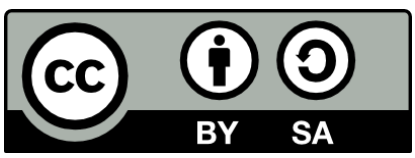
CSC 231: COMPUTER PROGRAMMING I

University of Ilorin
Centre for Open &
Distance Learning

CODL

Published by the Centre for Open and Distance Learning,
University of Ilorin, Nigeria

E-mail: codl@unilorin.edu.ng
Website: https://codl.unilorin.edu.ng



Course Development Team

Content Authors

Mr H.A Mojeeb

Content Editors

Abdulwahab Mahmud
Bankole Ogechi Ijeoma

Instructional Design

Koladafe Sunday Olawale
Ibrahim Mutiu Babatunde
Adimoha Daberechi Adimoha

From the Vice Chancellor

Courseware remains the nerve centre of Open and Distance Learning. Whereas some institutions and tutors depend entirely on Open Educational Resources (OER), CODL at the University of Ilorin considers it necessary to develop its own materials. Rich as OERs are and widely as they are deployed for supporting online education, adding to them in content and quality by individuals and institutions guarantees progress. Doing it in-house as we have done at the University of Ilorin has brought the best out of the Course Development Team across Faculties in the University. Credit must be given to the team for prompt completion and delivery of assigned tasks in spite of their very busy schedules. The development of the courseware is similar in many ways to the experience of a pregnant woman eagerly looking forward to the D-day when she will put to bed. It is customary that families waiting for the arrival of a new baby usually do so with high hopes. This is the apt description of the eagerness of the University of Ilorin in seeing that the centre for open and distance learning [CODL] takes off. The Vice-Chancellor, Prof. Sulyman Age Abdulkareem, deserves every accolade for committing huge financial and material resources to the centre. This commitment, no doubt, boosted the efforts of the team. Careful attention to quality standards, ODL compliance and UNILORIN CODL House Style brought the best out from the course development team. Responses to quality assurance with respect to writing, subject matter content, language and instructional design by authors, reviewers, editors and designers, though painstaking, have yielded the course materials now made available primarily to CODL students as open resources. Aiming at a parity of standards and esteem with regular university programmes is usually an expectation from students on open and distance education programmes. The reason being that stakeholders hold the view that graduates of face-to-face teaching and learning are superior to those exposed to online education. CODL has the dual-mode mandate. This implies a combination of face-to-face with open and distance education. It is in the light of this that our centre has developed its courseware to combine the strength of both modes to bring out the best from the students. CODL students, other categories of students of the University of Ilorin and similar institutions will find the courseware to be their most dependable companion for the acquisition of knowledge, skills and competences in their respective courses and programmes. Activities, assessments, assignments, exercises, reports, discussions and projects amongst others at various points in the courseware are targeted at achieving the objectives of teaching and learning. The courseware is interactive and directly points the attention of students and users to key issues helpful to their particular learning. Students' understanding has been viewed as a necessary ingredient at every point. Each course has also been broken into modules and their component units in sequential order At this juncture, I must commend past directors of this great centre for their painstaking efforts at ensuring that it sees the light of the day. Prof. M. O. Yusuf, Prof. A. A. Fajonyomi and Prof. H. O. Owolabi shall always be remembered for doing their best during their respective tenures. May God continually be pleased with them, Aameen.

Bashiru,A.Omipidan

Director, CODL

Professor S. A. Abdulkareem
Vice Chancellor

Foreword

Courseware is the livewire of Open and Distance Learning. Whereas some institutions and tutors depend entirely on Open Educational Resources (OER), CODL at the University of Ilorin considered it necessary to develop its materials. Rich as OERs are and widely as they are deployed for supporting online education, adding to them in content and quality by individuals and institutions guarantees progress.

Pursuing this goal has brought the best out of the Course Development Team across Faculties in the University. Despite giving attention to competing assignments within their work setting, the team has created time and eventually delivered. The development of the courseware is similar in many ways to the experience of a pregnant mother eagerly looking forward to the delivery date.

As with the eagerness for a coming baby, great expectation pervaded the air from the University Administration, CODL, Faculty and the writers themselves. Careful attention to quality standards, ODL compliance and UNILORIN CODL House Style brought the best out from the course development team. Response to quality assurance with respect to writing, subject matter content, language and instructional design by the authors, reviewers, editors and designers, though painstaking, has yielded the course materials now made available primarily to CODL students as open resources.

Aiming at parity of standards and esteem with regular university programmes is usually an expectation from students on open and distance education programmes. The reason being that stakeholders hold the view that graduates of face-to-face teaching and learning are superior to those exposed to online education. CODL has the dual mode mandate. This implies a combination of face-to-face with open and distance education.

With this in mind, the Centre has developed its courseware to combine the strength of both modes to bring out the best from the students. CODL students and other categories of students of the University of Ilorin and similar institutions will find the courseware to be their most dependable companion for the acquisition of knowledge, skills and competences in the respective courses and programmes.

Activities, assessments, assignments, exercises, reports, discussions and projects at various points in the courseware are targeted at achieving the Outcomes of teaching ang learning. The courseware is interactive and directly points the attention of students and users to key issues helpful to their particular learning. The student's understanding has been viewed as a necessary ingredient at every point. Each course has also been broken into modules and their component units in an ordered sequence. In it all, developers look forward to successful completion by CODL students.

Courseware for the Bachelor of Science in Computer Science housed primarily in the Faculty of Communication and Information Science provide the foundational model for Open and Distance Learning in the Centre for Open and Distance Learning at the University of Ilorin.

Henry O. Owolabi
Director, CODL

INTRODUCTION

In this course, Computer Programming I, a harmattan semester course, I will introduce you to the concept of programming using one of the most popular programming language; C Language. Through this course, I will expose you to the fundamental syntax and structure of C language, focusing on major aspects of programming such as variables, data types, operators, expression, statement, control structures, functions, and composite types. In the journey through this course, I will help you to understand that this course provides solid foundation for learning programming languages.

Course Goal

The goal of this course is to train you on how to use C-programming language to solve real-life computational problems by developing working programs.

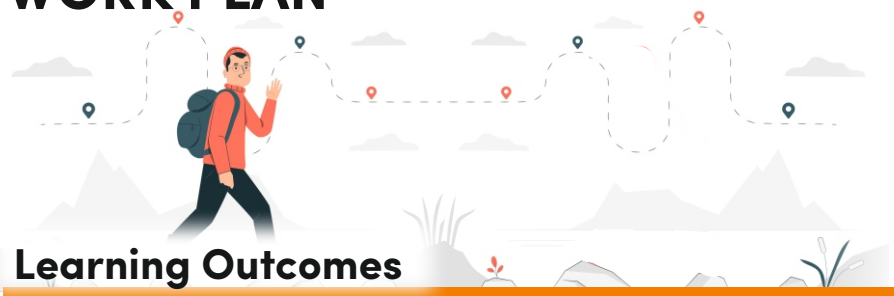
Required for:

Course requirements

CSC 212- Computer Programming II

CSC 233- Object-Oriented Programming (using JAVA)

WORK PLAN



Learning Outcomes

At the end of this course, you should be able to:

- I. Explain the structure and fundamental components of a C program
- II. Implement different control statements available in C

Week 01

- III. Work with composite and derived types in C to handle complex data structures
- IV. Create modular programs in C using Functions

Week 02

- V. Highlight The Application of C Language
- VI. Highlight Keywords in C language

Week 03

Course Guide

Module 1

Fundamentals of C programming Language

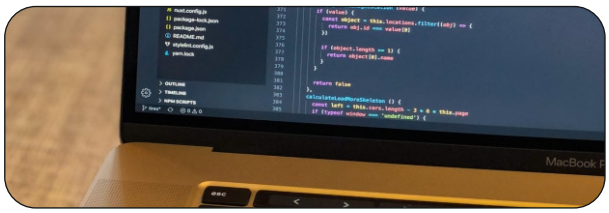
Unit 1 Introduction to C

Unit 2 Vocabulary and Types

Unit 3 Variable and Constant

Unit 4 Operators and Expressions

Unit 5 Statements and Structure of C program



Module 2

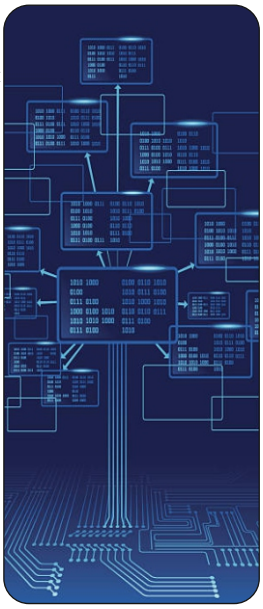
Standard Input/Output and Control Structures

Unit 1 Input and Output Functions in C

Unit 2 Selection Control Statements

Unit 3 Loops (Iteration Control Statement)

Unit 4 Termination Control Statements



Related Courses

CSC 112 – Introduction to Computer Science II

CSC 212 – Computer Programming II

CSC 233- Object-Oriented Programming (using JAVA)

Pre-requisite

CSC 112
Introduction to Computer Science II

VII. Describe the features of C Language

IX. Describe the historical backgrounnd of C Language

VIII. Write a program using C language

Week 04

Week 05

Module 3

Function In C

Unit 1 Function and its components

Unit 2 Function Call and Prototype

Unit 3 - Multiplexing Techniques

Module 4

Composites and User-Defined Types

Unit 1 Array and Enumeration Types

Unit 2 Pointers and Strings in C

Unit 3 Structures and Union

Course Requirements

Requirements for success

The CODL Programme is designed for learners who are absent from the lecturer in time and space. Therefore, you should refer to your Student Handbook, available on the website and in hard copy form, to get information on the procedure of distance/e-learning. You can contact the CODL helpdesk which is available 24/7 for every of your enquiry.

Visit CODL virtual classroom on http://codllms.unilorin.edu.ng. Then, log in with your credentials and click on CSC 231. Download and read through the unit of instruction for each week before the scheduled time of interaction with the course tutor/facilitator. You should also download and watch the relevant video and listen to the podcast so that you will understand and follow the course facilitator.

At the scheduled time, you are expected to log in to the classroom for interaction. Self-assessment component of the courseware is available as exercises to help you learn and master the content you have gone through.

You are to answer the Tutor Marked Assignment (TMA) for each unit and submit for assessment

Embedded Support Devices

Support menus for guide and references

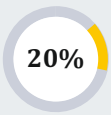
Throughout your interaction with this course material, you will notice some set of icons used for easier navigation of this course materials. We advise that you familiarize yourself with each of these icons as they will help you in no small ways in achieving success and easy completion of this course. Find in the table below, the complete icon set and their meaning.

 **Introduction**	 **Learning Outcomes**	 **Main Content**

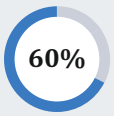
 Summary	 Tutor Marked Assignment	 Self Assessment
 Web Resources	 Downloadable Resources	 Discuss with Colleagues
 References	 Futher Reading	 Self Exploration

Grading and Assessment

TMA

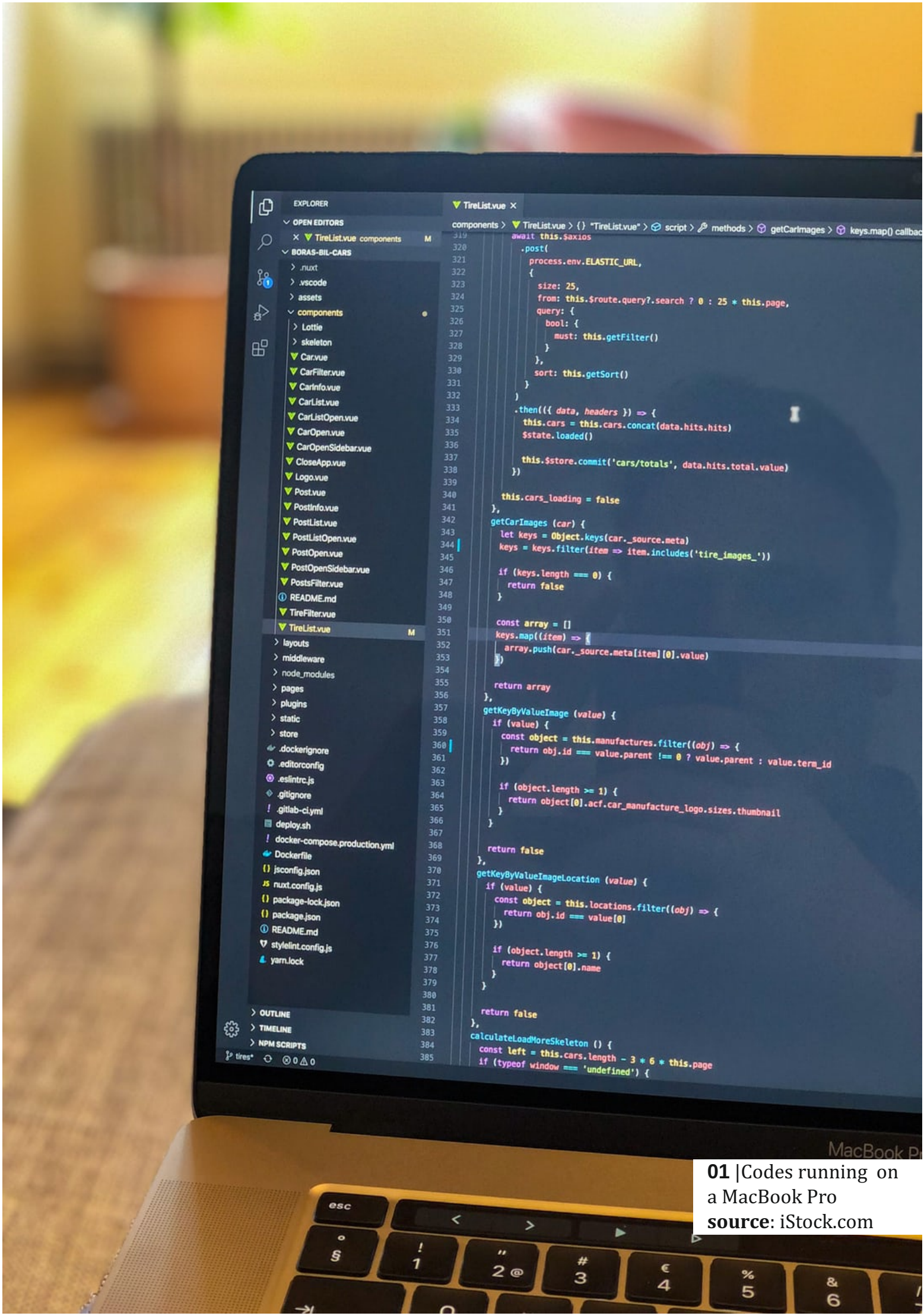


CA



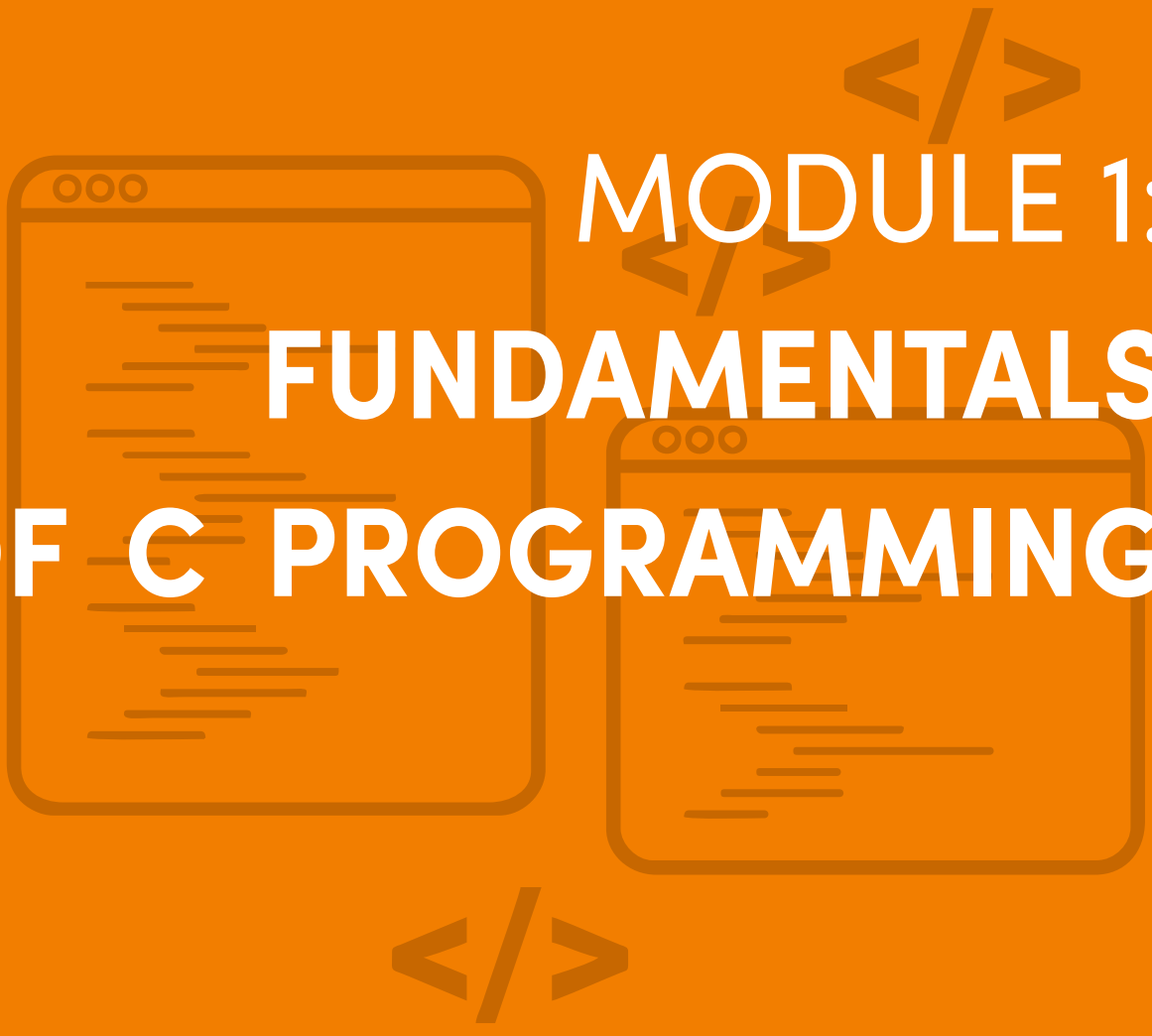
Exam

Total



01 |Codes running on a MacBook Pro
source: iStock.com

MODULE 1:
FUNDAMENTALS OF C PROGRAMMING



Unit 1:Introduction to C

Unit 2:Vocabulary and Types

Unit 3:Variable and Constant

Unit 4:Operators and Expressions

Unit 5:Statements and Structure of C program

C programming
source: iStock.com

UNIT 1

Introduction to C Programming

Introduction

I welcome you to the concept of programming using C language. Here, I will discuss in details, the history, features/characteristics, advantages, and applications of C programming language.

Learning Outcomes

At the end of this unit, you should be able to:

1. Discuss the historical background of C language
2. List at least 5 features of C language
3. State four advantages of C language over any other language
4. Highlight three applications of C language.

C Programming Language[SAQ1]

1 min

C is a general-purpose programming language with features economy of expression, modern flow control and data structures, and a rich set of operators. It will interest you to know that C is the most commonly used high-level programming language developed basically for system programming. It was used for writing operating systems. The first operating system written in C is Unix. Later operating systems like GNU/Linuxwere all written in C. Not only is C the language of operating systems, it is the precursor and inspiration for almost all of the most popular high-level languages available today. In fact, Perl, PHP, Python, and Ruby are all written in C. So, knowing C will enable you to understand and appreciate an entire family of programming languages built upon the traditions of C, such as Java, C++, C#, and so on.

While Assembly language can provide speed and maximum control of the program, C provides portability. Different processors are programmed using different Assembly languages, and having to choose and learn only one of them is too arbitrary. In fact, one of the main strengths of C is that it combines universality and portability across various computer architectures while retaining most of the control of the hardware provided by assembly language. Knowledge of C enables freedom.

History of C language

2 mins

I would like you to know that C was the direct successor of B, a stripped-down version of BCPL, created by Ken Thompson at Bell Labs, that was also a compiled language - User's Reference to B, used in early internal versions of the UNIX operating system. As noted in Ritchie's C History, "The B compiler on the PDP-7 did not generate machine instructions, but instead 'threaded code'; an interpretive scheme in which the compiler's output consists of a sequence of addresses of code fragments that perform the elementary operations. The operations typically— in particular for B — act on a simple stack machine". Thompson and Dennis Ritchie, also working at Bell Labs, improved B and called the result NB. Further extensions to NB created its logical successor, the traditional C in 1972. Most of UNIX was rewritten in NB, and then C, which resulted in a more portable operating system.

The portability of UNIX was the main reason for the initial popularity of both UNIX and C. Rather than creating a new operating system for each new machine, system programmers could simply write the few system-dependent parts required for the

machine and then writes a C compiler for the new system. Since most of the system utilities were thus written in C, it simply made sense also to write new utilities in C.

Do you know that American National Standards Institute began work on standardizing the C language in 1983 and completed the standard in 1989? The standard, ANSI X3.159-1989 "Programming Language C," served as the basis for all implementations of C compilers. The standardswere later updated in 1990 and 1999, allowing for features that were either in common useor were appearing in C++. The diagram in figure 1.1.1 depicts the evolution of the C programming language

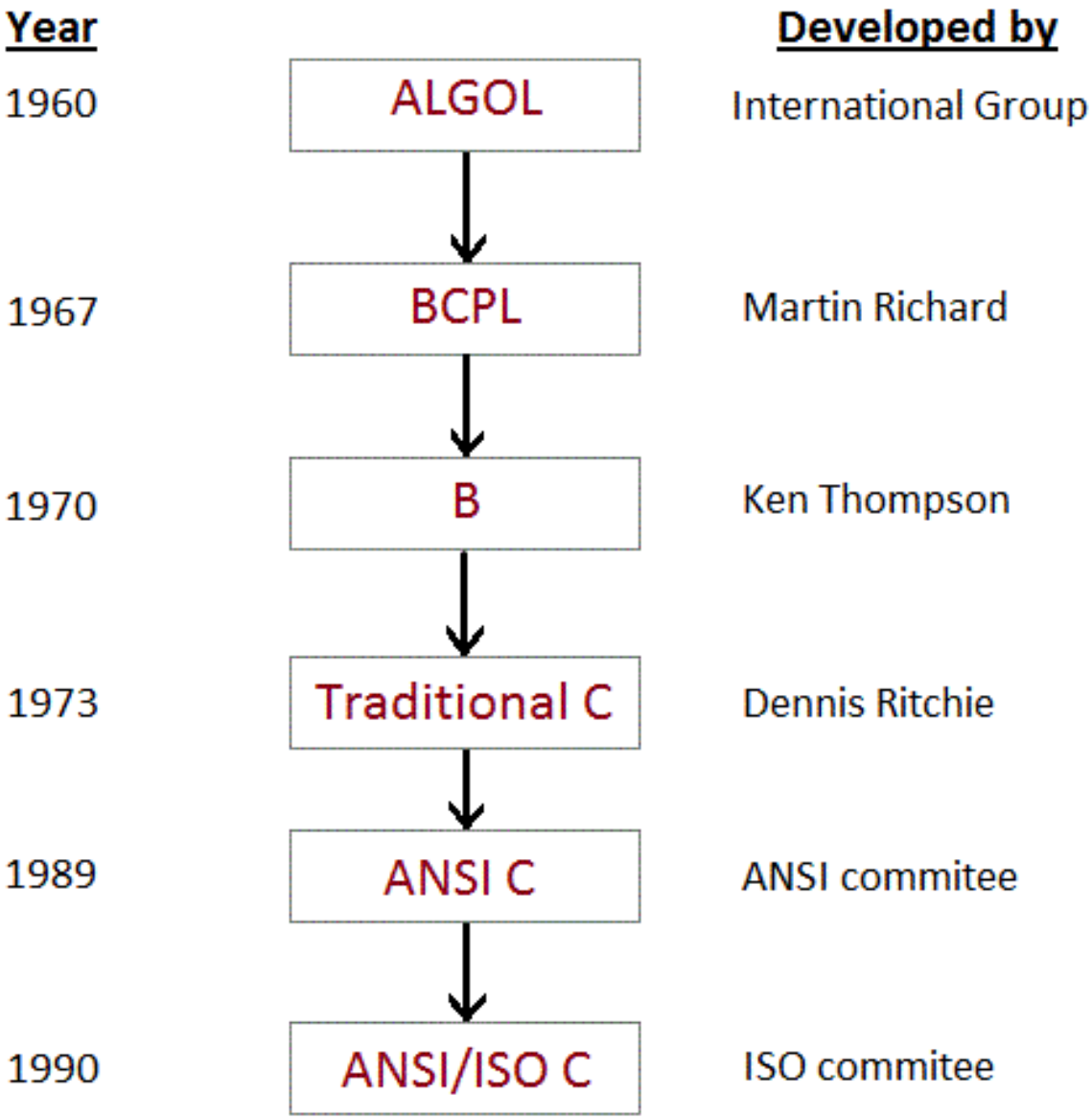


Figure 1.1.1: Evolution of C Language.

