mystr, “Hello”);

It sets value of mystr equal to “Hello”.

strcmp():

It is used to compare the contents of the two strings. If any mismatch occurs then it results the difference of ASCII values between the first occurrence of 2 different characters.

Syntax:

int strcmp(string 1, string 2);

Example:

char mystr_a[10] = “Hello”;

char mystr_b[10] = “Goodbye”;

– mystr_a == mystr_b; // NOT allowed!

The correct way is

if (strcmp(mystr_a, mystr_b))

printf ("Strings are NOT the same.");
else
printf("Strings are the same.");
Here it will check the ASCII value of H and G i.e, 72 and 71 and return the diference 1.

strcat():
It is used to concatenate i.e, combine the content of two strings.
Syntax:
strcat(string 1, string 2);

Example:
char fname[30]={“bob”};
char lname[]={“by”};
printf(“%s”, strcat(fname,lname));

Output:
bobby.

strlen():
It is used to return the length of a string.

Syntax:
int strlen(string);

Example:
char fname[30]={“bob”};
int length=strlen(fname);
It will return 3

strchr():
It is used to find a character in the string and returns the index of occurrence of the character for the first time in the string.
Syntax:

strchr(cstr);

Example:
char mystr[] = "This is a simple string";
char pch = strchr(mystr,‘s’);
The output of pch is mystr[3]

strstr():
It is used to return the existence of one string inside another string and it results the starting index of the string.
Syntax:
strstr(cstr1, cstr2);

Example:
Char mystr[]="This is a simple string";
char pch = strstr(mystr, “simple”);
here pch will point to mystr[10]

- String input/output library functions

Function prototype	Function description
int getchar(void);	Inputs the next character from the standard input and returns it as integer
int putchar(int c);	Prints the character stored in c and returns it as an integer
int puts(char s);	Prints the string s followed by new line character. Returns a non-zero integer if possible or EOF if an error occurs
int sprint(char s, char format,....)	Equivalent to printf,except the output is stored in the array s instead of printed in the screen. Returns the no.of characters written to s, or EOF if an error occurs
int sprint(char s, char format,....)	Equivalent to scanf, except the input is read from the array s rather than from the keyboard. Returns the no.of items successfully read by the function , or EOF if an error occurs

NOTE:
Character arrays are known as strings.

Self-review exercises:
1. Find the error in each of the following program segments and explain how to correct it:
• char s[10]; • strcpy(s,"hello",5); • prinf("%s\n",s); • printf("%s",'a'); • char s[12]; strcpy(s,"welcome home"); • If (strcmp(string 1, sring 2))
{ printf("the strings are equal\n");
}
2. Show 2 different methods of initializing character array vowel with the string of vowels "AEIOU"? 3. Writ a program to convert string to an integer?
4. Write a program to accept a line of text and a word. Display the no. of occurrences of that word in the text? 5. Write a program to read a word and re-write its characters in alphabetical order. 6. Write a program to insert a word before a given word in the text. 7. Write a program to count the number of characters, words and lines in the given text.

MODULE 8: STRUCTURE AND POINTERS IN C

UNIT 1: STRUCTURE AND POINTERS

CONTENTS

1.0 Introduction

2.0 Intended Learning Outcomes (ILOs)

3.0 Main Content

3.1 Structure

3.2 Pointers

3.3 Pointers and Addresses

3.4 Pointers and Function Arguments

3.5 Pointers and Arrays

1.0 Introduction

A structure is a user defined data type in C. A structure allows us to create data type that can be used to group items of different types into a single type. It is similar to array but array allows only a set of data values of one data type to be stored in it. It is equally similar to records in Pascal programming language. A pointer is an object in many programming languages that stores a memory address. This can be that of another value located in computer memory, or in some cases, that of memory-mapped computer hardware. A pointer references a location in memory, and obtaining the value stored at that location is known as dereferencing the pointer. As an analogy, a page number in a book's index could be considered a pointer to the corresponding page; dereferencing such a pointer would be done by flipping to the page with the given page number and reading the text found on that page. The actual format and content of a pointer variable is dependent on the underlying computer architecture.

2.0 Intended Learning Outcomes

By the end of the unit, you will be able to:

- Understand C structures and pointers

- Know how to define and use structures and pointers in C

3.0 Main Contents

3.1 Structure

A Structure is a user defined data type that can store related information together. The variable within a structure is of different data types and each has a name that is used to select it from the structure. C arrays allow you to define type of variables that can hold several data items of the same kind, but structure is another user defined data type available in C programming, which allows you to combine data items of different kinds.
Structures are used to represent a record. Suppose you want to keep track of your books in a library, you might want to track the following attributes about each book:

• Title
• Author
• Subject
• Book ID

Structure Declaration

It is declared using a keyword struct followed by the name of the structure. The variables of the structure are declared within the structure.

Example:

```
Struct struct-name
    {
        data_type var-name;
        data_type var-name;
    };
```

Structure Initialization

Assigning constants to the members of the structure is called initializing of structure.