Features of C Language[SAQ2]

High-Level Assembler: Programmers coming to C from high-level languages like BASIC, PASCAL, etc. are usually surprised by how low-level C is. It does very little for you, and if you want it done, it expects you to write the code yourself. C is really a little more than an assembler with a few high-level features

High Processing Speed: The execution of your program is everything to C. C runs your program as quickly as possible due to its closeness to the system as little translations are needed to run codes. Note that this does not mean the development speed is high. In fact, almost the opposite is true

Portability: Due to the fact that C is nothing more than a high-level assembler, one may assume the portability of C programs to be just about zero. However, a program written in C is as portable as any other programs written in any high-level language.

Structured Language: C language defines its unique structure based on functions. This allows codes to be organized in a specific way and make program writing and reading much easier.

Memory Allocation: C language allows both static and dynamic memory allocation, thereby enhancing adequate memory management by the programmer. Key constructs in C, such as structs, pointers, and arrays are designed to structure and manipulate memory in an efficient, machine-independent fashion.

Bit Manipulation: C language allows manipulation of data in its lowest form of storage –bit, a feature that is very uncommon in other high level programming languages. It defines some operations that could be performed on data at bit levels

Advantages of C Language[SAQ3]

The following are some advantages of C over other programming languages that you should know;

·C language produces portable codes which are easily maintained and tested

·C codes executes faster due to its closeness to the system

·C language is the most suitable high-level language for maintaining system performance and managing system resource due to its consumption of minimal system footprints

·C language possesses a big codebase for a low-level application

·Unlike most programming languages, C allows the programmer to write directly to memory thereby improving memory management and allocation

·Most new programming languages syntax are inherited from C, so knowing C makes learning other programming languages easier.

Applications of C Language[SAQ4]

C language is useful for the development of the following:

·Operating systems

·embedded systems

·other system programs where performance matters alot

·Users' application with a low-level interface as a high-level" interface would affect performance.

Summary

In this unit, I have taught you the following:

- C is a general-purpose programming language developed basically for programming UNIXOperatingSystem.
- C evolves from previous programming languages succession from Algo to BCPL to B to Traditional C to K&RC to ANSI C to ANSI/ISO C and the standardized C99.
- C language has some special features such as portability, high execution speed, memory allocation, high level assembler, structured language, and low-level bit manipulation. It also has some advantages over other high-level programming languages in terms of processing speed, resource use, low-level functions, dependable syntax, and memory management.
- C language can be applied to develop operating systems, embedded systems, system programs, and simple user application programs

Self-Assessment Questions

SAQ

1. Discuss the historical background of the C programming language.
2. State five features of C language.
3. Highlight four advantages of C language.
4. Enumerate three application areas of C language.

Tutor Marked Assessment

- Discuss the evolution of C language based on its genealogy
- The execution speed of C programs is very fast; however, this is in contrast to their development speed. Discuss.
- Identify four features that make C suitable for developing Operating Systems

Activity

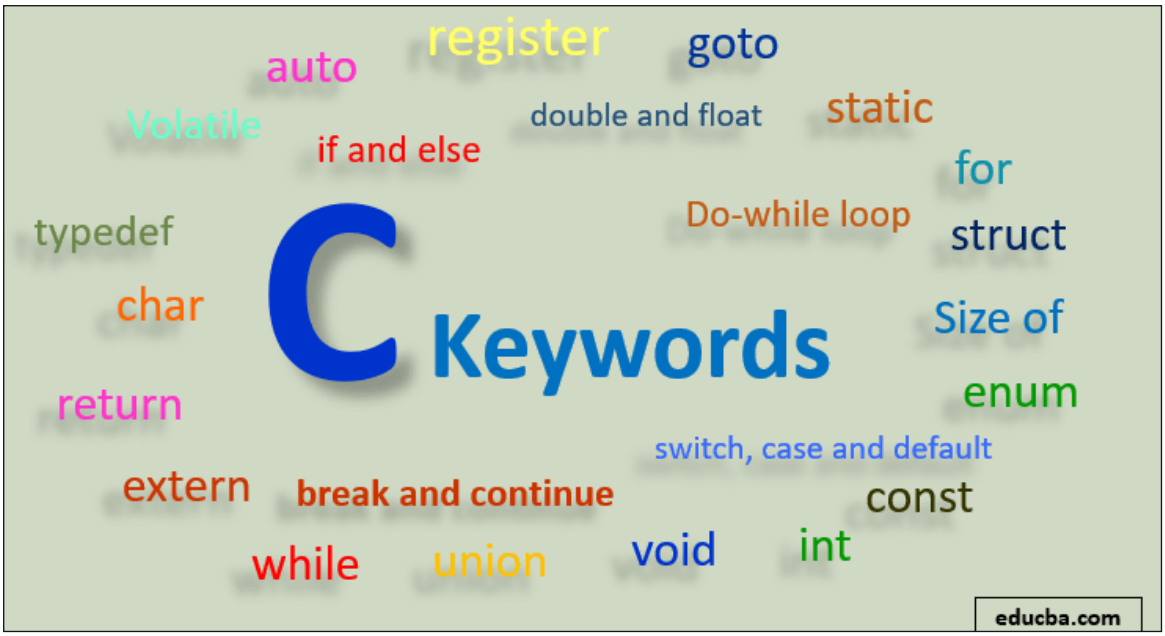
- ·Briefly discuss the history of C programming language
- ·Highlight five features of programming language

Further Reading

- https://en.wikipedia.org/wiki/C%20%28programming%20language%29
- https://en.wikipedia.org/wiki/operating%20systems
- https://en.wikipedia.org/wiki/Unix
- https://en.wikipedia.org/wiki/Linux
- https://en.wikipedia.org/wiki/compiled%20language
- https://www.bell-labs.com/usr/dmr/www/kbman.pdf
- https://www.bell-labs.com/usr/dmr/www/chist.html

References

- Balagurusamy, E (2007).
- Programming in ANSI C, 4th Edition, McGraw-Hill.
- Brian W. Kernighan Dennis M. Ritchie (2000).
- C Programming Language, the ANSCI C second edition. Printence Hall.



UNIT 2

Vocabulary and Types

Introduction

This is the unit where I will introduce you to the fundamental components that make up the C language. Concepts like character set, comments, and identifiers, keywords, datatypes, variables, and constants that form the vocabulary of C language are explained. We shed more light on these concepts as they form the solid foundations upon which all other concepts in C are built.

Learning Outcomes

At the end of this unit, you should be able to:

1. Clarify the elements of the two categories of the character set used in C
2. State the rule for constructing identifiers in C
3. Recall the four fundamental datatypes in C with their sizes in bytes.
4. Outline at least 10 keywords in C.
5. Declare, Initialise and assign appropriate values to variables

Character Set

The character set is the fundamental raw material of any language, and they are used to represent information. It is the set of all symbols recognizable in a language. Like natural languages, computer language will also have a well-defined character set, which is useful to build the programs.

The characters in C are grouped into two categories:

1. **Source character set**

a. Alphabets

b. Digits

c. Special Characters

2. **Execution character set**

a. Escape Sequence

Source character set

Alphabets: This is composed of upper and lower case letters of English alphabets

Uppercase letters A-Z

Lowercase letters a-z

Digits:0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Special Characters: the following special characters are representable in C:

~tilde %percent sign | vertical bar @at symbol +plus sign
< less than _ underscore - minus sign > greater than ^ caret
number sign = equal to & ampersand $ dollar sign
/slash (left parenthesis * asterisk \ back slash)right parenthesis
′ apostrophe : colon [left bracket "quotation mark ; semicolon
]right bracket ! exclamation mark , comma { left flower brace
? Question mark .dot operator } right flower brace

Execution Character Set

Did you know that Certain ASCII characters are unprintable, which means they are not

displayed on the screen or printer? Those characters perform other functions aside from displaying text. Examples are backspacing, moving to a newline, or ringing a bell. They are used in output statements. It is noteworthy to say that the escape sequence usually consists of a backslash and a letter or a combination of digits. An escape sequence is considered as a single character but a valid character constant. These are employed at the time of execution of the program. Execution characters set are always represented by a backslash (\) followed by a character. Each one of character constants represents one character, although they consist of two characters. These character combinations are called an escape sequence. Some escape sequences that you should know are:

Character	ASCII value	Escape Sequence	Result
Null	000	\0	Null
Alarm (bell)	007	\a	Beep Sound
Backspace	008	\b	Moves to the previous position
Horizontal tab	009	\t	Moves next horizontal tab
New line	010	\n	Moves next Line
Vertical tab	011	\v	Moves next vertical tab
Form feed	012	\f	Moves to initial position of next page
Carriage return	013	\r	Moves beginning of the line
Double quote	034	\"	Present Double quotes
Single quote	039	\'	Present Apostrophe
Question mark	063	\?	Present Question Mark
Back slash	092	\\	Present back slash
Octal number		\000	
Hexadecimal number		\x	

Identifiers and Keywords

An identifier in any programming language refers to the name given to program entities such as variables, functions, arrays, etc., created by the programmer. It is consequently used to refer to that entity in the program.

Every programming language has a rule for composing identifiers. In C, an identifier starts with a letter A to Z, a to z, or an underscore '_' followed by zero or more letters, underscores, and digits (0 to 9).

From the rules I've mentioned earlier, the following points should be noted:

·An identifier can be composed of letters, digits, and underscore.

·It must begin with either a letter or an underscore.

·It must not contain white space.

·Other punctuation characters such as @, $, and % are not allowed within identifiers

·Identifiers are Case-sensitive, i.e., account & ACCOUNT are two different identifiers

·C keywords cannot be used as identifiers.

·It can be of any length, but a maximum of first 31 characters are recognized.

Some valid identifiers are:

Age, _getchar, __sin, x1, sur_name, x_1, If, Main..

The invalid identifiers are:

324, short, price$, My Name, 2good, int, @middle

Keywords, on the other hand, are reserved identifiers that have special meaning to the compiler. Keywords cannot be used as names for any user-defined program entities. They are purposely used to instruct or give commands to the compiler. Some keywords in C are presented in the table below which is tagged as Table 1.

Table 1: Some Keywords in C

auto	else	long	switch
break	Enum	register	typedef
case	extern	return	union
char	float	short	unsigned
const	for	signed	void
continue	Goto	sizeof	volatile
default	if	static	while
do	Int	struct	…..

Data Types in C

C is a typed (though weakly) programming language. A typed programming language uses a type system to interpret the bitstreams in memory. A type is a rule that defines how to store values in memory and which operations are possible on those values. It defines the number of bytes available for storing values and hence the range of possible values. We use different types to store different information. The relation between types and raw memory is illustrated in figure 1.2.1 below:

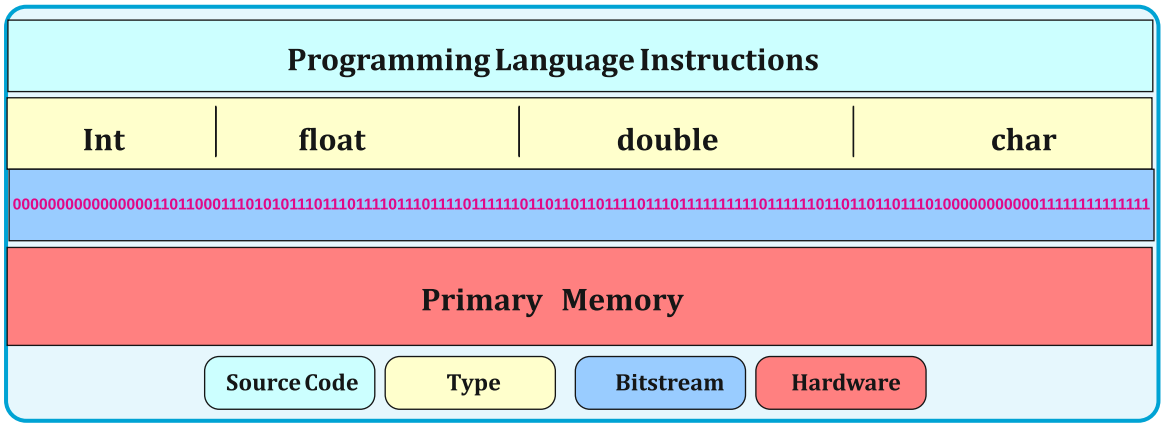


Figure1.2.1: Relation between Types and Memory

In this section, I will describe the four most common types in the C language and the ranges of values that these types allow.

The four basic types of the C language for performing calculations are:

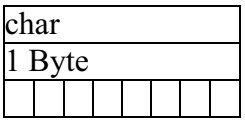
·char

·int

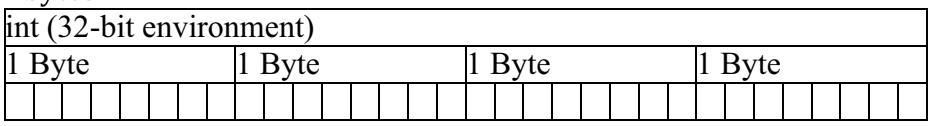
·float

·**double**

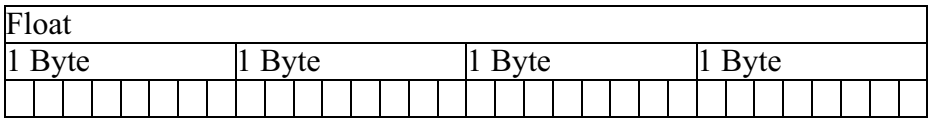
A char occupies one byte and can store a small integer value, a single character or a single symbol:



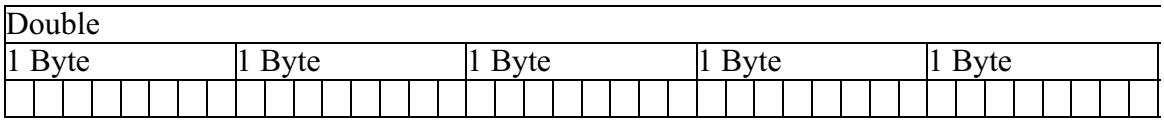
An int occupies one word and can store an integer value. In a 32-bit environment, an int occupies 4 bytes:



A float typically occupies 4 bytes and can store a single-precision, floating-point number:



A double typically occupies 8 bytes and can store a double-precision, floating-point number:



Type Qualifier

Several qualifiers can be applied to the basic types to modify their sizes, ranges, and nature. They include:

Size Specifiers: Size specifiers adjust the size of the int and double types.

Type int Size Specifiers

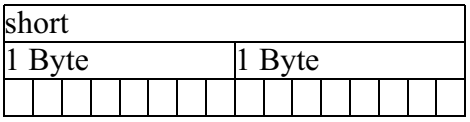
Specifying the size of an int ensures that the type contains a minimum number of bits. The three specifiers are:

·short

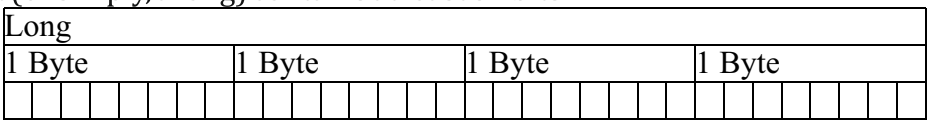
·long

·long long

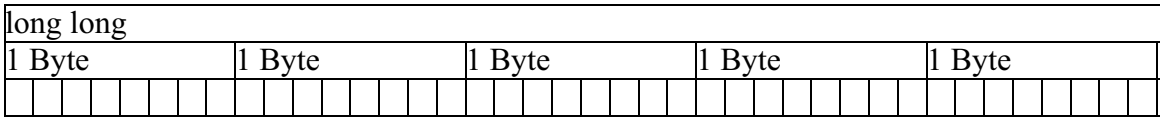
A **short int** (or simply, a short) contains at least 16 bits:



A long int (or simply, a long) contains at least 32 bits:



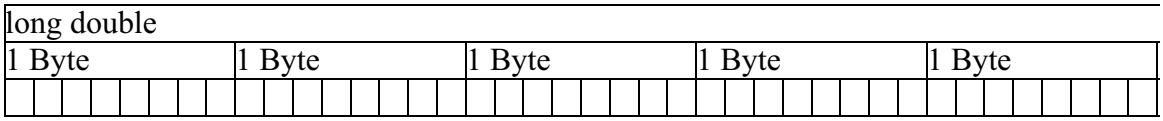
A long longint (or simply, a long long) contains at least 64 bits:



The size of a simple int is no less than the size of a short.

Type double Size Specifier

The size of a long double depends on the environment and is typically at least 64 bits:



Specifying the long double type only ensures that it contains at least as many bits as a double. The C language does not require a long double to contain a minimum number of bits.

Sign Qualifier: Sign qualifier specifies the sign of int or char type, which consequently affects the range of values for that type. The two keywords for specifying sign are:

·signed

·unsigned

unsignedint represents anint type with no sign; it has 32-bits and ranges from 0 to 232-1. unsigned char depicts an 8-bits number or character with ANSI values range from 0 to 255.

Note: unsigned can also work with long and short.

signed char represents a number from –128 to 127 (8-bit signed value).

const Qualifier: Any type can hold a constant value. A constant value cannot be changed. To qualify a type as holding a constant value we use the keyword const. A type qualified as const is unmodifiable. That is, if a program instruction attempts to modify a qualified const type, the compiler will report an error.

Value Ranges

The number of bytes allocated for a type determines the range of values the type can store.

Integral Types

The ranges of values for the integral types are shown below. Ranges for some types depend on the execution environment

Type	Size	Min	Max
char	8 bits	-128	127
char	8 bits	0	255
short	>=16 bits	-32,768	32,767
int	2 bytes	-32,768	32,767
int	4 bytes	-2,147,483,648	2,147,483,647
long	>=32 bits	-2,147,483,648	2,147,483,647
long long	>=64 bits	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

Note that the limits on both a char and an int vary with the execution environment.

Floating-Point Types

The limits on a float and double depend on the execution environment:

Type	Size	Significant digits	Min Exponent	Max Exponent
float	minimum	6	-37	37
float	typical	6	-37	37
double	minimum	10	-37	37
double	typical	15	-307	307
long double	typical	15	-307	307

Note that both the number of significant digits and the range of the exponent are limited. The limits on the exponent are in base 10.

Summary

In this unit, I have taught you the following:

- The C character set is composed of English alphabets (A –Z, a – z), digits (0..9) and escape sequences (such as \t, \n, \0, \\, \", etc.)
- Identifiers are names given to program entities such as variables, functions, arrays, etc., created by the programmer and are composed as: starts with a letter A to Z, a to z, or an underscore '_' followed by zero or more letters, underscores, and digits (0 to 9).
- There are four basic datatypes C language that specify the kind of data that can be processed in the language and their sizes in the memory. These datatypes are char, int, float, and double. Each of these datatypes has its size in bytes and range of values.
- Constants are other entities in C language that refer to fixed values that do not change during the execution of a program,
- A variable is an entity in the program that its values can be altered during program execution. It has five attributes: name, type, value, address, and scope. Variables must be declared in C before use using the format: Datatype Varname1[, Varname2,……Varnamen];
- A declared variable can be initialized and assigned values that match with its specificdatatype. Based on where they are declared within the program, variables can be local or global. Based on the storage class, variables can be automatic, extern, static, register, and volatile.
- Constant variables store values that can never be changed during execution. It can be defined using the const qualifier and the #define preprocessor.

Self-Assessment Questions

FAQ

1. What are the elements of the two categories of the character set used in C?
2. Highlight the rule for composing identifiers in C
3. List the four fundamental datatypes in C with their memory sizes in bytes.
4. List at least 10 keywords in C.

Tutor Marked Assessment

- Differentiate between identifier and variable
- To use named constant, using the keyword const is saver than #define preprocessor, Justify.
- Identify the valid and invalid identifiers from the following list:
  Ade, 1word, sur name, _123, age2, name_1, last_name, first-name, ACCOUNT, account.name, _size_2, and 2go.

Activity

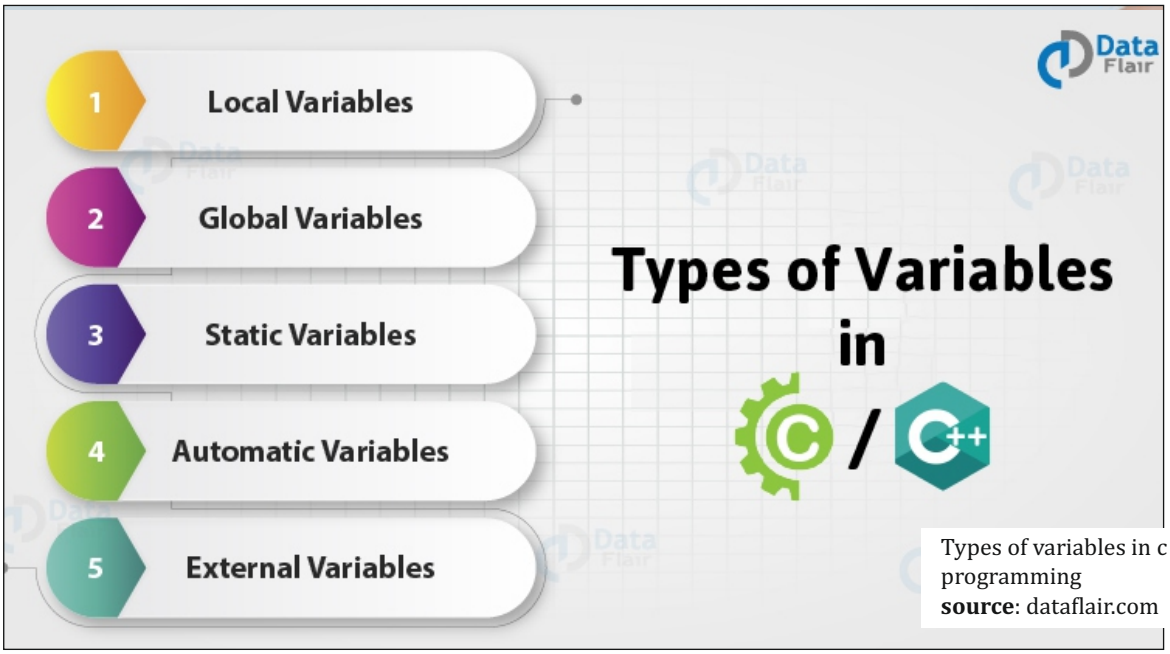
- Mention two rules for composing identifiers in C programming
- Give an example each of variable declaration, initialization, and assignment concerning the four basic data types

Further Reading

- http://aboutc.weebly.com/c-character-set.html
- https://en.wikibooks.org/wiki/Computer%20Programming%2FVariables
- https://en.wikibooks.org/wiki/Computer%20Programming%2FTypes

References

- Balagurusamy, E (2007).
- Programming in ANSI C, 4th Edition, McGraw-Hill.
- Deitel, P. J., and Deitel, H. (2006).
- C-How to program (5th Ed).
- Prentice-Hall, ISBN 978-0132404167



Types of variables in c programming
source: dataflair.com

UNIT 3

Variable and Constant

Introduction

This unit will expose you to the concept of variables and constants as a crucial foundation for understanding programming. Attributes of variable, the syntax for declaring and initializing variables, storage classes of variable and constant variables are discussed.

Learning Outcomes

At the end of this unit, you should be able to:

1. Highlight the attributes of a variable
2. State the syntax for declaring variables.
3. Initialize and assign appropriate values to variables
4. List the five kinds of constants available in C

Constants

1 min

When we refer to a Constant, it is a fixed value that does not change during the execution of a program. They are representations of values of some types, which are called Literals in C. Figure 1.3.1 shows the types of constants in C.

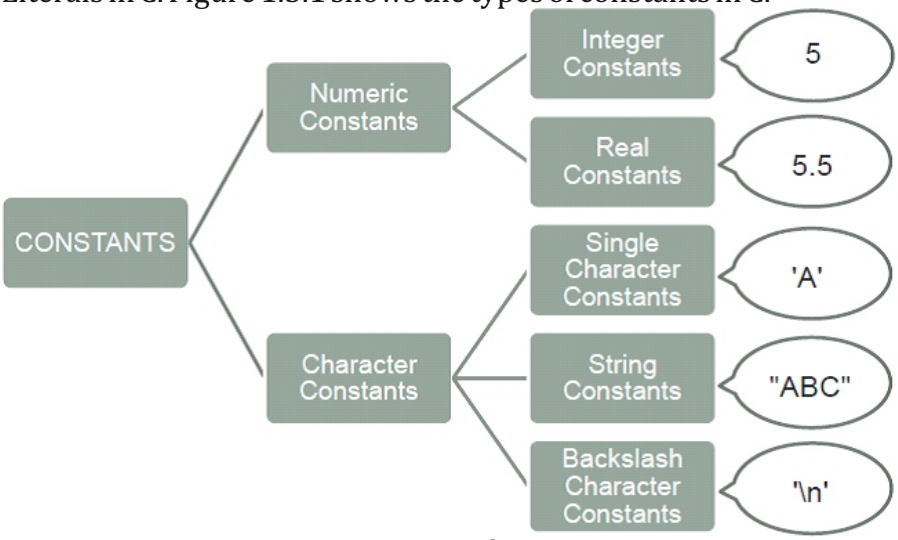


Figure.1 3.1: Types of Constants in C

C allows literal/constant values for integer, character, and floating-point types:

Integer literals are specified as whole numbers, e.g.,46, -2, 1000, 200, etc.

Floating point literals specify a number with decimal point e.g. 3.14159, 0.345, -1.8930, 0.5, etc. you can also use mantissa/exponent (scientific) notation e.g -3.14159e2, 4.5845e-6, etc

Character literals are specified with a single character in single quotes, e.g. 'a,' 'c,'' B,' etc. Special characters are specified with escape characters enclosed in single quotes, e.g. '\n,' '\t,' '\\,' etc.

String literals are represented as a sequence of zero or more characters enclosed in double-quotes, e.g., "ABC," "John Wick," "Hello world," etc.

Variable [SAQ1,2,3 &4]

3 mins

I will present a Variable to you as a program entity that can assume values that can be altered during program execution. Variables are simply names used to refer to some location in memory – a location that holds a value with which we are working. It may help to think of variables as a placeholder for a value. You can think of a variable as being equivalent to its assigned value. So, if you have a variable i that stores an integer value 4, then it follows that i + 1 will equal 5.Like most programming languages, C can use and process named variables and their contents. **Associated with all variables**

are the following five(5) attributes:

Identifier: It denotes the name of the variable by which it can be referenced.

Value: the data referred to by the variable

Datatype: determines the type of data that can be stored by the variable and its size

Address: The location in the memory where the value of the variable is stored

Scope: the section or area within the program where the variable exist and can be referenced.

Declaring, Initializing, and Assigning Variables

In C, all variables must be declared before use. A declaration creates and binds the variable with all its attributes. It specifies a type followed by a list of one or more identifiers of that type. Declaration has the following generic form:

datatypevarname[, varname2, ..., varnamen];

Elements within the square bracket are optional

For example: *intsome_number;*

This statement means we are declaring some space for a variable called some number, which will be used to store integer data. Note that we must specify the type of data that a variable will store. Multiple variables can as well be declared with one statement. Other examples:

int count; float a; double percent, total; unsigned char x,y,z; longintaLongInt;

longAnotherLongInt; unsigned long a_1, a_2, a_3; unsigned long int b_1, b_2, b_3;

After declaring variables, you can assign a value to a variable later on using a statement like this: *some_number = 3;*

Note that the value to be assigned must match the type specified during a declaration

You can also assign a variable the value of another variable:
anumber = anothernumber;

Or assign multiple variables the same value with one statement:
anumber = anothernumber = yetanothernumber = 3;

Assuming variables anumber, anothernumber and yetanothernumber have been declared already.

We can also declare and assign some content to a variable at the same time, such as:

intsome_number = 3;

This is called **initialization**. In C, all variables must be initialized before use, or else it will cause a garbage error since C does not collect garbage automatically, and the value of an uninitialized variable is undefined in the C standards. Examples of variable initialization are as follow:

char esc = '\\';

inti = 0;

int limit = MAXLINE+1;

floateps = 1.0e-5;

If the variable in question is not automatic, the initialization is done once only, conceptionally before the program starts executing, and the initializer must be a constant expression. An explicitly initialized automatic variable is initialized each time the function or blocks it is in is entered; the initializer may be any expression. External and static variables are initialized to zero by default. Automatic variables for which is no explicit initializer have undefined (i.e., garbage) values.Where declarations appear in the program affects the scope of the variable in question and its visibility. So, in terms of scope and visibility, the variable is of two types:

·Local variable

·Global variable

To declare a local variable, you place the declaration at the beginning (i.e., before any non-declarative statements) of the block to which the variable is intended to be used. To declare a global variable, declare the variable outside of any block. If a variable is global, it can be read and written from anywhere in your program.

Global variables are not considered good programming practice and should be avoided whenever possible, except if necessary. They inhibit code readability, create naming conflicts, waste memory, and can create difficult-to-trace bugs. Excessive usage of globals is usually a sign of laziness or poor design. However, if there is a situation where local variables may create more obtuse and unreadable code, there's no shame in using globals.

Storage Class of Variables

The storage class provides information about the location and the visibility of variables within a program. They specify how long the variable exists, stores, and its access control. There are five storage classes for variables in C, and each of them has their respective keywords, these are:

·auto: auto is a modifier that specifies an "automatic" variable that is automatically created when in scope and destroyed when out of scope. If you think this sounds like pretty much what you've been doing all along when you declare a variable, you're right: all declared items within a block are implicitly "automatic."

Features of automatic variables:

1. Keyword used - auto

2. Storage - Memory

3. Default value - Garbage value (random value)

4. Scope - Local to the block in which it is defined

5. Lifetime - Value persists while the control remains within the block

6. Keyword optionality– Optional

Extern: extern is used when a file needs to access a variable in another file that it may not have included directly. Therefore, extern provides the compiler with sufficient information to access the remote variable. A local variable qualified with the extern keyword will have a global scope. Extern is often used to make a local variable accessible by other blocks in the program.

Features of extern variable:

1. Keyword used - extern

2. Storage - Memory

3. Default value - Zero

4. Scope - Global (all over the program)

5. Lifetime - Value persists until the program's execution comesto an end

6. Keywc rd optionality - Optional if declared outside all the functions

Register: register is a hint to the compiler to attempt to optimize the storage of the given variable by storing it in a register of the computer's CPU when the program is run. Most optimizing compilers do this anyway, so the use of this keyword is often unnecessary. In fact, ANSI C states that a compiler can ignore this keyword if it so desires – and many do.

Features of register variables:

1. Keyword used - register

2. Storage - CPU registers (values can be retrieved faster thanfrom memory)

3. Default value - Garbage value

4. Scope - Local to the block in which it is defined

5. Lifetime - Value persists while the control remains withinthe block

6. Keyword optionality - Mandatory to use the keyword

Volatile: volatile is a special type of modifier that informs the compiler that the value of the variable may be changed by external entities other than the program itself. This is necessary for certain programs compiled with optimizations – if a variable were not defined volatile, then the compiler may assume that certain operations involving the variable are safe to optimize away when in fact, they aren't. Volatile is particularly relevant when working with embedded systems (where a program may not have complete control of a variable)and multi-threaded applications.

Static: static is a useful keyword that makes a variable to retain its value in the memory even when its block is exited. When you declare a global variable as static, you cannot access the variable through the extern keyword from other files in your project. This is called static linkage. When you declare a local variable as static, it is created just like any other variable. However, when the variable goes out of scope (i.e., the block it was local to is finished), the variable stays in memory, retaining its value until the program ends. While this behaviour resembles that of global variables, static variables still obey scope rules and, therefore, cannot be accessed outside of their scope. This is called static storage duration

Constant Variable

We use qualifier const for the declaration of any variable to specify that its value will not be changed. This declares a variable as being constant, therefore the name, constant variable.

Examples:

const double e = 2.71828182845905;

const double pi = 3.1415926;

constintmaxlength = 2356;

constintval = (3*7+6)*5;

floateps = 1.0e-5;

Apart from the const keyword, we can also use the preprocessor constant to create named constant or can be called macro in our program. This can be achieved using the #define preprocessor. The generic form is: #define CONSTNAME literal

Example: #define PI 3.14159

What happened?

The C preprocessor runs before the compiler.

Every time it sees the token PI, it substitutes with the value 3.14159.

The compiler is then run with this "pre-processed" C code.

For some purposes, we say #define can be harmfully used, and it is usually preferable to use a const keyword instead. It is possible, for instance, to #define, say, NUM as the number 3, but if you try to print the macro, thinking that NUM represents a string that you can show on the screen, the program will have an error. #define also has no regard for type. It disregards the structure of your program, replacing the text everywhere (in effect, disregarding scope), which could be advantageous in some circumstances, but can be the source of serious bugs.We will say it is good convention to write #defined words in all capitals, so a programmer will know that this is not a variable that you have declared but a #defined macro.

Constants are useful for several reasons:

·Tells the reader of the code that a value does not change

·Makes reading large pieces of code easier

·Tells the compiler that a value does not change

·The com piler can potentially compile faster code.

Summary

In this unit, I have taught you the following:

- Constants are other entities in C language that refer to fixed values that do not change during the execution of a program such as 10, 0.004, 'a,' "John," "\t."
- A variable is an entity in the program that its values can be altered during program execution. It has five attributes: name, type, value, address, and scope. Variables must be declared in C before use using the format: Datatype Varname1[, Varname2,……Varnamen];
- A declared variable can be initialized and assigned values that match with its specificdatatype. Based on where they are declared within the program, variables can be local or global. Based on the storage class, variables can be automatic, extern, static, register, and volatile.
- Constant variables store values that can never be changed during execution. It can be defined using the const qualifier and the #define preprocessor.

Self-Assessment Questions

SAQ

1. State the five attributes of a variable
2. Write the syntax for declaring variables and give examples.
3. Describe how a variable is initialized and assigned values in C
4. Enumerate with examples the five kinds of constants available in C

Tutor Marked Assessment

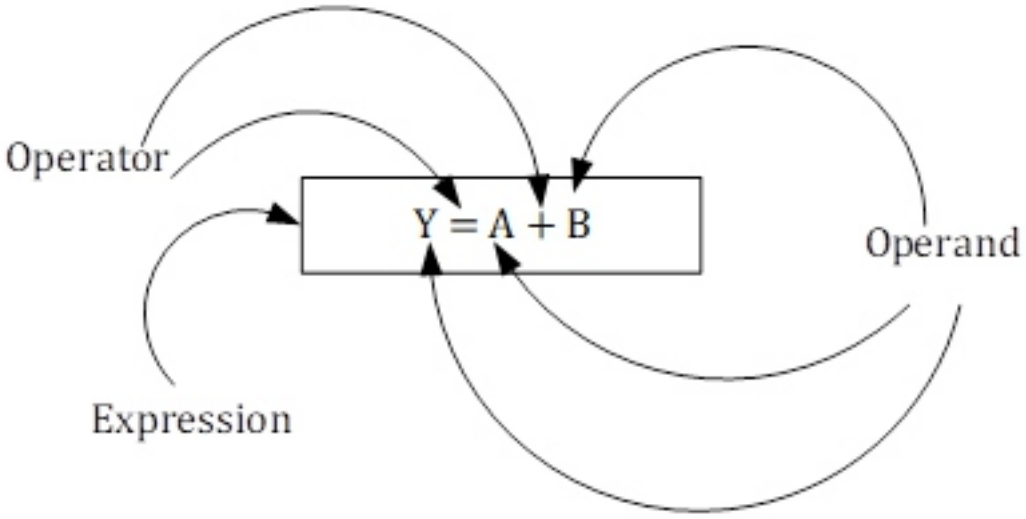
- To use named constant, using the keyword const is safer than #define preprocessor, Justify.
- Distinguish between the following storage classes of variables: auto, extern, static, register, and volatile

Further Reading

- http://aboutc.weebly.com/c-character-set.html
- https://en.wikibooks.org/wiki/Computer%20Programming%2FVariables
- https://en.wikibooks.org/wiki/Computer%20Programming%2FTypes

References

- Balagurusamy, E (2007).
- Programming in ANSI C, 4th Edition, McGraw-Hill.
- Deitel, P. J., and Deitel, H. (2006).
- C-How to program (5th Ed).
- Prentice-Hall, ISBN 978-0132404167



A diagram depicting operators , expressions and operand.
source: codesansar.com

UNIT 4

Operators and Expression

Introduction

In this unit, I will discuss the fundamentals of operators and expressions in C. This focuses on different types of operators available in C to operate on data and how the operators are used to form expressions. It also covers operator precedence and type conversions in mixed mode expression. This unit is essential in understanding the manipulation of data to produce information.

Learning Outcomes

At the end of this unit, you should be able to:

1. Identify at least five types of operators supported by C.
2. Arrange operators according to their precedence level.
3. Evaluate mixed-typed expressions.
4. Use casting to explicitly handle type conversion in expressions.

Operators [SAQ1-4]

10 mins

An operator is a symbol that tells the computer to perform some computation on the value provided. Operators are used in manipulating data and variables, which are referred to as operand. Generally, operators are divided into three: Unary, Binary, and Tenary operators. Unary operators operate only on one operand. Binary operators work on two operands. Ternary operators operate more than two operands.

Arithmetic Operators

Arithmetic operators are binary operators used to perform a basic mathematical operation. The binary arithmetic operators are: + (addition), - (subtraction), * (multiplication), / (division), and % (modulus). The table below presents how these operators are used in C.

Arithmetic Operators

Operator	Symbol	Action	Example
Addition	+	Adds operands	x + y
Subtraction	-	Subs second from first	x - y
Negation	-	Negates operand	-x
Multiplication	*	Multiplies operands	x * y
Division	/	Divides first by second (integer quotient)	x / y
Modulus	%	Remainder of divide op	x % y

4 / 3 returns 1 instead of 1.33 because 4 and 3 are both integer value

5 / 2 returns 2

Do you also observe that the expression x % y produces the remainder when x is divided by y, and thus is zero when y divides x exactly. For example:

4% 2 returns 0

4 % 3 returns 1

5 % 8 returns 5

The % operator cannot be applied to a float or double. The direction of truncation for / and the sign of the result for % are machine-dependent for negative operands, as the action was taken on overflow or underflow.

The binary + and - operators have the same precedence, which is lower than the precedence of *, / and %, which is, in turn, lower than unary + and -. Arithmetic operators associate left to right.

Relational Operators

Relational operators allow you to compare variables or values by magnitude. The operators return a value of 1 if the result of the operation is true, 0 if false.The relational operators are

>,>=, <,<=, ==,!=. The meanings of these operators are presented in the table below:

Relational Operators

- Relational operators allow you to compare variables.
  - They return a 1 value for true and a 0 for false.

Operator	Symbol	Example
Equals	==	x == y NOT x = y
Greater than	>	x > y
Less than	<	x < y
Greater/equals	>=	x >= y
Less than/equals	<=	x <= y
Not equal	!=	x != y

- There is no bool type in C. Instead, C uses:
  - 0 as false
  - Non-zero integer as true

For example:

4>2 returns 1

5<3 returns 0

2 ==2 returns 1

8>=9 returns 0.

Relational operators are often used to make branching decisions in a program.

>,>=, < and <=have the same precedence. Just below them in precedence are the equality operators ==and !=.

Relational operators have lower precedence than arithmetic operators, so an expression like

i< lim-1 is taken as i< (lim-1), as would be expected.

Logical Operators

Logical operators are used in comparing the results of two or more relational expressions. Comparison is based on logical reasoning. There are three (3) logical operators in c: && (AND), || (OR) and ! (NOT).

The logical operators && and ||: Both of these operators produce 1 if the relationship is true and 0 for false. Both of these operators are short-circuited; if the result of the expression can be determined from the first operand, the second is ignored. The && operator has higher precedence than the || operator.

&& is used to evaluate expressions left to right and returns a 1 if both relations are true. For example:

```
int x = 7;

int y = 5;

if(x == 7 && y == 5) {

...

}
```

Here, the && operator checks the left-most expression, then the expression to its right. If there were more than two expressions chained (e.g.,x && y && z), the operator would check x first, then y, continue rightwards to z if neither x nor y is zero. Since both relations return true, the && operator returns true, and the code block is executed.

```
if(x == 5 && y == 5) {

...
```

}

The && operator checks in the same way as before, and finds that the first expression is false. The && operator stops evaluating as soon as it finds a statement to be false, and returns a false.

||, on the other hand, is used to evaluate expressions left to right and returns a 1 if at least one of the expressions is true. For example, consider the following code:

/* Using the same variables as before. */

if(x == 2 || y == 5) {

...}

The || operator checks both expressions, finds that the latter is true, and returns true. The || operator here helps us check the left-most expression, finds it false, but continues to evaluate the next expression. It finds that the next expression returns true, stops, and returns a 1. Do you observe that the && operator ceases when it finds an expression that returns false, the || operator ceases when it finds an expression that returns true?

It is worth noting that C does not have Boolean values (true and false) commonly found in other languages. It instead interprets a 0 as false, and any nonzero value as true.

You will see that the precedence of && is higher than that of ||, and both are lower than relational and equality operators, so expressions like i< lim-1 && (c=getchar()) != '\n'&& c != EOF need no extra parentheses. But since the precedence of != is higher than assignment, parentheses are needed in(c=getchar()) != '\n'to achieve the desired result of assignment to c and then comparison with '\n'.

The unary negation operator! converts a non-zero operand into 0, and a zero operand into 1. A common use of !is in constructions like

if (!valid) rather thanif (valid == 0)

It will be hard for us to generalize about which form is better. Constructions like !valid read nicely (``if not valid''), but more complicated ones can be hard to understand. ! has higher precedence compared to && and ||

Bitwise operators

C allows you to operate on the bit representations of integer variables using some special operators. Generally, these operators are called bit-wise operators. All integers can be thought of in binary form. For example, suppose integers have 16-bits

6552010 = 1111 11111111 00002 = FFF016 = 1777608

In C, hexadecimal literals begin with 0x, and octal literals begin with 0.

x=65520;base 10

x=0xfff0;base 16 (hex)

x=0177760;base 8 (octal)

The shift operators << and >>

x <<nShifts the bits in x n positions to the left, shifting in zeros on the right.

If x = 1111 11111111 00002

x<< 1 equals 1111 1111 1110 00002

x >>nShifts the bits in x n positions right. Shifts in the sign if it is a signed integer (arithmetic shift), shifts in 0 if it is an unsigned integer

x>> 1 is 0111 1111 1111 10002 (unsigned)

x>> 1 is 1111 11111111 10002 (signed)

Bitwise logical operations(&, |, ^ and ~)

These operators work on all integer types.

&(Bitwise AND):

x= 0xFFF0

y= 0x002F

x&y= 0x0020

| (Bitwise Inclusive OR)

x|y = 0xFFFF

^ (Bitwise Exclusive OR)

x^y = 0xFFDF

~ (The complement operator)

~y = 0xFFD0

Complements all of the bits of y.

Shift Operators, Multiplication and Division

Since, multiplication and division are often slower thanthe bitwise shift, multiplying by 2 can be replaced by shifting 1 bit to the left. For example:

n = 10

n*2 gives the same result as n<<1

n*4 gives the same result as n<<2

Likewise, division by 2 can be replaced by shifting 1 bit to the right.

n = 10

n/2gives the same result as n>>1

n/4gives the same result as n>>2;

Increment and Decrement Operators

C provides two unary operators for incrementing and decrementing variables. The increment operator ++ adds 1 to its operand, while the decrement operator -- subtracts 1. Programmers have frequently used ++ to increment variables, as in

if (c == '\n')

++nl;

The unusual aspect is that ++ and -- may be used either as prefix operators (before the variable, as in ++n) or postfix operators (after the variable: n++). In both cases, the effect is to increment n. But the expression ++n increments n before its value is used, while n++ increments n after its value has been used

This means that in a context where the value is being used, not just the effect, ++n, and n++ are different. If n is 5, then

x = n++;

sets x to 5, but

x = ++n;

sets x to 6.

However, in both cases, n becomes 6, this is called side effect. The increment and decrement operators can only be applied to variables; an expression like (i+j)++ is illegal.

In a context where no value is wanted, just the incrementing effect, as in

if (c == '\n')

nl++;

prefix and postfix are the same.

Assignment Operators

The assignment operator = assigns the value of the right-hand side (rhs) to the left-hand side (lhs) in an expression. It stores the value of the right operand into the location determined by the left operand, which must be an lvalue (a value that has an address and therefore can be assigned to).

The value of the whole expression is the value of the rhs. For example, the expression:

x=3assigns 3 to x and has the value 3.

x = (y = 3) +1 y is assigned 3 , the value of (y=3) is 3 and x is assigned 4

Compound assignment operators can be formed by combining = with some other operators in the format x op= y as a shorthand for x = x op (y). we use the following compound assignment in C :*=, /=, %=, +=, -=, <<=, >>=, &=, ^=, and |=. Hence, the following expressions are the same:

OperatorEquivalent to:

x *= yx = x * y

y -= z + 1y = y - (z + 1)

a /= ba = a / b

x += y / 8x = x + (y / 8)

y %= 3y = y % 3

Since the value of the assignment expression is the value of the left operand, assignments can be chained; e.g., the expression a = b = c = 0; would assign the value zero to all three variables.

Comma Operator

In C, an expression can be composed of multiple sub-expressions separated by commas. The commas allow us to associate all the sub-expressions to one assignment operator. The implication of this is that the sub-expressions are evaluated left to right, and the entire expression evaluates to the value of the rightmost sub-expression.

Example:

x = (a++, b++);

a is incremented, b is assigned to xandb is incremented. Note that parenthesis is required because the comma operator has lower precedence than the assignment operator.

The comma operator is often used in for loops. Example:

inti, sum;

for (i=0,sum=0;i<100;i++){

sum += i;

}

Conditional Operator (? :)

A conditional operator in C is a ternary operator that takes three operands; conditions to be checked, the value when the condition is true, and value when the condition is false. It is represented by the symbol '?:'. The first operand (specified before the '?:') is the evaluating (conditional expression.

The second and third operands are expressions that can yield numerical values. If the return value of the conditional expression is true, the second expression is evaluated. Otherwise, the third expression is evaluated. Hence, the result of the conditional operator is the result of the expression considered for evaluation.

The generic form we use for conditional operator is

exp1 ?exp2 : exp3

this equivalent to:

if exp1 is true (non-zero)

value is exp2 (exp3 is not evaluated)

if exp1 is false (0),

value is exp3 (exp2 is not evaluated)

Example:

z = (x > y) ?x : y;

This is equivalent to:

```
if (x > y)
   z = x;
else
   z = y;
```

The conditional operator essentially allows you to embed an "if" statement into an expression.

Expressions

Programming languages support operators that combine variables and constants for transforming existing data into new data. The combination of these elements (operators, variables, and or constants forms what we call "expression" in programming. In C, an expression is a combination of variables, constants, and operators written according to the syntax of C language. In C, every expression evaluates to a value, i.e., every expression results in some value of a certain type that can be assigned to a variable.

Operator Precedence and Associativity

Each operator in C has precedence associated with it. We use the precedence to determine how an expression involving more than one operator is evaluated. There are distinct levels of precedence, and an operator may belong to one of these levels. The operators of higher precedence are evaluated first. The operators of the same precedence are evaluated from right to left or from left to right, depending on the level. This is known as the associativity property of an operator. The table given below gives the precedence of each operator when encountered in an expression.

Order	Category	Operator	Operation	Associativity
1	Highest precedence	( ) [ ] ? : : .	Function call	L ? R Left to Right
2	Unary	! ~ + - ++ - - & * Size of	Logical negation (NOT) Bitwise 1's complement Unary plus Unary minus Pre or post-increment Pre or post decrement Address Indirection Size of operant in bytes	R ? L Right -> Left
3	Member Access	.* ?*	Dereference Dereference	L ? R
4	Multiplication	* / %	Multiply Divide Modulus	L ? R
5	Additive	+ -	Binary Plus Binary Minus	L ? R
6	Shift	<< >>	Shift Left Shift Right	L ? R
7	Relational	< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	L ? R

8	Equality	== !=	Equal to Not Equal to	L ? R
9	Bitwise AAND	&	Bitwise AND	L ? R
10	Bitwise XOR	^	Bitwise XOR	L ? R
11	Bitwise OR	|	Bitwise OR	L ? R
12	Logical AND	&&	Logical AND	L ? R
14	Conditional	? :	Ternary Operator	R ? L
15	Assignment	= *= %= /= += -= &= ^= |= <<= >>=	Assignment Assign product Assign reminder Assign quotient Assign sum Assign difference Assign bitwise AND Assign bitwise XOR Assign bitwise OR Assign left shift Assign right shift	R ? L
16	Comma	,	Evaluate	L ? R

Type Conversion in Mixed-Type Expressions

C permits the mixing of constants and variables of different types in an expression. C automatically converts any intermediate values to the proper type so that the expression can be evaluated without losing any significance. This automatic type conversion is known as implicit type conversion. During an evaluation of arithmetic and relational expressions it adheres to very strict rule and type conversion. The rule is: "If the operands are of different types, the lower type is automatically converted to the higher type before the operation proceeds." This is called promotion.