Syntax:

```
struct struct_name
{
    data _type member_name1;
    data _type member_name2;
} struct_var={constant1,constant2};
```

Accessing the Members of a structure

A structure member variable is generally accessed using a '.' operator.

Syntax:

strcut_var.
member_name;

The dot operator is used to select a particular member of the structure. To assign value to the individual
Data members of the structure variable stud, we write,

stud.roll=01;
stud.name="Rahul";

To input values for data members of the structure variable stud, can be written as,
scanf("%d",&stud.roll);
scanf(''%s",&stud.name);

To print the values of structure variable stud, can be written as:
printf("%s",stud.roll);
printf("%f",stud.name);

3.2 Pointers

A pointer is a variable that contains the address of a variable. Pointers are much used in C, partly because they are sometimes the only way to express a computation, and partly because they usually lead to more compact and efficient code than can be obtained in other ways. Pointers and arrays are closely related; this chapter also explores this relationship and shows how to exploit it. Pointers have been lumped with the goto statement as a marvelous way to create impossible to understand programs. This is certainly true when they are used carelessly, and it is easy to create pointers that point somewhere unexpected. With discipline, however, pointers can alsobe used to achieve clarity and simplicity. This is the aspect that we will try to illustrate. The main change in ANSI C is to make explicit the rules about how pointers can be manipulated, in effect mandating what good programmers already practice and good compilers already

enforce. In addition, the type void * (pointer to void) replaces char * as the proper type for a generic pointer.

3.3 Pointers and Addresses

Let us begin with a simplified picture of how memory is organized. A typical machine has an array of consecutively numbered or addressed memory cells that may be manipulated individually or in contiguous groups. One common situation is that any byte can be a char, a pair of one-byte cells can be treated as a short integer, and four adjacent bytes form a long. A pointer is a group of cells (often two or four) that can hold an address. So if c is a char and p is a pointer that points to it, we could represent the situation this way:

The unary operator & gives the address of an object, so the statement

p = &c;

assigns the address of c to the variable p, and p is said to ``point to" c. The &operator only applies to objects in memory: variables and array elements. It cannot be applied to expressions, constants, or register variables. The unary operator * is the indirection or dereferencing operator; when applied to a pointer, it accesses the object the pointer points to. Suppose that x and y are integers and ip is a pointer to int. This artificial sequence shows how to declare a pointer and how to use & and *:

```
int x = 1, y = 2, z[10];
int *ip;
ip = &x;
y = *ip;
*ip = 0;
ip = &z[0];
```

The declaration of x, y, and z are what we've seen all along. The declaration of the pointer ip.

int *ip;

is intended as a mnemonic; it says that the expression *ip is an int. The syntax of the declaration for a variable mimics the syntax of expressions in which the variable might appear. This reasoning applies to function declarations as well. For example,

double *dp, atof(char *);

says that in an expression *dp and atof(s) have values of double, and that the argument of atof is a pointer to char. You should also note the implication that a pointer is constrained to point to a particular kind of object: every pointer points to a specific data type. If ip points to the integer x, then *ip can occur in any context where x could, so

*ip = *ip + 10;

increments *ip by 10. The unary operators * and & bind more tightly than arithmetic operators, so the assignment

y = *ip + 1

takes whatever ip points at, adds 1, and assigns the result to y, while

*ip += 1

increments what ip

points to, as do

++*ip and (*ip)++

The parentheses are necessary in this last example; without them, the expression would increment ip instead of what it points to, because unary operators like * and ++ associate right to left. Finally, since pointers are variables, they can be used without dereferencing. For example, if iq is another pointer to int,

iq = ip

copies the contents of ip into iq, thus making iq point to whatever ip pointed to.

3.4 Pointers and Function Arguments

Since C passes arguments to functions by value, there is no direct way for the called function to alter a variable in the calling function. For instance, a sorting routine might exchange two outoforder arguments with a function called swap. It is not enough to write

swap(a, b);

where the swap function is defined as

```
void swap(int x, int y)
{
        int temp;
        temp = x;
        x = y;
        y = temp;
}
```

Because of call by value, swap can't affect the arguments a and b in the routine that called it. The function above swaps copies of a and b. The way to obtain the desired effect is for the calling program to pass pointers to the values to be changed:

```
swap(&a, &b);
```

Since the operator & produces the address of a variable, &a is a pointer to a. In swap itself, the parameters are declared as pointers, and the operands are accessed indirectly through them.

```
        void swap(int *px, int *py) /* interchange *px and *py */
        {
        int temp;
        temp = *px;
        *px = *py;
        *py = temp;
        }
```

Pointer arguments enable a function to access and change objects in the function that called it. As an example, consider a function getint that performs free-format input conversion by breaking a stream of characters into integer values, one integer per call. getint has to return the value it found and also signal end of file when there is no more input. These values have to be passed back by separate paths, for no matter what value is used for EOF, that could also be the value of an input integer. One solution is to have getint return the end of file status as its function value, while using a pointer argument to

store the converted integer back in the calling function. This is the scheme used by scanf as well. The following loop fills an array with integers by calls to getint:

```
int n, array[SIZE], getint(int *);
for (n = 0; n < SIZE &&getint(&array[n]) != EOF; n++) ;
```