The table below lists the type of the promoted operand.

	Right Operand						
Left Operand	long double	double	float	long long	long	int	short
long double	long double	long double	long double	long double	long double	long double	long doubl
double	long double	double	double	double	double	double	doubl
float	long double	double	float	float	float	float	float

long long	long double	double	float	long long	long long	long long	long long
long	long double	double	float	long long	long	long	Long
int	long double	double	float	long long	long	int	Int
short	long double	double	float	long long	long	int	short
char	long double	double	float	long long	long	int	short

For example,

1034 * 10 evaluates to 10340 // an int result

1034 * 10.0 evaluates to 10340.0 // a double result

1034 * 10L evaluates to 10340L // a long result

1034 * 10.f evaluates to 10340.0f // a float result

However, with assignment expression,the conversion is done in two ways. If the left operand in an assignment expression is of a higher type than the right operand, the compiler promotes the right operand to the type of the left operand. For example,

```
int num;

double cash;

num= 23;

cash = num; //  the type of num is promoted to double, and cash has a value of 23.000000
```

If the left operand in an assignment expression is of a lower type than the right operand, the compiler truncates the right operand to the type of the left operand. This is called narrowing. For example,

```
int num;

double cash;

cash= 23.45;

num = cash; //  the type of cash is narrowed to int and num has a value  of 23
```

Explicit Conversion

Many times, there may exist a situation where we want to force a type conversion in a way that is different from the automatic conversion. Consider for example the calculation of the ratio of the number of female and male students in a class

Ratio = female_students / male_student

Since female_students and male_students would be declared as integers, the decimal part will be rounded off, and its ratio will represent a wrong figure. This problem can be solved by converting locally one of the variables to the floating-point as shown below:

Ratio = (float) female_students / male_students

The operator float converts the female_students to floating-point for evaluation of the expression. Then using the rule of automatic conversion, the division is performed by floating-point mode, thus retaining the fractional part of the result. The process of such a local conversion is known as explicit conversion or casting. The general form is:

(type_name) expression

Summary

In this unit, I have taught you the following:

- The various types of operators including, arithmetic, relational, logical, assignment, bitwise, and conditional operators.
- That expression is formed from the combination of operators, variables, and constant values and is evaluated to a value.
- The precedence of all the types of operators and their associativity.
- Those expressions are evaluated based on the precedence level of the operators and their associativity.
- That mixed-type expression contained variables and operands of different types, and the type of value returned after an evaluation is determined by implicit conversion.
- The two modes of implicit type conversion: promotion or narrowing.
- That programmer can explicitly perform type conversion if the need arises through casting.

Self-Assessment Questions

SAQ

1. Illustrate the operators under the following categories: Arithmetic, Relational, Logical, Bitwise, Increment & Decrement, Assignment, and Conditional operators.
2. Reorder the following operators with respect to their precedence level and the order in which they are evaluated in an expression: *, &&, %, +, &, <, ||, ++, ==, !=, ^, -, /, !
3. Evaluate the following expression and state the datatype of the resulting values, given that a = 4 and b = 2:

   1. 2* 4.8 / b – (5>4.5)2. 3<a* 3 || 3.0 > 1

   3. 5 > a++ && 2 < ++b || 0 4. c = 3.0 + a++ / b--
4. Apply casting on the expressions in C above so that the datatype of the result of expression:

   i.1 is intii. 2 is longiii. 3 is doubleiv. 4 is int

Tutor Marked Assessment

- Write an expression to accurately calculate the average age of 5 students in a class using the appropriate type and casting.
- State the rules for evaluating a mixed-mode expression.
- Using suitable examples, describe how type conversion is handled in assignment expression.

Activity

- List the types of operators
- Mention two types of operators and their associativity

Further Reading

- http://ecomputernotes.com/what-is-c/types-and-variables/what-is-expressions-type-of-expression
- https://ict.senecacollege.ca/~ipc144/pages/content/expre.html

References

- Balagurusamy, E (2007).
- Programming in ANSI C, 4th Edition, McGraw-Hill.
- Shola, P. B. (2002).
- Learn C-Programming Language (2nd Ed).
- Reflect Publishers, ISBN 978-047-174-X

Structure of C program
source: iStock.com

UNIT 5

Statements and Structure of C Program

Introduction

I welcome you to this unit which will focus on statements as the building blocks of C programs. It explains the various kinds of statements that can be contained in C programs. It also provides a comprehensive explanation of the structure of a complete C program. Furthermore, it presents and analyses the first program to be considered in this course. Lastly, the process of compiling and running the program using codeblocks editor is discussed.

Learning Outcomes

At the end of this unit, you should be able to:

1. Define statement
2. Describe the four types of statement in C
3. Write a simple program of your choice following the standard structure of C programs

Statements

You will find it interesting to know that statement is a command that instructs the computer to take a specific action, such as a display to the screen, or collect input. In the most general sense, a statement is the part of your program that can be executed. A computer program in C is made up of a series of statements grouped into blocks. In C, a statement is terminated with a semi-colon (;). For instance, an expression becomes a statement if it is ended with a semi-colon. For example:

x++; is a statement created from the expression x++;

Assignment expressions are statements by default since they cause the system to perform an action. So, they must be terminated with a semicolon. Examples of statements are:

a=a+1;

a--;

printf("%d",a);

Types of Statement

There are four types of statements in C. They are:

·**Type Declaration Statement:** we use this type of statement to define the types of variables used in a C program. Examples:

int count;

float a;

double percent, total;

charx,y,z;

·**Input/Output Statement:** we use this statement to perform the function of supplying input data to a program and obtaining the output results from it. Examples:

printf ("Hello");

putchar(c);

puts(str);

getchar();

scanf(%d,&a);

gets(str);

·**Expression Statement:** we use this type of statement used to perform arithmetic calculations and computations to return a value. It also includes function calls since function also returns a value to its caller. Expression statements are used basically to transform input data into useful information. Examples:

Area = PI * r * r;

y = x / i;

area(7);

mean = (a+b+c)/3;

·**Control Statement:** This type of statement is used to control the order of execution of other statements in a program. They are used for selection, repetition, branching, and termination. Examples include if statement, if-else statement, switch statement, while statement, for a statement, break statement, and goto statement. Detailed discussion on the control statement is presented in another unit.

Based on composition, a statement can as well be categorized into three:

·**Simple Statement:** this is composed of a single statement that ended with a semi-colon (;). Examples:

int I = 1;

puts("thank you for coming");

y = 2* 4 / a;

swap(a,b);

·**Compound Statement:** Sequence of one or more statements can be combined into one by putting them inside a curly bracket { }. This type of statement is called a compound statement. The C compiler treats the collection of these statements like they are a statement. Example:

```
{

printf ("Hello, ");

printf ("world! \n");

}
```

·**Null Statement:** This type of statement is composed of no expression but only the semicolon (;). It is a statement that causes the compiler to do nothing. It is useful when handling conditions in programs. Example:

```
;  it is a null statement.
```

Suggestions on Writing Statements

C is a free form language, so you may type the statements in any style you feel comfortable, such as the following:

```
a=

a+

1;a--;
```

Line breaks can be anywhere, and spaces can be added as one wish. However, here are some recommendations to be followed while writing statements.

DO: stay consistent with your use of whitespace.

DO: put block braces on their line. This makes the code easier to read.

DO: line up block braces { } so that it is easy to find the beginning and end of a block.

AVOID: spreading a single statement across multiple lines if there is no need. Try to keep it on one line.

Structure of a Program in C [SAQ1-3]

A program in C is composed of one or more functions organized logically. Each function performs a specific task. A function is composed of a group or sequence of C statements that are executed together to produce a result. One of the functions in any C program must be named main. Execution of the program begins from the main function, and all other functions in the program must be reachable from the main function for them to

be executed.

A Program Template in C

A program in C consists of the following parts:

·**Documentation:** This consists of comments, some description of the program, and any other useful points that can be referenced later.

·**Preprocessor Directives:** This consists of links and definitions. Link provides instruction to the compiler to link function from the library function and definition consists of symbolic constants.

·**Global Declaration:** Consists of function declaration and global variables.

·**Main function:** main(){}Every C program must have a main() function which is the starting point of the program execution.

·**Subprograms:** include other user-defined functions.

Let us explore the sections with an example. We will write a program to find the area of a circle for a given radius of 10cm.

Analysis: The formula to compute the area of a circle is πr2, where π is PI = 3.1416, and r is the radius of the circle. Now let us write the C code to compute the area of the circle.

```
/*
Author: Hammed Adeleye
Date: 25/ 8/ 2019
Description: program to find the area of a circle using the
radius r
 */
// the actual code begins from here
#include <stdio.h>
#define PI 3.1416
float radius;
float area(float r);
int main(void)
{
radius = 10;
printf("Area: %f", area(radius));
return 0;
}

float area(float r) {
return PI * r * r;
}
```

The above code will give the following output:

Area: 314.160000

Now let us analyze each section in the program above.

Documentation: This section contains a multi-line comment describing the code. A single line informing the reader about the beginning of the program.

```
/*
Author: Hammed Adeleye
Date: 25/ 8/ 2019
Description: a program to find the area of a circle using
the radius r
 */
// the actual code begins from here
```

In C, we can create a single line comment using two forward slashes // and we can create multi-line comment using /* */.Comments are ignored by the compiler and are used to write notes and document code.

Preprocessor Directives: This section includes the header file (Link).

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

We are including the stdio. h input/output header file from the C library. This section also contains a constant (Definition).

```
#define PI 3.1416
```

In the above code, we have created a constant PI and assigned 3.1416 to it.

The #define is a preprocessor compiler directive that is used to create constants. We generally use uppercase letters to create constants. The #define is not a statement and must not end with a semicolon (;)

Global declaration: This section contains both variable and function declaration.

```
float radius;
float area(float r);
```

We have declared a variable radius to be of float type and area function, which takes a floating number (i.e., number with decimal parts) as an argument and returns floating number.

main( ) function: This section contains the main() function

```
int main(void)
{
radius = 10;
printf("Area: %f", area(radius));
return 0;
}
```

This is the main() function of the code. Inside this function, we have assigned 10 to the global variable radius declared in the global declaration section. Then we have called the printf() function. The first argument contains "Area: %f," which means we will print the floating number. In the second argument, we are calling the area() function and passing the value of radius (which equals 10) to it. The main function returns an int value and takes no argument; the void in the parenthesis in the main header is optional. The return statement is important since the return type is specified.

Subprograms: This section contains a subprogram, an area() function that is called from the main() function.

```
float area(float r) {
return PI * r * r;
}
```

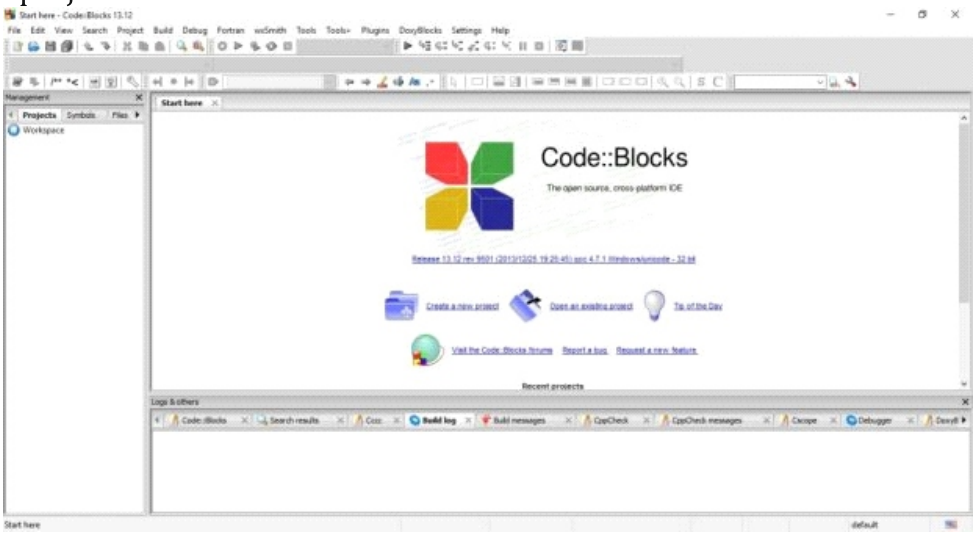
This is the definition of the area() function. It receives the value of radius in variable r and then returns the area of the circle using the following formula PI * r * r.

Compiling and Running a C program

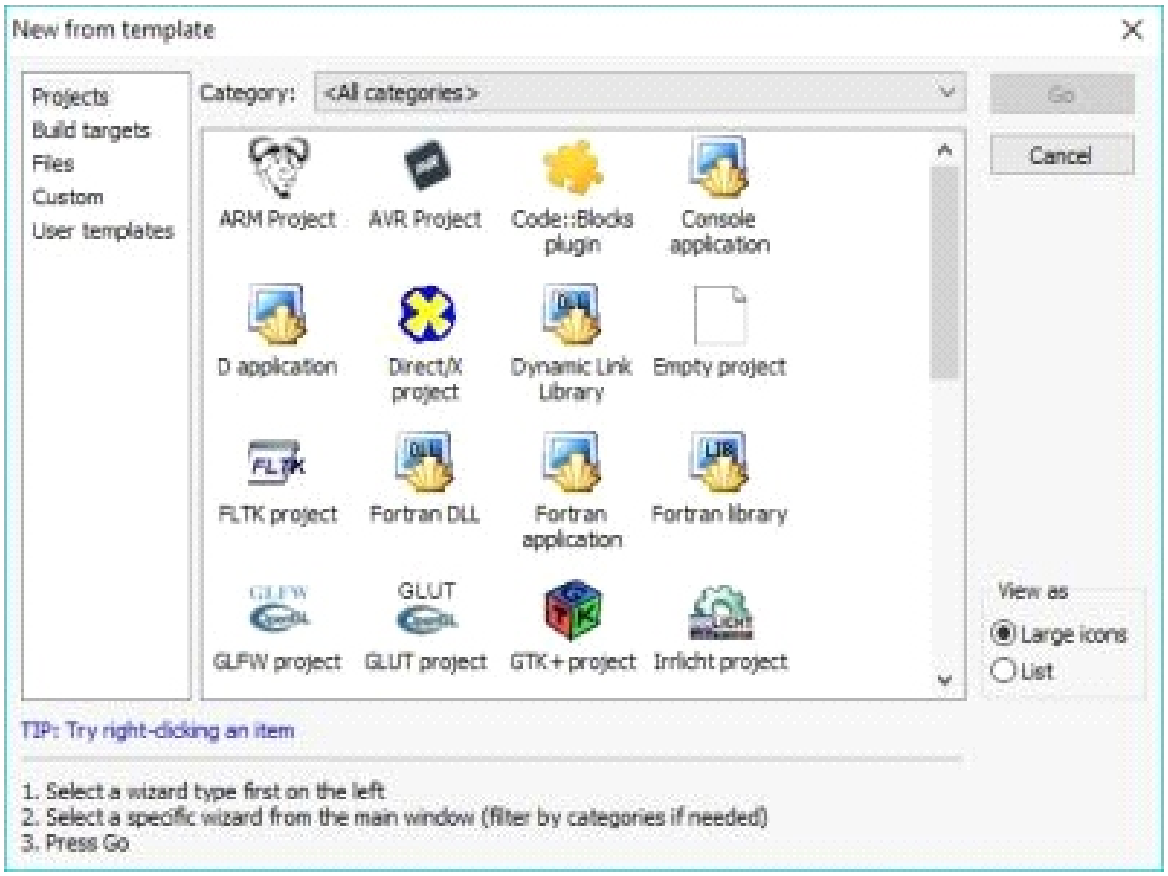
To be able to write and run a C program, a code editor with a compiler is needed. There are various IDE applications available for editing, compiling, and running a C program. In this course, Code Blocks IDE is recommended to compile and run all the code examples. Code blocks is an open-source integrated development environment that contains the GNU GCC C/C++ compiler. It was built to meet the most demanding needs of its users and was designed to be very extensible and fully configurable. It is one of the most common C compilers available to programmers to properly edit, debug, compile, build and run C, C++, and FORTRAN programs. The compiler can be downloaded from https://sourceforge.net/projects/codeblocks/files/Binaries/17.12/Windows/codeblocks-17.12-setup.exe/download .

After installation, follow the following steps to edit, compile, and run your programs.

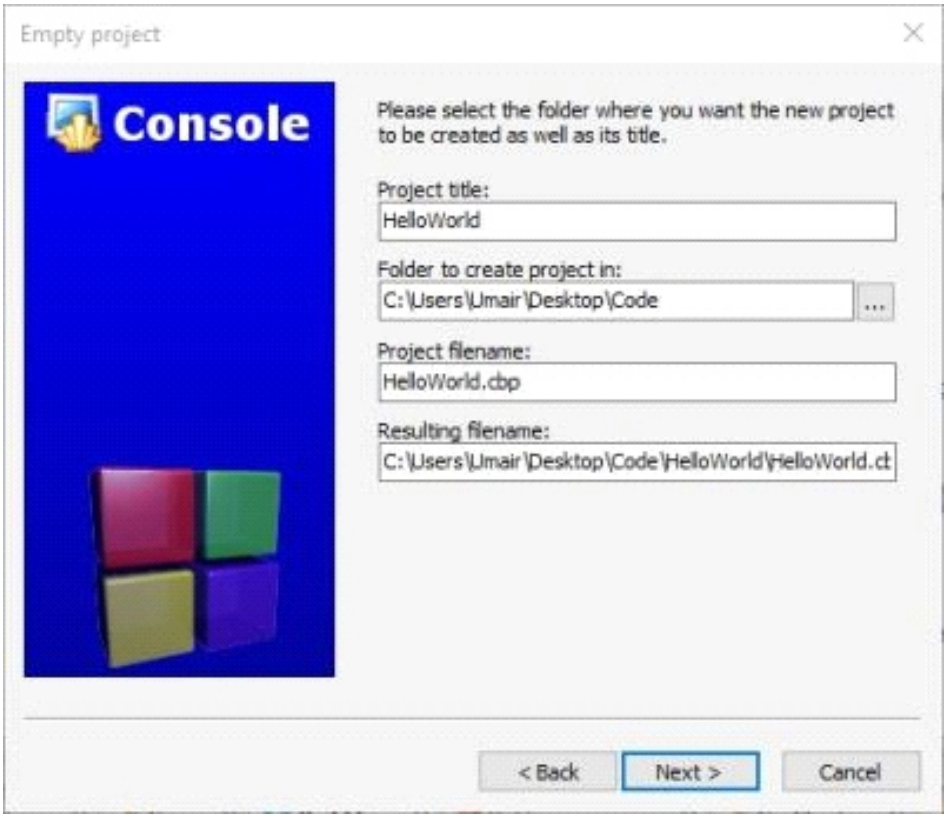
Step 1: Under the tab titles 'Start here', click on 'Create a new project.' This will create a new project folder to store all the different code files.



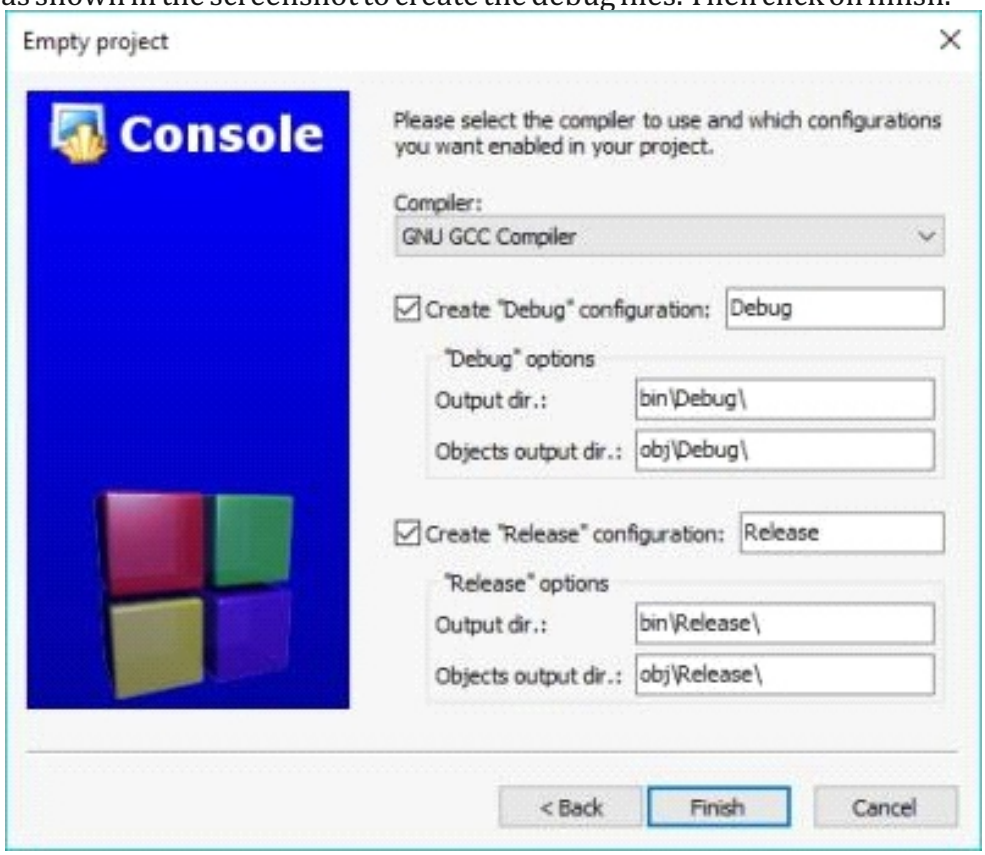
Step 2: Select the Empty project from the different categories displayed in the menu box, then click on Go.



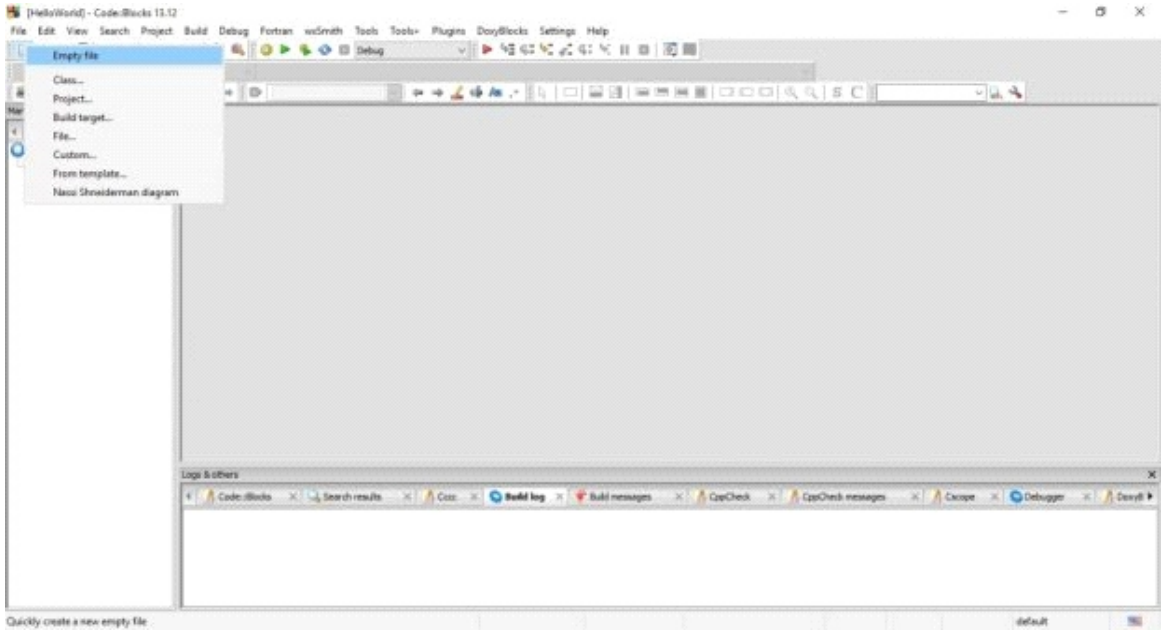
Step 3: Enter the name of your project; this will be the folder where your files will be stored. Also, make a folder on your drive for the code you will write. Then click on next



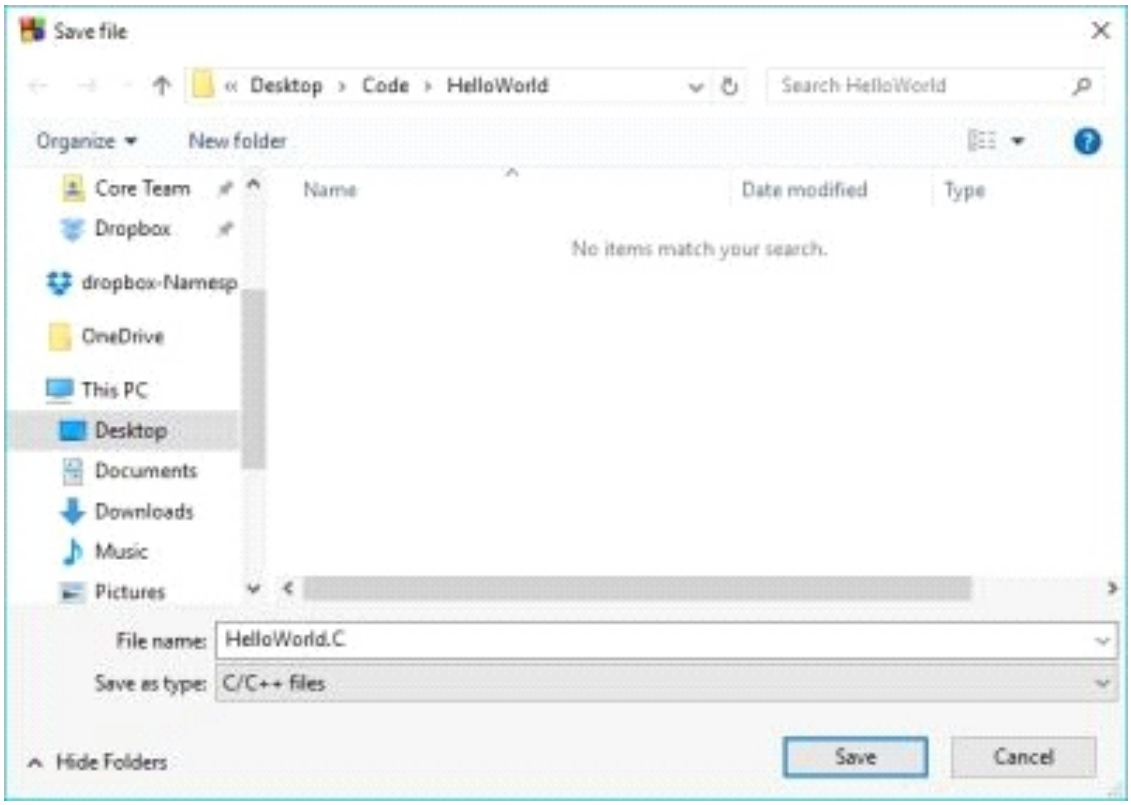
Step 4: Make sure that the selected compiler is the one we just installed. Check the boxes as shown in the screenshot to create the debug files. Then click on finish.



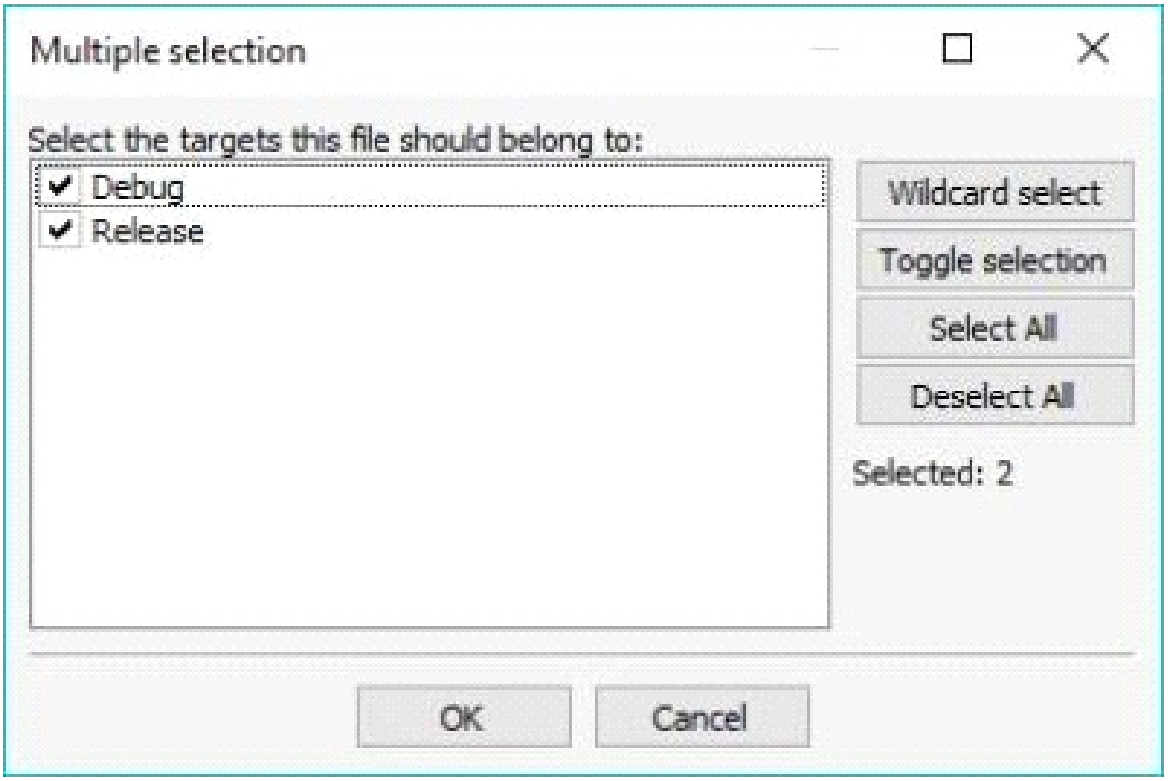
Step 5: From the topmost toolbar, select the new file icon and create an empty file. This is where you will be writing the code. If you cannot see any toolbars, head on to the 'View' drop-down, select toolbars, and select Main and Compiler, the toolbars should now appear on your screen. Similarly, if you can't see the Manager window on the left and the Log window at the bottom, you can select to view them from the 'View' drop-down.



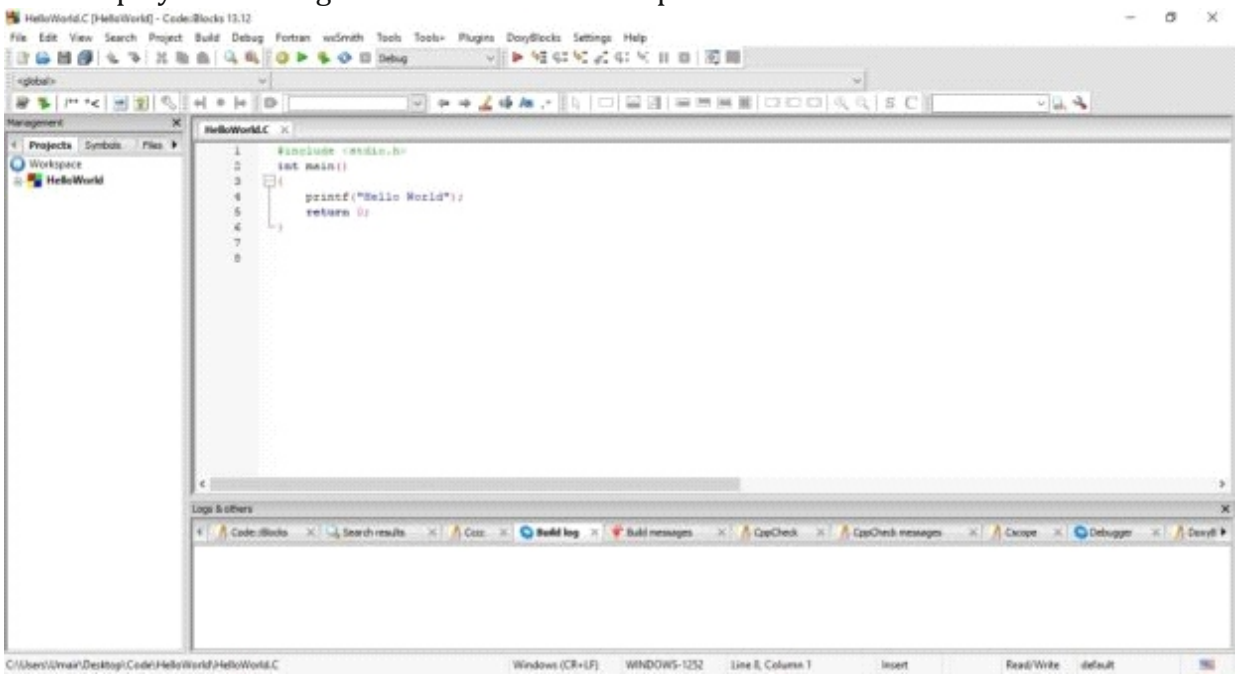
Step 6: Save the file with the extension '.c.' and your desired filename, then click on save. Notice how the default folder is the project folder we created in step 2.



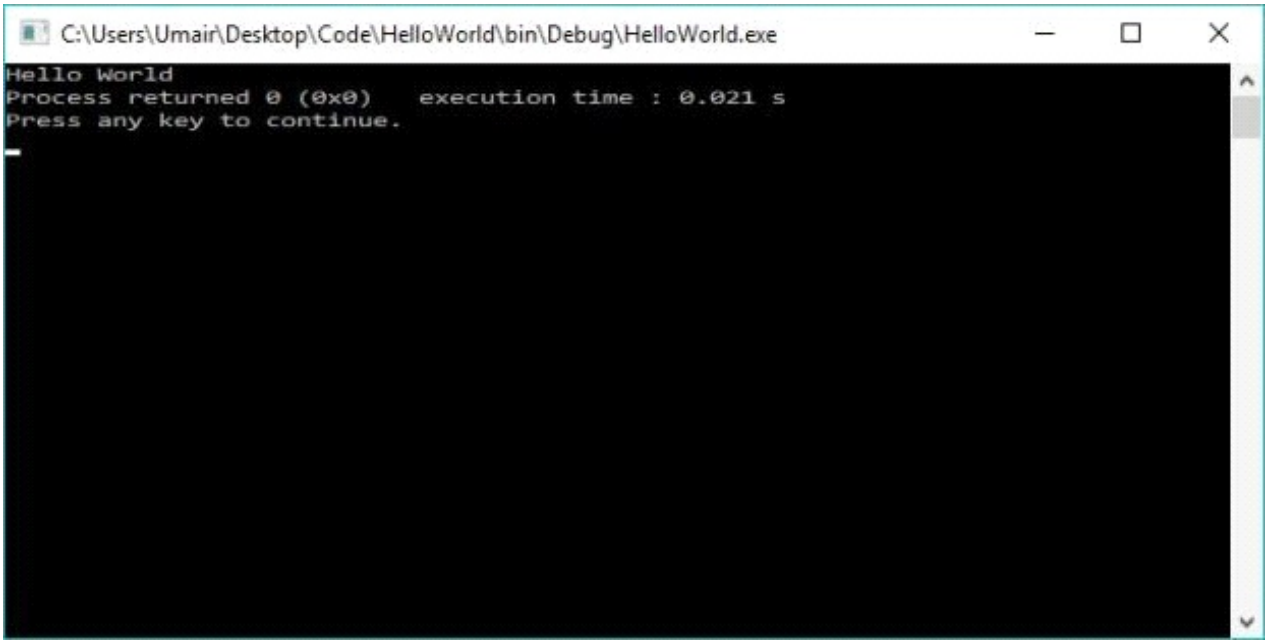
Step 7: Check the boxes shown below to include the files in the Debug folders and click on OK.



Step 8: Write the code for Hello World and click the Build and Run icon located in the toolbar on the top. The icon has a yellow gear and a green play logo. This will compile and immediately run your program. If there is any error during compilation, they will be displayed in the log toolbar under the code space



Step 9: You have successfully written, compiled, and run your first C program. .



Now I believe you can run our first program area.c to test your understanding of these steps.

All codes written henceforth would be run through this process.

Summary

In this unit, I have taught you the following:

- A statement is a command that instructs the computer to take a specific action and forms the part of your program that can be executed.
- There are four types of statements that can be encountered in C programs: Type declaration, Input/Output, Expression and Control statements
- By composition, every C statement can be single, compound or null
  A program in C is composed of the function organized logically in which one of them must be main() function.
- A complete program in C is structured to contain the following sections or parts in order: Documentation, Preprocessor Directives, declarations, Main functions, and subprogram.
- C programs can be edited, compiled, debugged, and run using Code Blocks IDE.

Self-Assessment Questions

SAQ

1. Define a Statement?
2. Mention and explain the various types and categories of a statement in C
3. With an appropriate code and example, explain the basic structure of a C program

Tutor Marked Assessment

- Give two examples, each of the following types of statements: expression statement, control statement, input/output statement, and declarative statement.
- Write a simple program in C to calculate and print the average age of three students in a class

Activity

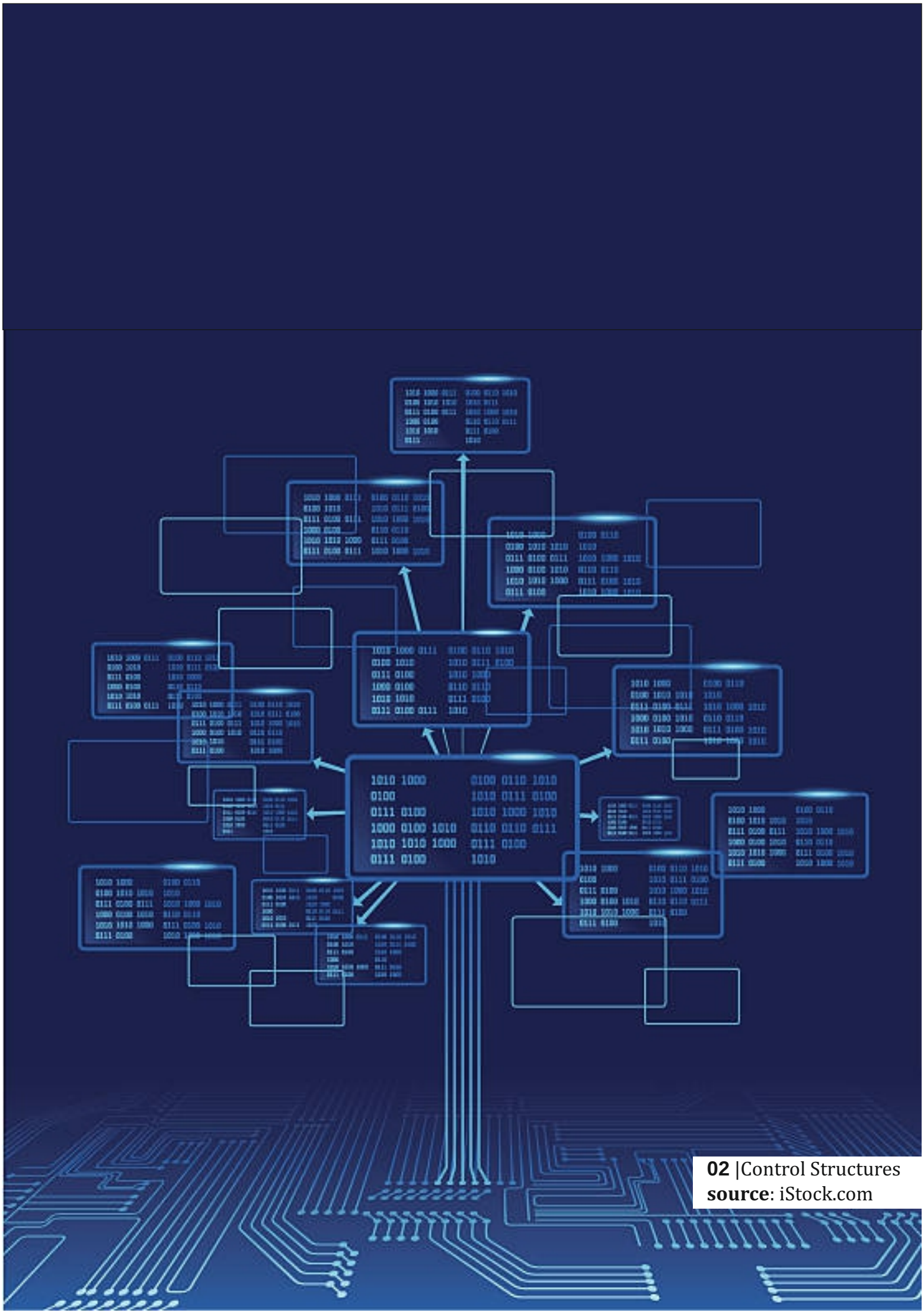
- What is a Statement?
- Design a program in C to calculate the average age of five teachers in a staff room

Further Reading

- https://en.wikibooks.org/wiki/C_Programming/Statements
- https://www.dyclassroom.com/c

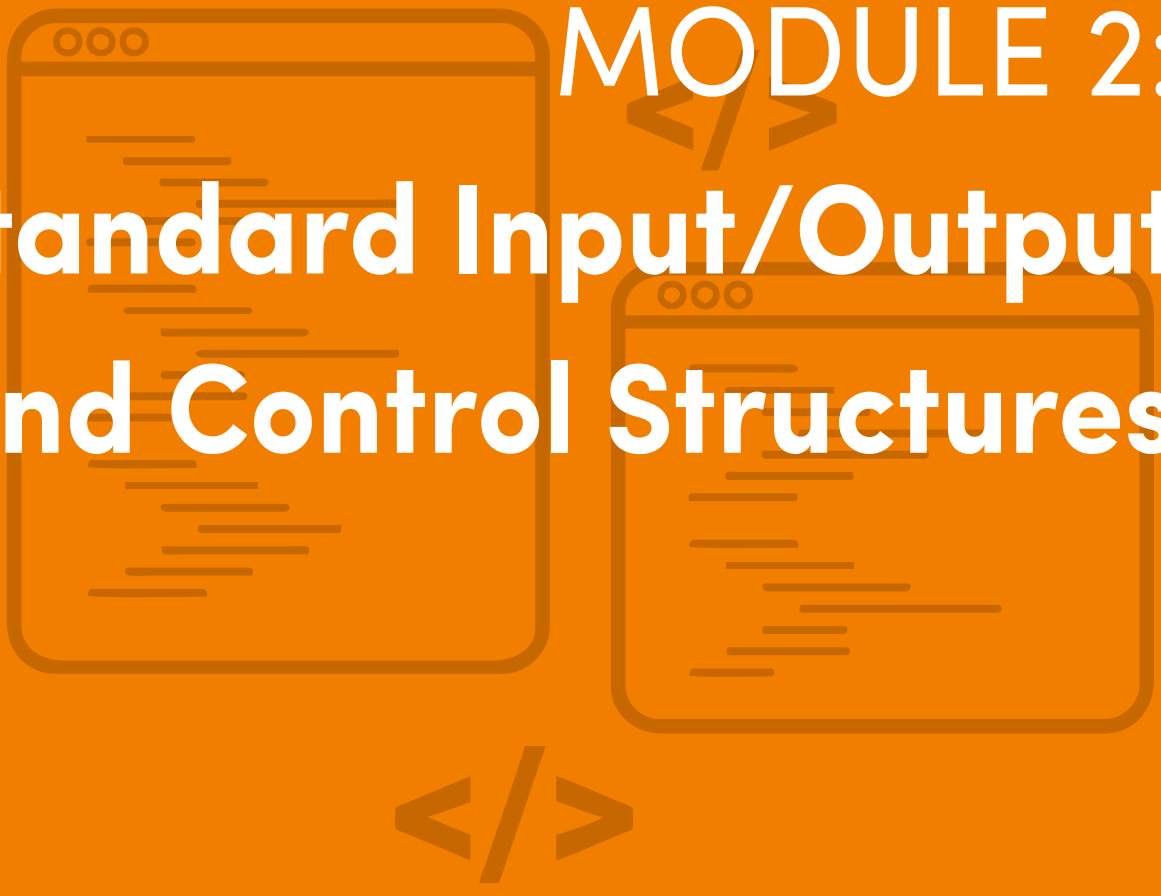
References

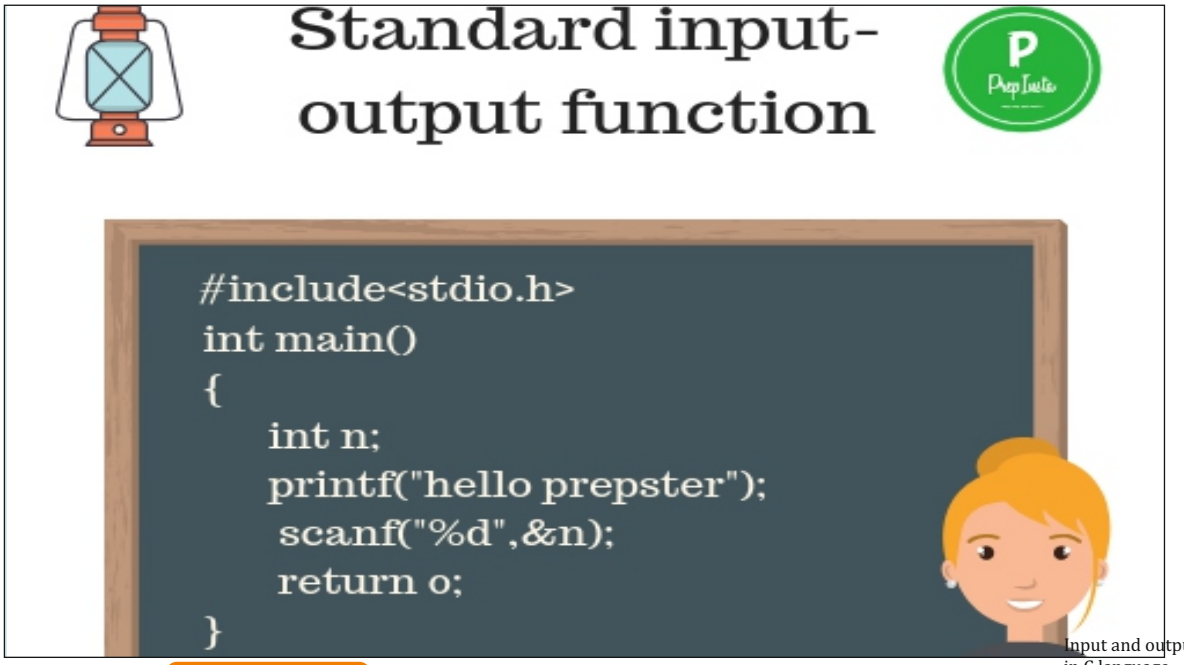
- Brian W. Kernighan Dennis M. Ritchie (2000).
- C Programming Language, the ANSCI C second edition.
- Deitel, P. J., and Deitel, H.(2006).
- Prentice Hall, ISBN 978-0132404167
- C-How to Program (5th Edition).



02 |Control Structures
source: iStock.com

MODULE 2:
Standard Input/Output and Control Structures





Input and output functions in C language
source: prepinsta.com

UNIT 1

Input/Output Functions

Introduction

In this unit, I will introduce you to the concepts of input and output techniques in C. It describes the functions used to read in data from keyboard or files and write out information to the output screen or files. It presents the basic character and string-based input/output functions and how they are used. It further focuses on the commonly used formatted input/output function with program examples.

Learning Outcomes

At the end of this unit, you should be able to:

1. Describe the character and string-based input functions
2. Write simple programs that read in data and display results on the screen
3. Write programs using formatted input/output functions

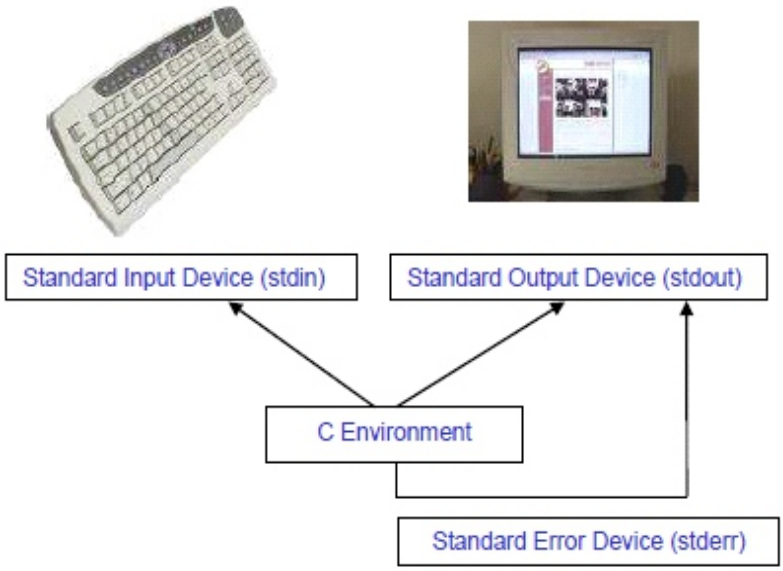
Input /Output Streams in C

1 min

The C environment connects us to the system's standard input and output devices via a buffer often called streams. The buffers are used to send data to and from the C program. A stream is a source or destination of data that may be associated with a disk or other peripheral. A stream is connected to a file or device by opening it; the connection is broken by closing the stream.