Each call sets array[n] to the next integer found in the input and increments n. Notice that it is essential to pass the address of array[n] to getint. Otherwise there is no way for getint to communicate the converted integer back to the caller. Our version of getint returns EOF for end of file, zero if the next input is not a number, and a positive value if the input contains a valid number.

```
#include <ctype.h>
int getch(void);
void ungetch(int);
int getint(int *pn)
{
        int c, sign;
        while (isspace(c = getch()));
        if (!isdigit(c) && c != EOF&& c != '+' && c != '-')
{
        ungetch(c);
        return 0;
}
sign = (c == '-') ? -1 : 1;
if (c == '+' || c == '-')
c = getch();
for (*pn = 0; isdigit(c), c = getch())
*pn = 10 * *pn + (c - '0');
*pn *= sign;
 if (c != EOF)
ungetch(c);
return c;
}
```

Throughout getint, *pn is used as an ordinary int variable. We have also used getch and ungetch so the one extra character that must be read can be pushed back onto the input.

3.5 Pointers and Arrays

In C, there is a strong relationship between pointers and arrays, strong enough that pointers and arrays should be discussed simultaneously. Any operation that can be achieved by array subscripting can also be done with pointers. The pointer version will in general be faster but, at least to the uninitiated, somewhat harder to understand. The declaration

int a[10];

defines an array of size 10, that is, a block of 10 consecutive objects named a[0], a[1], .,a[9]. The notation a[i] refers to the i-th element of the array. If pa is a pointer to an integer, declared as

int *pa;

then the assignment

pa = &a[0];

sets pa to point to element zero of a; that is, pa contains the address of a[0]. Now the assignment

x =*pa;

will copy the contents of a[0] into x. If pa points to a particular element of an array, then by definition pa+1 points to the next element, pa+i points i elements after pa, and pa-i points i elements before. Thus, if pa points to a[0], *(pa+1) refers to the contents of a[1], pa+i is the address of a[i], and *(pa+i) is the contents of a[i]. These remarks are true regardless of the type or size of the variables in the array a. The meaning of ``adding 1 to a pointer," and by extension, all pointer arithmetic, is that pa+1 points to the next object, and pa+i points to the i-th object beyond pa. The correspondence between indexing and pointer arithmetic is very close. By definition, the value of a variable or expression of type array is the address of element zero of the array. Thus after the assignment

pa = &a[0];

pa and a have identical values. Since the name of an array is a synonym for the location of the initial element, the assignment

pa=&a[0]
can also be written as
pa = a;

Rather more surprising, at first sight, is the fact that a reference to a[i] can also be written as
*(a+i). In evaluating a[i], C converts it to *(a+i) immediately; the two forms are equivalent. Applying the operator & to both parts of this equivalence, it follows that &a[i] and a+i are also identical: a+i is the address of the i-th element beyond a. As the other side of this coin, if pa is a pointer, expressions might use it with a subscript; pa[i] is identical to *(pa+i). In short, an array-and-index expression is equivalent to one written as a pointer and offset. There is one difference between an array name and a pointer that must be kept in mind. A pointer is a variable, so pa=a and pa++ are legal. But an array name is not a variable; constructions like a=pa and a++ are illegal. When an array name is passed to a function, what is passed is the location of the initial element. Within the called function, this argument is a local variable, and so an array name parameter is a pointer, that is, a variable containing an address. We can use this fact to write another version of strlen, which computes the length of a string. int strlen(char *s)

```
{
int n;
for (n = 0; *s != '\0', s++)
n++;
return n;
}
```

Since s is a pointer, incrementing it is perfectly legal; s++ has no effect on the character string in the function that called strlen, but merely increments strlen's private copy of the pointer. That means that calls like
strlen("hello, world");
strlen(array); strlen(ptr);
all work.
As formal parameters in a function definition,

char s[]; and char *s;
are equivalent; we prefer the latter because it says more explicitly that the variable is a pointer. When an array name is passed to a function, the function can at its convenience believe that it has been handed either an array or a pointer, and manipulate it accordingly. It can even use both notations if it seems appropriate and clear. It is possible to pass part of an array to a function, by passing a pointer to the beginning of the subarray. For example, if a is an array,

f(&a[2])and f(a+2)

both pass to the function f the address of the subarray that starts at a[2]. Within f, the parameter declaration can read
f(int arr[])
{ ... } or
f(int *arr) { ... }

So as far as f is concerned, the fact that the parameter refers to part of a larger array is of no consequence. If one is sure that the elements exist, it is also possible to index backwards in an array; p[-1], p[2], and so on are syntactically legal, and refer to the elements that immediately precede p[0]. Of course, it is illegal to refer to objects that are not within the array bound.

4.0 Self-study Exercise

1. Write a program using structures to read and display the information about an employee.
2. Write a program to read, display, add and subtract two complex numbers.
3. Write a program to enter two points and then calculate the distance between them.

5.0 Further Readings

Fundamentals of Structured Programming, Lubia Vinhas, 2016.
Structured Programming with C++ by Kjell Backman, 2012.