There are three streams or buffers connected to the C environment to aid the reading and writing of data by programs. Theses streams are stdin, stdout, and stderr. The connection between them and the C environment is depicted in figure 2.1.1.

- The C environment and the input and output operations are shown in the following figure.



stdin refers to the keyboard, stdout, and stderr refers to the VDU, and input/output takes place as a stream of characters. C provides functions to use these streams to read in and write out data in the studio or library. The library must be included in a program to allow the actions, and when the program begins execution, the three streams stdin, stdout, and stderr are already open.

Character based Input/Output Functions [SAQ1]

1 min

The character-based input and output functions are used basically to read in a character at a time. They read in a character and return the integer equivalent of the

character read. The character-based input and output functions are given below:

INPUT: intgetc(FILE *stream): returns the next character of a stream as an unsigned char (converted to an int), or EOF if the end of file or error occurs.

intgetchar(void):returns the next character of a stream as an unsigned char (converted to an int), or EOF if the end of file or error occurs. Unlike getc(), getchar() takes no argument.

OUTPUT: intputc(int c, FILE *stream): writes the character c (converted to an unsigned char) on stream. It returns the character written or EOF for error.

intputchar(int c): writes the character c (converted to an unsigned char) on stream. It returns the character written or EOF for error. Unlike putc() that takes two arguments, putchar() takes one.

```
/* a program to accept and display a character*/
#include <stdio.h>
int main ()
{
charAlph;
Alph = getc(stdin); // accept a character
fflush (stdin); //to clear the stdin buffer. Its optional
putc(alph,stdout);
return 0;
}
```

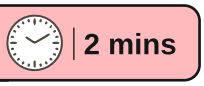
Run the program on your code blocks to see how it works.

Another program that uses the getchar() and putchar() functions is given below:

```
/* a program to input and display a character using the
functions putchar() and getchar()*/
#include <stdio.h>
int main ()
{
char c;
c = getchar(); // read a character
putchar(c); // display the character read
return 0;
}
```

Note the way the input/output functions differ in the way they are used and the number of arguments they are required to work.

String Based Input/Output Functions[SAQ2&3]



String-based input/output functions are used to read sequence or a line of characters at a time. The functions provided by C for string-based input/output are:

INPUT: char *gets(char *s): reads the next input line into the array or pointer variable s; it replaces the terminating newline with '\0'. It returns s, or NULL if the end of file or error occurs. It can be used to read in a string of characters delimited by a newline character. In other words, the string read by this function can contain embedded spaces. It is often used to read texts line by line

OUTPUT: int puts(const char *s): writes the strings and a newline to stdout. It returns EOF if an error occurs, non-negative otherwise. It can as well take string literals such as display prompt to inform the user about certain actions to be taken.

The following program example uses gets() and puts() functions

```
/* A program to accept and display string given by the
user*/
#include <stdio.h>
int main ()
{
charStr[21];
puts("Enter a string maximum of 20 character"); //display
prompt
gets(Str); //Read in String
puts(Str); // display the input string on the screen
return 0;
}
```

Another program example that requests for the name of the user and displaysthe welcome message:

Hello. How are you?

(name)

```
/* A program to accept user's name and welcome the user by
displaying the supplied name*/
#include <stdio.h>
int main ()
{
char name[26]; // of 25 maximum characters
puts("Welcome!  Please  enter  your  name:  maximum  25
characters");
gets(name); //Read in the user's name
puts("Hello. How are you?);
puts(name);
return 0;
}
```

Formatted Input/Output Functions

Formatted input and output functions are used to read or write a specific type of values as specified by the programmer. They allow us to read all types of data in a user-defined format. They often contain conversion characters that instruct the system to convert the stream of characters to the specified types. They provide more programming flexibility and power than other input/output functions. They are defined in the stdio.h headers like other input/output functions.

Formatted Output function

C language provides the printf() function that can be used to output information (both data from variables and text) to standard output in a user formatted manner. It has the following definition:

intprintf(const char *format, ...)

It converts and writes output to stream under the control of the format. The return value is the number of characters written, or negative if an error occurred.

It takes a format string and lists parameters for output(represented as ... in the definition) it is used as:

printf(format string, arg1, arg2, ...);

e.g. printf("The result is %d and %d\n", a, b);

The format string contains two types of objects: ordinary characters, which are copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive argument to printf(). Each conversion specification begins with the character % and ends with a conversion character. Between the % and the conversion character there may be, in order:

A number specifying a minimum field width. The converted argument will be printed in a field at least this wide, and wider if necessary. If the converted argument has fewer characters than the field width, it will be padded on the left (or right, if the left adjustment has been requested) to make up the field width. The padding character is normally space,

A period, which separates the field width from the precision. A number, the precision that specifies the maximum number of characters to be printed from a string, or the number of digits to be printed after the decimal point for eor f conversions, or the number of digits to be printed for an integer (leading 0s will be added to make up the necessary width).

A length modifier h, l (letter ell), or L. ``h'' indicates that the corresponding argument is to be printed as a short or unsigned short; ``l'' indicates that the argument is a long or unsigned long, ``L'' indicates that the argument is a long double.

The conversion characters and their meanings are given below:

c Single character

d Signed decimal integer

x Hexadecimal number

f Decimal floating point number

e Floating point in "scientific notation"

s Character string (more on this later)

u Unsigned decimal integer

% Just print a % sig

If the character after the % is not a conversion character, the behaviour is undefined.

Note that there must be one conversion specifier for each argument being supplied to printf() function and ensure you use the correct specifier for the type of data you are printing. In otherwords, the type and number of specifiers must match and march with the type and number of the list of arguments supplied.

A code example on the use of printf is given below:

```
/* A program on the use of printf() function*/
#include <stdio.h>

int main() {

int ten=10,x=42;

char ch1='o', ch2='f';

printf("%d%% %c%c %d is %f\n", ten,ch1,ch2,x, 1.0*x / ten );

return 0;

}
```

What is the output of the program?

Other versions of printf() function are fprintf() and sprintf(). They are both equivalents in terms of definition. While fprint() is used to print to a file, sprint() is used to print to a string or an array of characters.

Formatted Input Function

C provides the scanf() function that deals with formatted input conversion. It has the following definition:

intscanf(const char *format, ...)

It reads from a stream under control of format and assigns converted values through subsequent arguments, each of which must be a pointer. It returns when a format is exhausted. Scanf() returns EOF if the end of the file or an error occurs before any conversion; otherwise, it returns the number of input items converted and assigned.

The format string usually contains conversion specifications, which are used to direct interpretation of input. The format string may contain:

·Blanks or tabs, which are not ignored.

·Ordinary characters (not %), which are expected to match the next non-whitespace character of the input stream.

·Conversion specifications, consisting of a %, an optional number specifying maximum field width, an optional h, l, or L indicating the width of the target, and a conversion character.

A conversion specification determines the conversion of the next input field. Usually, the result is placed in the variable pointed to by the corresponding argument (often supplied by the use of & with the variable name to return the address of the variable. An input field is defined as a string of non-whitespace characters; it extends either to the next white space character or until the field width, if specified, is exhausted. This implies that the scanf will read across line boundaries to find its input since newlines are white space (White space characters are blank, tab, newline, carriage return, vertical tab, and formfeed).

The conversion character indicates the interpretation of the input field. The corresponding argument must be a pointer. The legal conversion characters are given below:

d decimal integer; int*

i integer; int*. The integer may be in octal (leading 0) or hexadecimal (leading 0x or 0X).

o octal integer (with or without leading zero); int *.

u unsigned decimal integer; unsigned int *.

x hexadecimal integer (with or without leading 0x or 0X); int*.

c characters; char*. The next input characters are placed in the indicated array, up to the number given by the width field; the default is 1. No '\0'is added.

s string of non-whitespace characters (not quoted); char *, pointing to an array of characters large enough to hold the string and a terminating '\0' that will be added.

e,f,g floating-point number; float *.

n writes into the argument the number of characters reads so far by this call; int *. No input is read. The converted item count is not incremented.

[...] matches the longest non-empty string of input characters from the set between brackets; char *. A '\0' is added. []...] includes] in the set.

Examples:

```
      scanf ("%d", &x);        /* reads a decimal integer */
      scanf  ("%f",  &rate);     /*  reads  a  floating  point
value */
```

The ampersand (&) is used to get the "address" of the variable. All the C function parameters are "passed by value". So if we used scanf("%d",x) instead, the value of x is passed. As a result, scanf will not know where to put the number it reads.

It is also possible to read more than one variable at a time, for example:

```
int n1, n2;  float f;
scanf("%d%d%f", &n1, &n2, &f);
```

While supplying the numbers, white spaces are used to separate numbers since the scanf() stops reading a numeric value immediately space is encountered.

10 20.3

It is also possible to use other characters or strings to separate the numbers in the format string, e.g.:

```
scanf("value=%d,ratio=%f", &value,&ratio);
```

However, you must provide the input along with the specified characters or strings like: value=27,ratio=0.8

Summary

In this unit, I have taught you the following:

- Inputs data and Output information are read and written in a sequence of characters using the stdin, stdout and stderr buffers or streams
- The character-based input/output functions are getchar(), getc(), putchar() and putc()
- Line of strings can be read and written using the string-based input/output functions gets() and puts()
- Formatted input and output tasks can be handled with the functions scanf() and printf()

Self-Assessment Questions

SAQ

1. List and explain the use of character and string-based input functions
2. Write a program that requests for name and address of the user separately and display the information on the screen.
3. Write a program to read in three integer number, find their average and display the result on the screen using formatted input and output functions

Tutor Marked Assessment

- Illustrate how to read into multiple variables at a time using the scanf() function
- You are faced with a programming task that reads input that contained embedded blank spaces in the input and output data. Suggests the appropriate input/output function to be used in this case. Justify your answer

Further Reading

- http://www.javatpoint.com/c-programming-language-tutorial

References

- Brian W. Kernighan Dennis M. Ritchie (2000).
- C programming Language, the ANSCI C second edition.Printence Hall.
- Deitel, P. J., and Deitel, H.(2006).
- C-How to Program (5th Ed).
- Prentice Hall, ISBN 978-0132404167.

Control Statements
source: educba.com

UNIT 2

Selection Control Statements

Introduction

I welcome you to unit 2, Selection Control Statements. This is the unit where I will discuss the control statement, its definition, characteristics, and types. Specifically, it focuses on teaching you different selection control statements in C, their description, construct, evaluation steps, and implementation examples to solve practical problems.

Learning Outcomes

At the end of this unit, you should be able to:

1. Define a control structure
2. Describe the four types of control structures in C
3. State the format of three selection control statements in C
4. Implement different selection control statements in C to solve simple and common practical problems.

Control Statement [SAQ1&2]

By default, statements in programming languages execute sequentially in the order in which they are written. This sequential order can be altered by the programmer using the control statements or structures. The control flow statement or control structures of a language determine the order in which other statements in the program are executed. In solving programming problems, we sometimes need to specify that a statement or a group of statements are to be carried out conditionally, only is some conditions are true. Also, we might need to be able to execute a statement or group of statements repeatedly based on certain conditions. Also, sometimes we might want to conditionally stop the execution of a statement or group of statements if some conditions are satisfied. These kinds of situations are handled in C using some kinds of control statements. This section focuses on the different kind of control statements available in C to complete these tasks.

Types of Control Statements

Control statements in C can be of three categories based on the situations mentioned above:

·**Selection Control Statement:** this control statement selects some statements to execute from a sequence of statements based on specified conditions. Statements are executed only when specified conditions are true; otherwise, they are skipped. They are also called Conditional control statements. Selection statements in C include if statement, if-else statement, if-else ladder statement, and switch statement.

·**Iteration Control Statement:** this control statement repeats the execution of some selected statements for a specific number of times or until a condition is met. This is referred to as looping in programming. In C, the iteration control statement includes while statement, a do-while statement, and for a statement.

·**Termination Control Statement:** this type of control statement stops the execution of a block of statements conditionally based on some criteria. They might cause a loop to stop before its stopping criteria are met or cause an exit from a block of statement before control reaches the end of the block. Termination control structure includes the break statement, continue statement, and an exit statement.

·**Branching control statement:** this kind of statement causes conditional or unconditional jump from one part of the program to another part of the program from which execution continues. Anexample isthe goto statement in C and function call.

This unit focuses specifically on the selection control statements and their use.

Selection Control Statements

4 mins

Selection control statement can be implemented in C using:

·if statement

·if-else statement

·Nested if-else statement

·switch statement

Simple if Statement

An if statement consists of a Boolean expression followed by one or more statements. Boolean expressions are expressions that can yield true or false values. If the boolean expression evaluates to true, then the block of statement inside the if block will be executed. If the expression evaluates to false, then the first set of statements after the if block will be executed. Since it does not have a boolean data type, a non-zero value is taken as true, and a zero value is taken as false. Consequently, the if expression can be arithmetic, relational and logical expressions.

Construct:

if (expression)

Stmt;

where Stmt can be null, single or compound statements.

Evaluation steps:

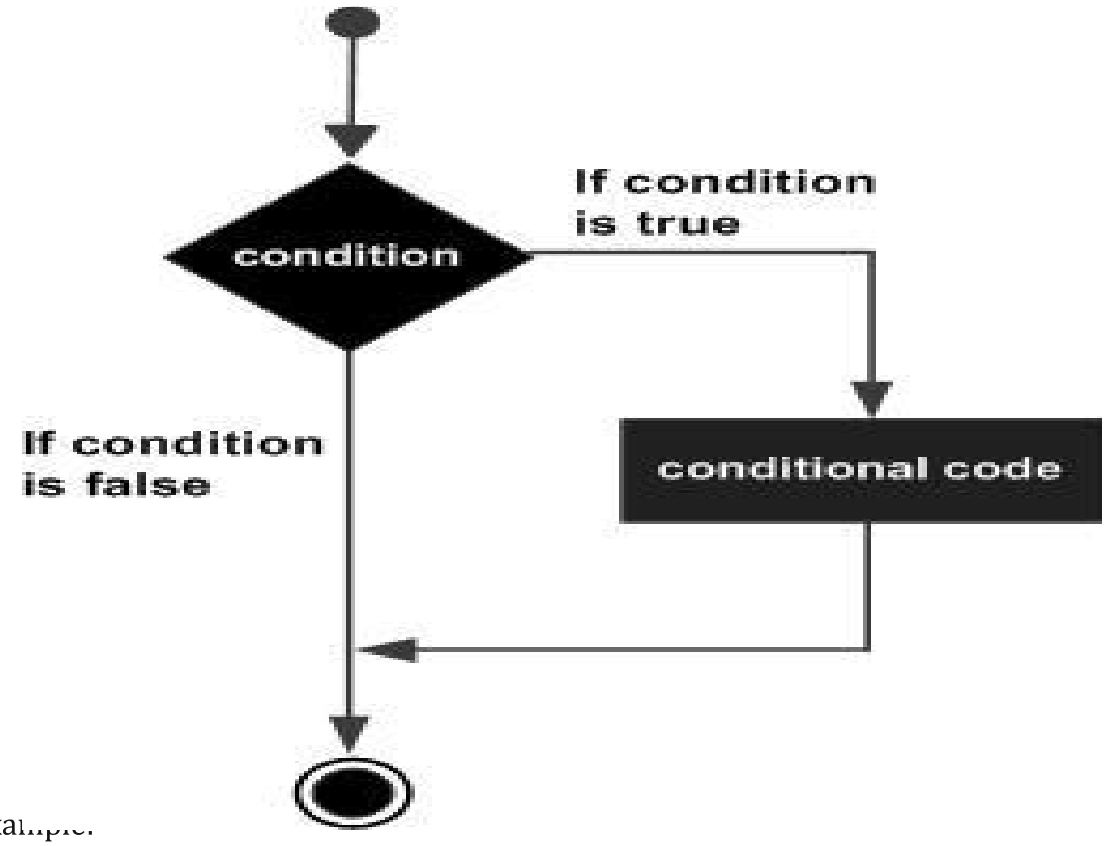
·Evaluate expression

·if an expression is non-zero then:

·Execute Stmt

·Otherwise, go to next statement after Stmt

The if statement is depicted in the flow chart below:

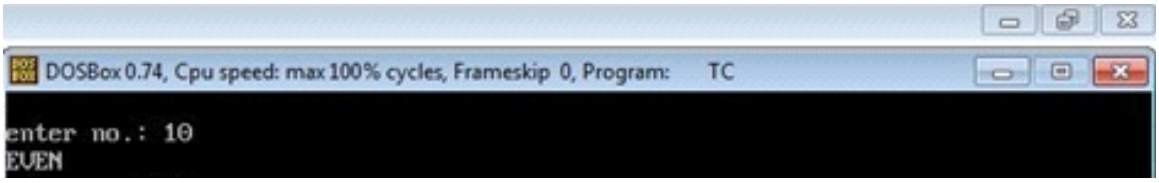


Example:

Consider the following program that checks the evenness of the number supplied by the user.

```
/* A program to check if number supplied by the user is even
*/
#include <stdio.h>

# include <conio.h>

int main() {

inta;

printf("enter a Number:”);

scanf( “%d”, &a);

if (a%2 == 0)

printf("EVEN”);

getch();

return 0;

}
```

Output:



If-else Statement

An if statement can be followed by an optional else statement that executes when the Boolean expression returns false (zero value). If the boolean expression evaluates to true, then the if block will be executed, otherwise the elseblock will be executed.

Construct:

```
if (expression)

statement1;

else

statement2;

next statement;
```

where Statemet1 and statement2 can be null, single or compound statements.

Evaluation steps:

·Evaluate expression

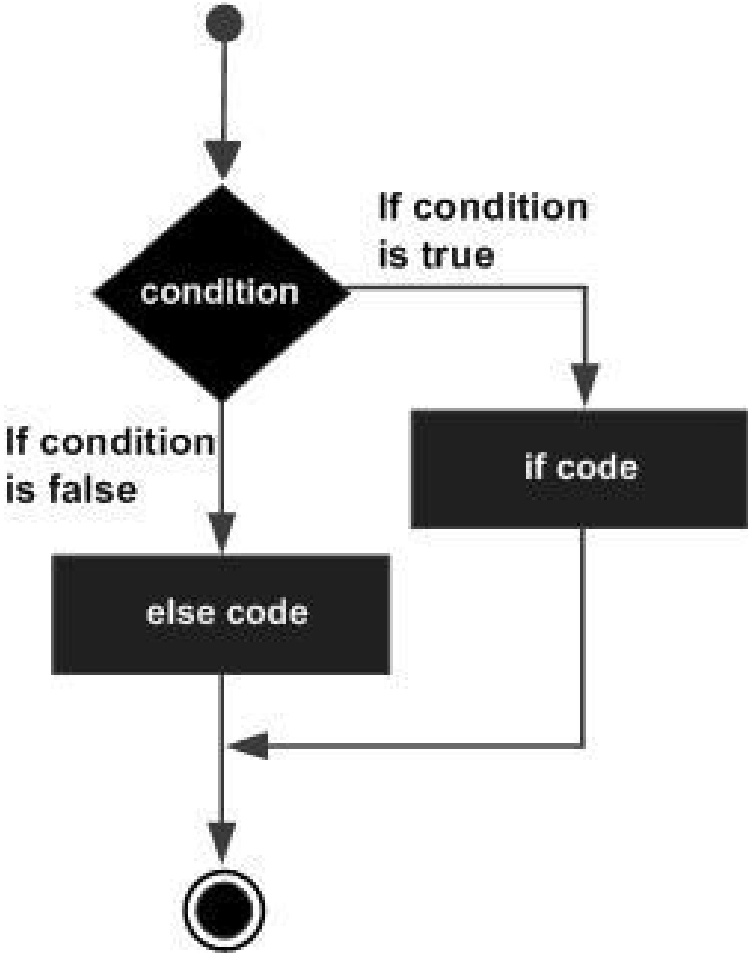
·If expression yields non-zero value then

·Execute statement1

·Otherwise, execute statement 2

·Go to next statement

The if-else statement is depicted in the flow chart below:



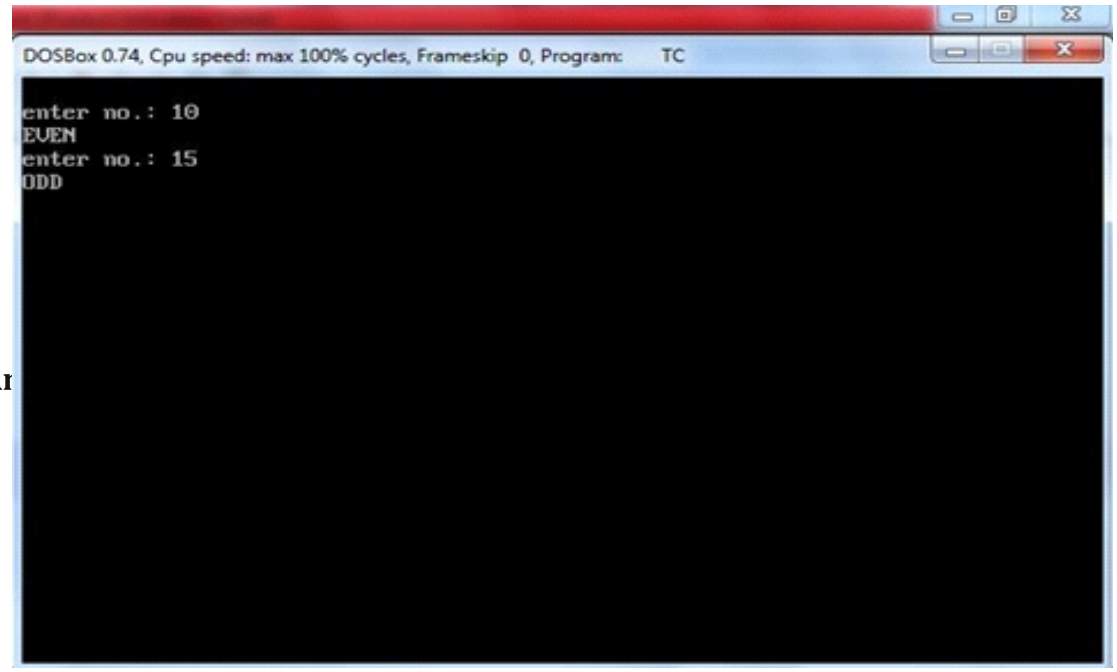
Example:

```
/* A program to check if number supplied by the user is even or
odd */
#include <stdio.h>

int main() {

inta;

printf("enter a Number:”);

scanf( “%d”, &a);

if(a%2 == 0)

printf("EVEN”);
```

Output:

```
else

printf(“ODD”);

return 0;

}
```

Ar




```
#include <stdio.h>
intx,y;
int main ()
{
printf ("\nInput an integer value for x: ");
scanf ("%d", &x);
printf ("\nInput an integer value for y: ");
scanf ("%d",&y);
if (x > y)
      printf ("x is greater than y\n");
else
      printf ("x is smaller than y\n");
return 0;
}
```

Run the code above on your code blocks and study its output

Nested if/ if-else Statement

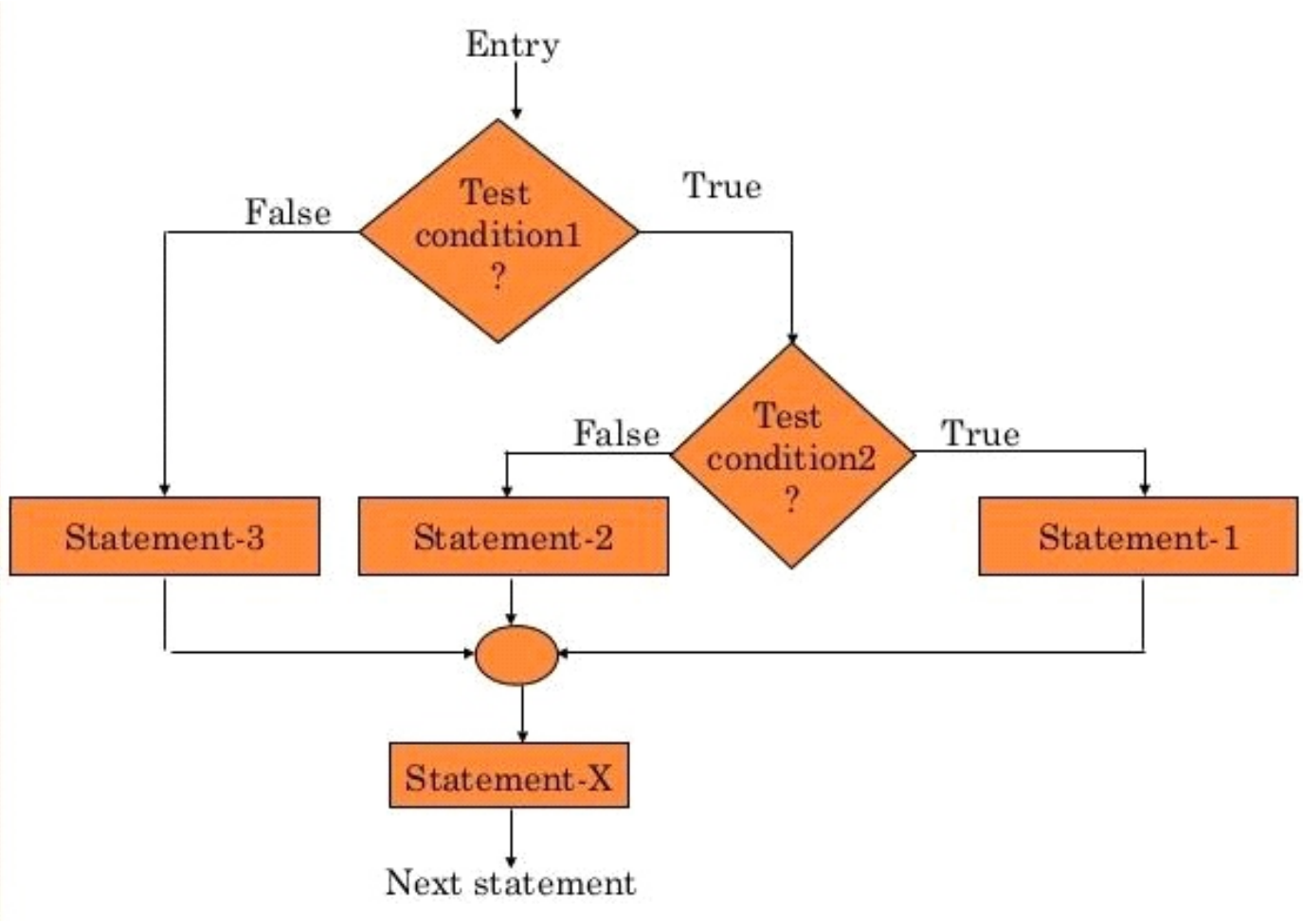
This statement uses one if or if-else statement inside another if or if-else statement. It tests for more than one expression. It executes only the inner if statement if all the expressions return true, the inner else statement is executed if one of the expressions is false and the outer else if none of the expression return true.

Construct:

```
if (expression1){
        If (expresion2)
statement1;
else
statement2;
}
else{
if ( expression3)
statement3;
        else
                statement4;
}
```

where Statemet1, statement2,statement3, and statement4 can be null, single, or compound statements.

Evaluation steps: is depicted in the flow chart below:

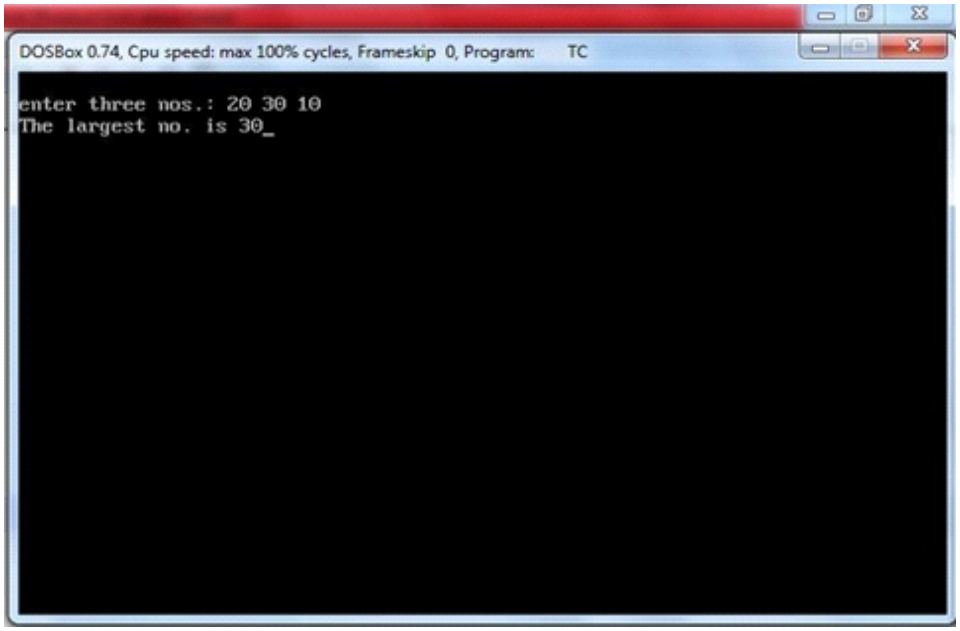



```
/* A program to find and return the largest of three numbers
supplied by the user */
#include <stdio.h>

int main() {

inta,b,c;

printf("\n enter three Number:”);

scanf( “%d %d %d”, &a, &b, &c);

if(a>b){
```

Example:

```
        if(a>c)
                printf("the largest number is %d", a);
        else
                printf("the largest number is %d", c);
   }
else{
        if(b>c)
                printf("the largest number is %d", b);
        else
                printf("the largest number is %d", c);
}
return 0;
}
```

Output:



Try to run the code with a different set of three numbers and see how it performs.

3.2.4 if...else if Ladder Statement

This if...else if statement is used to execute one statement from multiple conditions. It provides multiple options for which only one must be true to execute its statement and other expressions after that are not evaluated. It has the following construct:

Construct:

if (expression)

statement1;

else if (expression)

statement2;

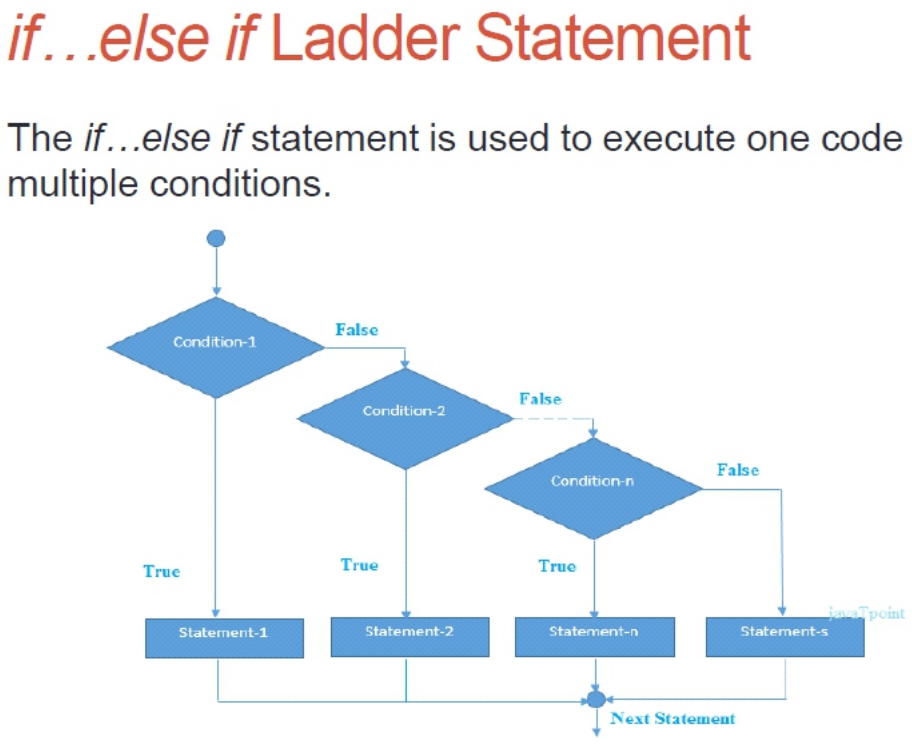
.else

statementn;

next statement;

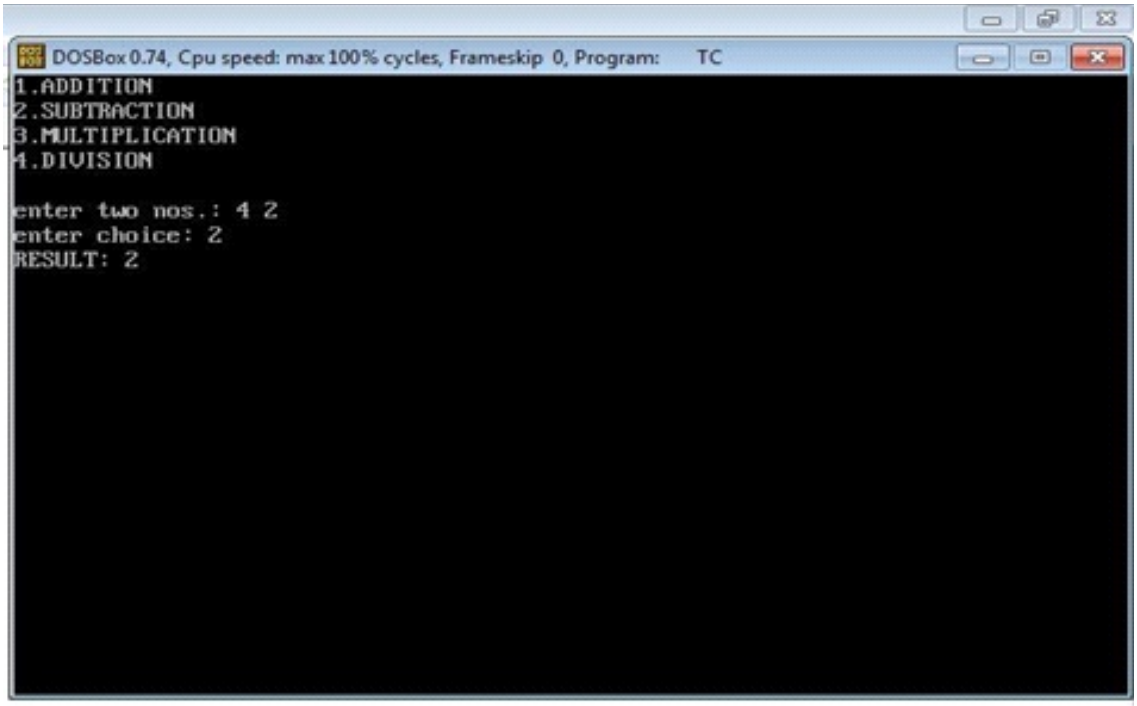
where Statemet1, statement2, to statementn can be null, single or compound statements.

Evaluation steps: is depicted in the flow chart below:



```
/* A program to perform arithmetic operations on two integer
data based on user selection */
#include <stdio.h>

int main() {

intn1,n2,ch;

printf("1.Addition”);

printf("\n2.Substractio”);

printf("\n3.Multiplocation”);

printf("\n4.Division”);

printf("\n enter the two numbers”);

scanf( “%d %d”, &n1, &n2);

printf("\n enter your choice: ”);

scanf( “%d”, &ch);

if(ch==1)

      printf("Result : %d”, n1+n2);

else if(ch==2)

      printf("Result : %d”, n1-n2);

else if(ch==3)

      printf("Result : %d”, n1*n2);

else if(ch==4)

      printf("Result : %d”, n1/n2);

else

      printf("Wrong chice”);

return 0;

}
```

Output:



Switch Statement

A switch statement allows a variable to be tested for equality against a list of values known as a case. A switch statement is preferred to be used to implement multiple choices.

Construct:

```
switch(expression)
{
        case constant_expr1 : statement1
        case constant_expr2 : statement2
        ...
        caseconstant_exprk : statementn
        default : statements
}
```

where Statemet1, statement2, to statementn can be null, single or compound statements.

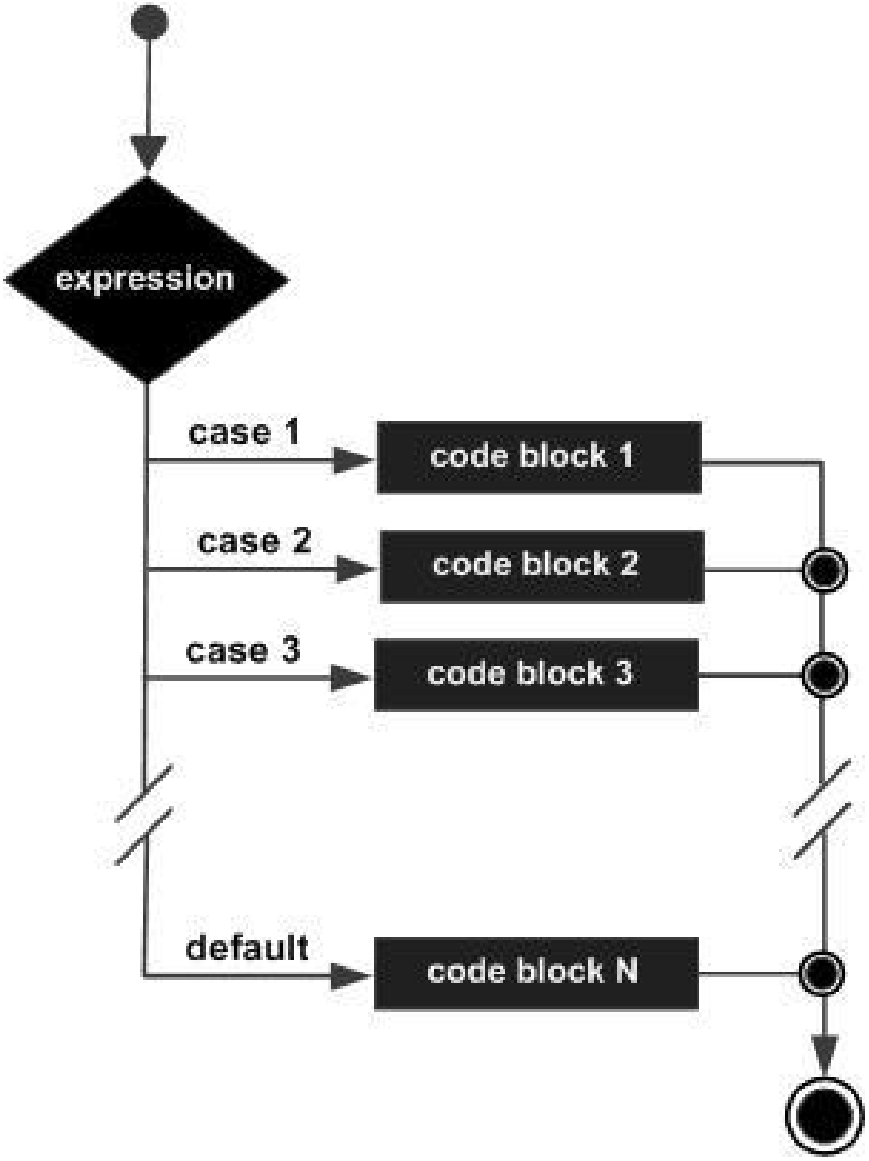
Evaluation steps:

·Expression is evaluated.

·The program jumps to the corresponding constant_expr that equals the value of the expression.

·All statements after the constant_expr are executed until a break (or goto, return) statement is encountered

The control flow for the statement is depicted in the chart below:

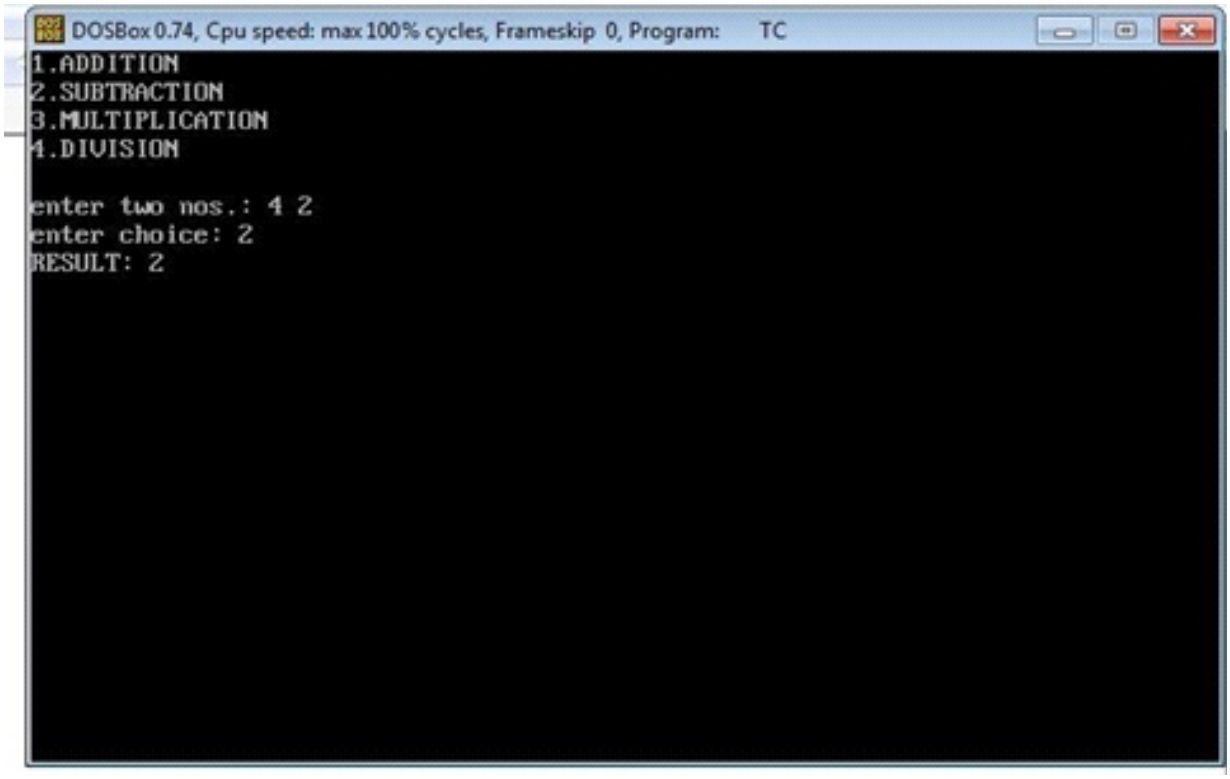


Example:

```
/* A program to perform arithmetic operations on two integer
data based on user selection using switch statement */
#include <stdio.h>

int main() {

intn1,n2,ch;

printf("1.Addition”);

printf("\n2.Substractio”);

printf("\n3.Multiplocation”);

printf("\n4.Division”);

printf("\n enter the two numbers”);

scanf( “%d %d”, &n1, &n2);

printf("\n enter your choice: ”);

scanf( “%d”, &ch);

switch(ch)

  {

       case 1: printf("Result : %d”, n1+n2); break;

case 2: printf("Result : %d”, n1-n2); break;

case 3: printf("Result : %d”, n1*n2); break;

case 4: printf("Result : %d”, n1/n2); break;

default: printf("Wrong chice”);  }

return 0;}
```

Note the break statement after the statement of each case. This is to avoid falling through the problem of the switch statement. With this, only the statement that matches the supplied value in the case would be executed.

Output:



Summary

In this unit, I have taught you the following:

- Control statements are used to control the order of execution of other statements in the program
- In C, there are four categories of control statements: selection, iteration, termination, and branching.
- The selection control statements in C comprises of the if, if-else, nested if, if..else if and switch statements
- Each control statement can be implemented to solve practical, real-life problems.

Self-Assessment Questions

SAQ

1. What is the control structure?
2. List and explain the types of control structures in C
3. Describe the format of each of the selection control statements in C
4. Write a program to find the largest of 5 numbers to be supplied by the user, using an appropriate selection statement

Tutor Marked Assessment

Study the code snipprt below and describe how it works:

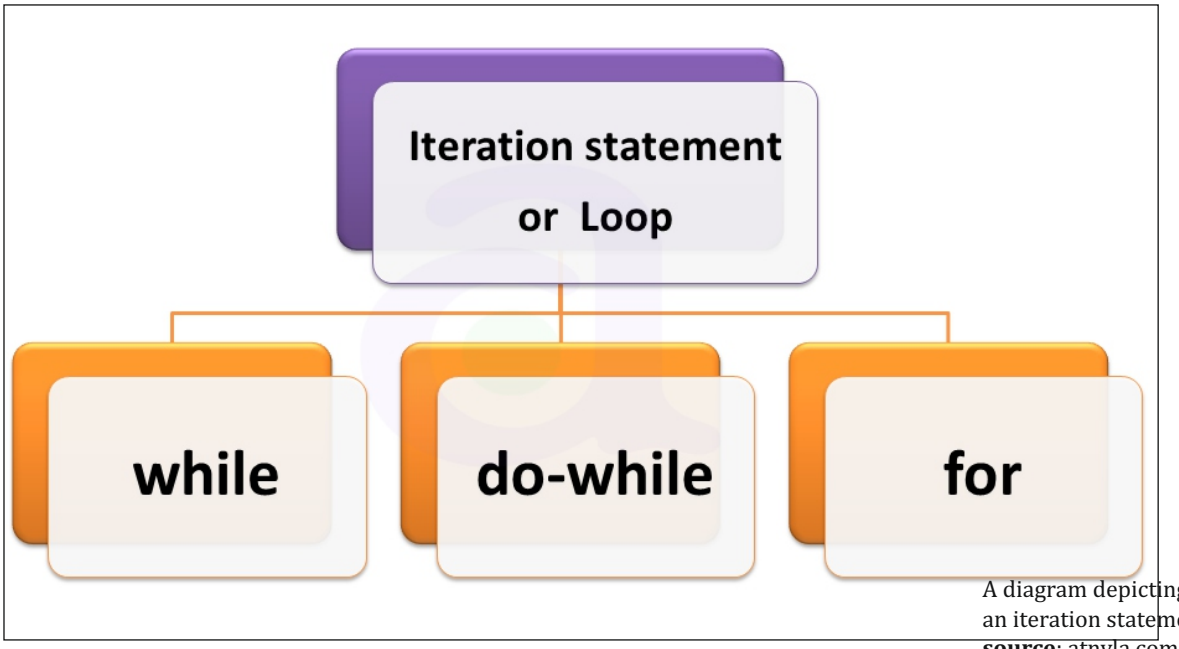
```
int a;
printf("1. Open file..\n");
printf("2. Save file.\n");
printf("3. Save as..\n");
printf("4. Quit.\n");
printf("Your choice:");
scanf("%d", &a);
switch(a)
{
case 1: open_file();break;
case 2: save_file();break;
case 3: save_as();break;
case 4: return 0;
default: return 1;
}
```

Further Reading

- http://www.tutorialspoint.com/cprogramming
- http://www.javatpoint.com/c-programming-language-tutorial

References

- Shola, P. B.(2002).
- Reflect Publishers, ISBN 978-047-174-X
- Learn C-Programming Language (2nd Ed).
- Deitel, P. J., and Deitel, H.(2006).
- C-How to program (5th Ed).
- Prentice Hall, ISBN 978-0132404167



A diagram depicting an iteration statement
source: atnyla.com

UNIT 3

Loops (Iteration Control Statements)

Introduction

This unit discusses iteration control statements used for repetition in programs. Their constructs, evaluation step, and flow chart are highlighted. You will also be introduced to their use in solving programming problems with practical examples.

Learning Outcomes

At the end of this unit, you should be able to:

1. Describe all the loop statement in C
2. Use loop statements to solve programming problems

Loops and its types [SAQ1]

A loop in C language is used to execute a block of code or a part of the program several times. It saves code by allowing programmers to put the block of codes that requires several executions inside a loop block rather than writing the block of codes many times. By this, the program development time is reduced, and more programming power is given to the programmer. There are three types of loop statements in C; these are:

·while loop

·do...while loop

·for loop

Each of these statements is discussed in detail in the following sections.

While Loop[SAQ2]

A while loop in C iterates a statement or block of statements until the condition supplied returns a zero value (false). Condition is given before the code in the form of arithmetic, relational, and logical expression. So the code may execute 0 or more times.

Construct:

```
while (expression){

statements;

}

next statement;
```

where Statementscan be null, single, or compound statements.

Evaluation steps:

Expression is evaluated

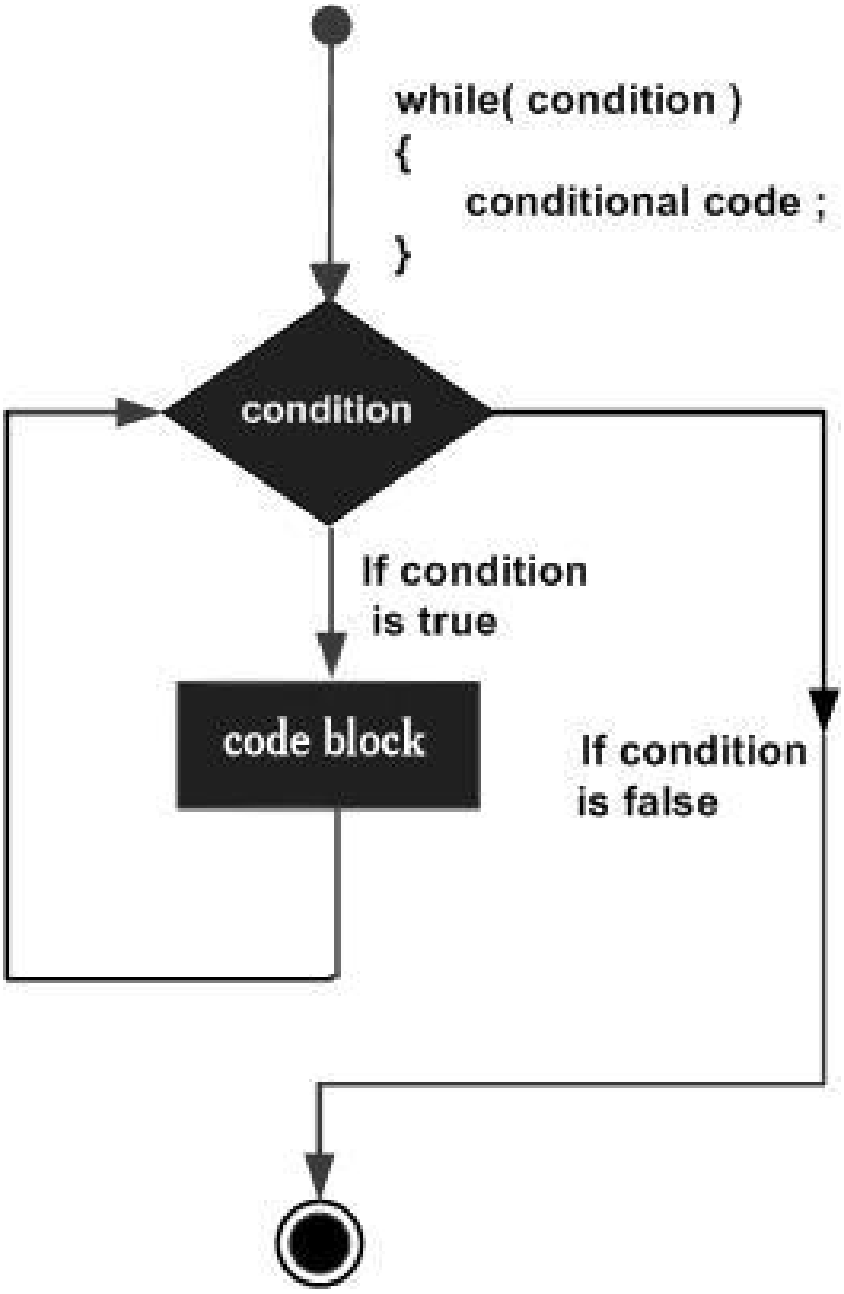
If the expression returns 0, go to step 5 (loop is exited)

If the expression is nonzero,the statement is executed

Go to step 1

Next statement

The execution steps of while loop is depicted in the flow chart below:



The while loop continues to test for the condition and execute the code block until the condition returns zero.

Example

```
/* A program to print the first 10 natural numbers */
#include <stdio.h>

int main() {

intj = 1;

while(j <= 10){
```

```
printf("%d \n", j);
j++;
}
return 0;
}
```

Note the increment in the body of the while block. It is the responsibility of the programmer to include the increment; otherwise, the loop will execute indefinitely. The function of the increment is to move the while loop condition towards the termination point. The increment is applied to the control variable of the while loop (here variable j).

Output:

Example

Source Code	Output

`#include <stdio.h>`
`#include <conio.h>`
 	
`void main()`
`{`
`int i=1;`
 	
`clrscr();`
 	
`while(i<=10)`
`{`
`printf("%d \n",i);`
`i++;`
`}`
 	
`getch();`
`}`
	1
	2
	3
	4
	5
	6
	7
	8
	9
	10

Another example that counts the number of character in the input string is given below

```
#include <stdio.h>
/* count the number of characters in input  string */
intmain()
{
longnc;
```

Run the program on your code blocks and study how it works.

Sometimes the expression in the while loop statement is omitted to allow the loop for an unknown number of times, and the break statement is used within the body to exit the loop when a certain condition is met. More detail about this can be found under the termination control statement.

do…while loop [SAQ2&3]

Do…while statement like the while statement iterates the block of statements until the condition specified returns zero value. However, the condition to be tested is given after the block of statements. So, the statements will execute for at least once whether the condition is true or false.

Construct:

do {

statement;

}
while (expression);

Where statements can be single or compound statements. Note the semi-colon(;) after the expression. Like a while loop, but with condition test at the bottom and always executes at least once.

Evaluation steps:

Execute statement.

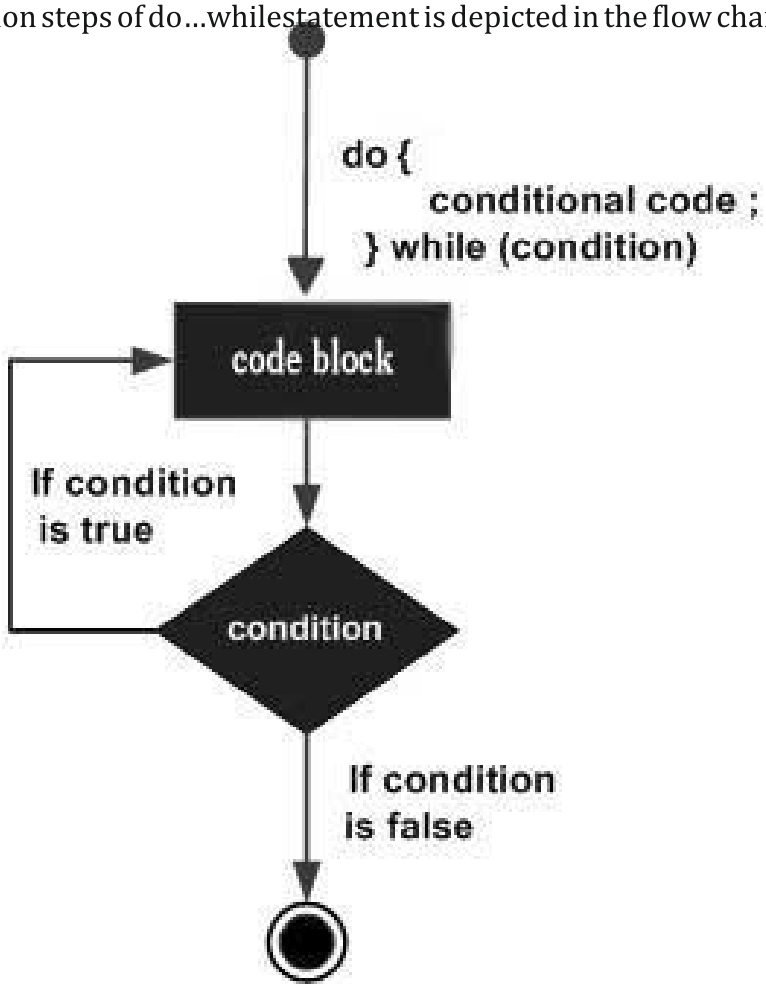
Evaluate expression.

If the expression returns zero value, go to step 5.

If the expression returns on-zero value go to step 1

Execute the next statement after a while.

The execution steps of do...whilestatement is depicted in the flow chart below:



Example:

```
/* A program to print the first 10 natural numbers a do
while version */

#include <stdio.h>

int main() {

int j = 1;

do{

printf("%d \n", j);

j++;

}

while(j <= 10);

return 0;

}
```

Output:

```
#include <stdio.h>
int main ()
{
int selection = 0;
do {
printf ("\n");
printf ("\n1 - Add a Record ");
printf ("\n2 - Change a Record ");
printf ("\n3 - Delete a Record ");
printf ("\n4 - Quit ");
printf ("\n\nEnter a selection: ");
scanf("%d", &selection);
    } while (selection<1 || selection>4);
printf ("You chose %d\n", selection);

return 0;

}
```

Example

Source Code	Output

`#include <stdio.h>`
`#include <conio.h>`
 	
`void main()`
`{`
`int i=1;`
 	
`clrscr();`
 	
`while(i<=10)`
`{`
`printf("%d \n",i);`
`i++;`
`}`
 	
`getch();`
`}`
	1
	2
	3
	4
	5
	6
	7
	8
	9
	10

Another example:

Run the code above on your code blocks tofind outwhat it does.

The while and do...while loops inherently have two drawbacks. They do not ensure that the control variable is initialized and requires that the programmer supplies the increment or decrement statement to drive the loop towards termination. These problems are solved in the third type of loop statement; for.

for Loop[SAQ4]

We use a loop to iterate the code block until the condition specifies return false. It contains initialization, condition, and increment/decrement section given before the code block. So the code may execute zero or more times. With the for loop, the construct has provided a section for control variable initialization and its increment/decrement to force the loop to the termination. It is the most convenient loop statement for programmers and the most important looping structure in C.

Construct:

for(initial ; condition ; increment)

statement;

Initial, condition, andincrementare C expressions and statements can be null, single, or compound statements.

Evaluation steps:

·Initial is evaluated. Usually, an assignment statement.

·Condition is evaluated. Usually, a relational expression.

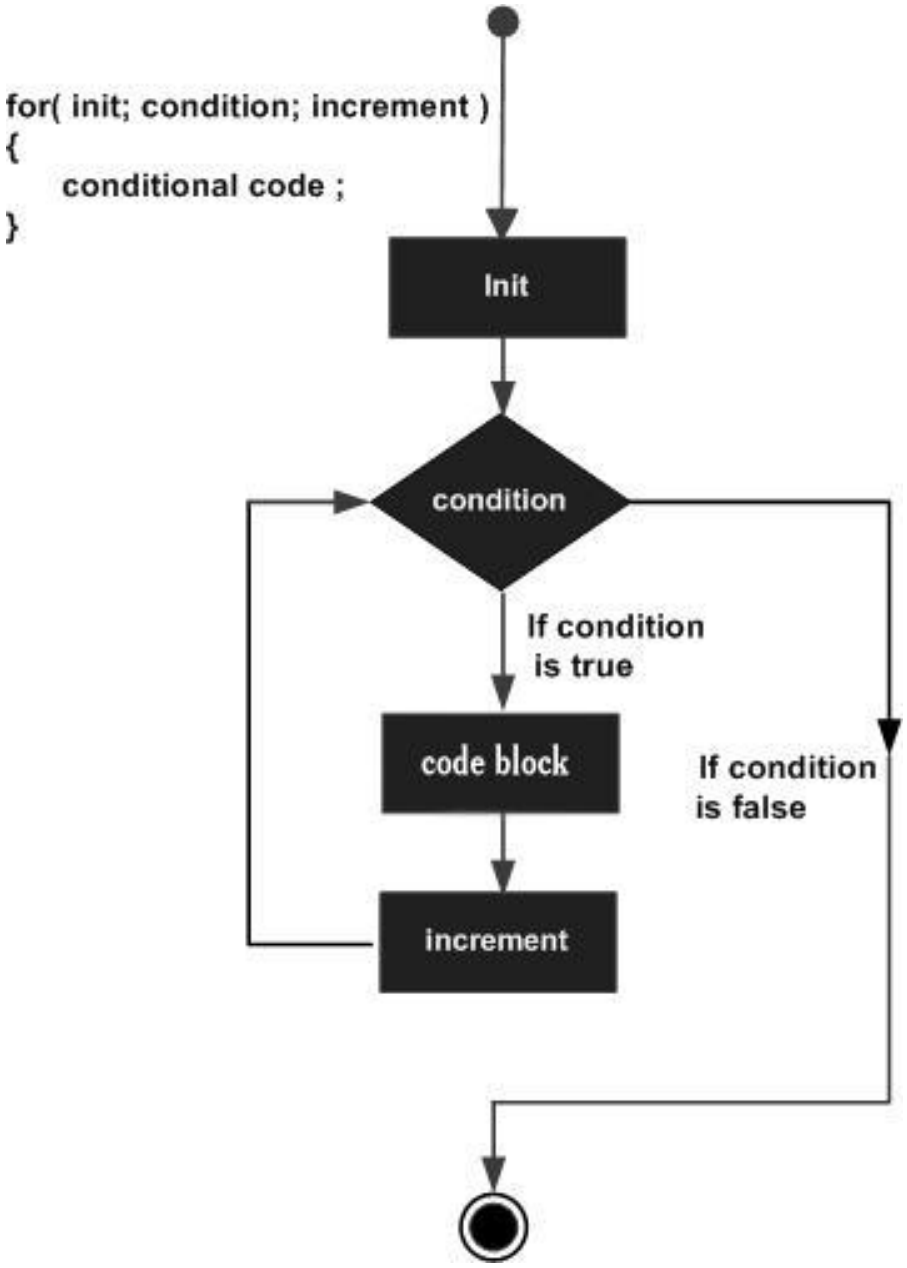
·If condition is false (i.e.,returns 0), fall out of the loop (go to step 6.)

·If condition is true (i.e., nonzero), execute statement

·Execute increment and go back to step 2.

·Execute next statement after for block

The execution steps of for loop is depicted in the flow chart below:




```
/* A program to print the first 10 natural numbers a for loop version */

#include <stdio.h>

int main() {

int j;

for(j=1;j<= 10; j++)

      printf("%d \n", j);

return 0;

}
```

Output:

Example

Source Code	Output

`#include <stdio.h>`
`#include <conio.h>`
 	
`void main()`
`{`
`int i=1;`
 	
`clrscr();`
 	
`while(i<=10)`
`{`
`printf("%d \n",i);`
`i++;`
`}`
 	
`getch();`
`}`
	1
	2
	3
	4
	5
	6
	7
	8
	9
	10

You can observe that the for-loop code is more compact and neater than the other loop statements.

```
#include <stdio.h>
int main () {
intcount,x,y;
intctd;
   /* 1. simple counted for loop */
for (count =1; count <=20; count++)
printf ("%d\n", count);
   /* 2. for loop counting backwards */
for (count = 100; count >0; count--) {
x*=count;
printf("count=%d x=%d\n", count,x);
   }
```

Grammatically, the three components of a for loop(initial, condition and increment/decrement) are expressions. Most commonly, initialand increment/decrementare assignments or function calls, and the conditionis a relational expression. Any of the three parts can be omitted, although the semicolons must remain. If initialor increment/decrementis omitted, it is simply dropped from the expansion. If the test, condition, is not present, it is taken as permanently true. For example, consider the program:

```
/* 3. for loop counting by 5's */
for (count=0; count<1000; count += 5)
          y=y+count;
/* 4. initialization outside of loop */
count = 1;
for ( ; count < 1000; count++)
printf("%d ", count);
   /* 5. very little need be in the for */
count=1; ctd=1;
for ( ; ctd; ) {
printf("%d ", count);
count++;
      ctd=count<1000;
   }
   /*  6.  compound  statements  for  initialization  and
increment */
for (x=0, y=100; x<y; x++, y--) {
printf("%d %d\n", x,y);
   }
return 0;
}
```

Nested for Loop

for statements (and any other Control statements) can go inside the loop of another for a statement. This is known as a nested loop in C.

```
#include <stdio.h>
int main( ) {
int rows=10, columns=20;
int r, c;
for ( r=rows ; r>0 ; r--)
   {
for (c = columns; c>0; c--)
printf ("X");
printf ("\n");
    }
return 0;
}
```

For example:

The for statement print to the screen 20 by 10 matrix of the character "X." The outer for a loop traverse the row while the inner for a loop traverse the column.

Note the following equivalence of for and while statements.

```
for ( ; condition ; )
stmt;
```

is equivalent to

```
while (condition)
      stmt;

for (exp1; exp2; exp3)
stmt;
```

is equivalent to

```
exp1;
while(exp2) {
stmt;
     exp3;
   }
```

Summary

In this unit, I have taught you the following:

- Iteration control statement in C is used to execute a block of statements several times, and they are: while do…while and for a statement. They are also called loops.
- The loop can be nested to perform complex computation such as matrices calculations

Self-Assessment Questions

1. Give a concise description of the while, do... while and for loop statements in C
2. Write a simple C program that reads in 20 number from the user and return their product using do…while and for statements

Tutor Marked Assessment

- In tabular form differentiate between the three-loop statement in C based on syntax and execution steps

Further Reading

- C Tutorial: http://www.tutorialspoint.com/cprogramming
- C Programming Language Tutorial. URL: http://www.javatpoint.com/c-programming-language-tutorial

References

- Shola, P. B.(2002).
- Reflect Publishers, ISBN 978-047-174-X
- Deitel, P. J., and Deitel, H.(2006).
- Prentice Hall, ISBN 978-0132404167
- Learn C-Programming Language (2nd Ed).
- C-How to Program (5th Edition).

```
#include<stdio.h>
main()
{
    int i, j;
    for (i=1; i<=6; i++)
    {
        for (j=1; j<=i; j++)
        {
            printf("*");
        }
        printf("\n");
```

Control Statements
source: educba.com

UNIT 4

Termination Control Statements

Introduction

In this lesson, I will expose you to termination control structures with a focus on how to use them with loops using practical examples. You will be introduced to their constructs, evaluation step, and flow chart. Lastly, the branching control statement "goto" is discussed.

Learning Outcomes

At the end of this unit, you should be able to:

1. Differentiate between break and continue termination statements
2. Solve a programming problem using the combination of loops and termination control statements

Termination and branching Control Structures

The flow of control in any loop can be altered or stopped through the use of the termination control statements. C provides two termination control statements that can be used within loops and switch to alter or stop the flow of execution within the loop. The two control statements are explained in the following section

break Statement [SAQ1]

The break statement is used to exit the loop or switch block immediately; it is encountered. It is useful for stopping on conditions not controlled in the loop condition.

Construct:

break;

For example, consider the for statement below:

```
for (x=0; x<10000; x++) {
     if ( x*x % 5==1) break;
   ... do some more work ...
}
```

The loop terminates if x*x % 5 == 1that is when x equals 6 rather than the stopping criteria specified in the condition of the for loop, i.e. when x equals 9999.

A more practical example is given below in a program that reads and finds the sum of positive numbers only.

Example:

```
/* A program to find and print the sum of only positive
numbers read from the keyboard */
#include <stdio.h>

int main() {

intx, sum = 0;

for (int n = 1; n <= 10; n++) {

             scanf(“%d”, &x)

             if ( x < 0) break;
```

```
            sum += x;
        }
Printf( " the sum is : %d", sum);
return 0;
}
```

The while loop is exited immediately, a negative value is read from the keyboard, and the sum of the positive number read so far is printed on the screen.

To note also is that a break statement is very useful to exit an infinite loop. Consider the example program given below:

```
#include <stdio.h>
int main() {
constintmycard=3;
int guess;
for(;;) // The notation for(;;) is used to create an infinite
        // for loop.while(1) creates an infinite while loop
        //instead.
{
      printf("Guess my card:");
      scanf("%d",&guess);
      if(guess==mycard)
            {
            printf("Good guess!\n");
            break;
            }
      else
            printf("Try again.\n");
}
return 0;
}
```

The program allows the user to guess a number kept in the program continuously until the guess is right. The program breaks out of the infinite loop when the user guesses right.

An example of a break statement used within the loop can be found under the selection control statement discussed earlier.

continue Statement [SAQ1&2]

The continue statement causes the next iteration of the loop to be started immediately. It skips the current iteration and moves controls to the next iteration leaving the remaining code after continue statement unexecuted for that iteration. It must be noted that continue does not cause an exit from the loop but exit from the current iteration.

Construct:

continue;

For example:

```
for (x=0; x<10000; x++) {
     if (x*x % 5 == 1) continue;
     printf("%d ", 1/ (x*x % 5 - 1) );
}
```

Will execute loop when x*x % 5 == 1 (and avoid division by 0). continue statement is very useful when we do not want to include some certain values in our processing or computation.

For example, consider the code below that find the sum of the even number between 1 and 100.

```
/* A program to find the product of even numbers between 1
and 100, then print the result on the screen */

#include <stdio.h>

int main() {

int x, product = 1;

for ( x = 2; x<= 100; x++) {
```

```
            if ( x%2 > 0) continue;

            product*= x;

      }

Printf( "The product is : %d", product);

return 0;

}
```

Run the code on code blocks to check whether it works as expected.Note the difference between break and continue as presented below:

·A break can be used within loop and switch statement while continue can only be used within the loop

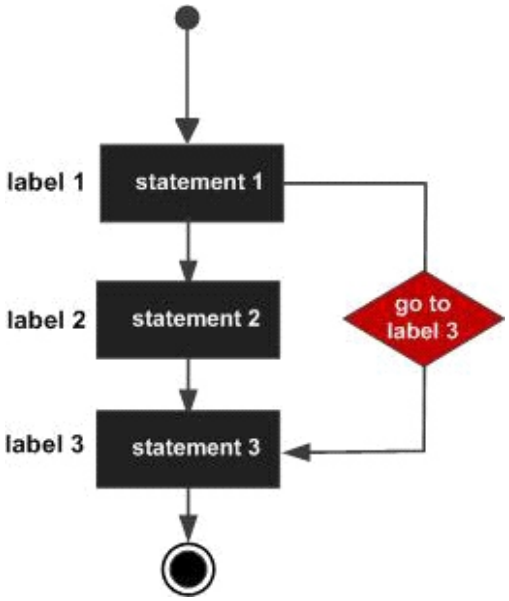
·When a break is encountered, it terminates execution and gets control out of the loop or switch, while when continue is encountered, the control is moved to the next iteration of the loop.

goto Statement

A goto statement in C programming provides an unconditional jump (or branch) to a labelled statement in the same block. A goto statement is not commonly used as it makes it difficult to trace the control flow of a program, making the program hard to understand and modify. Programmers often discourage their use in a program.

Construct:

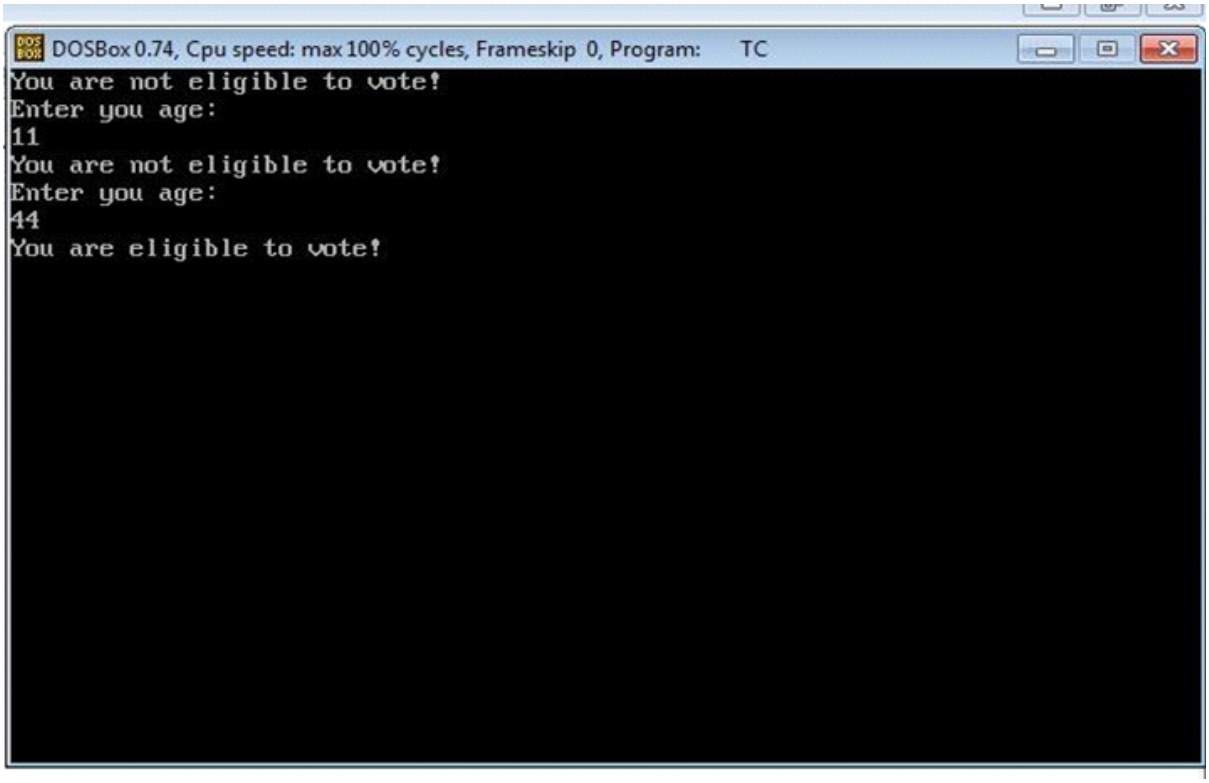
goto [label] ; It is executed as depicted in the flow chart below:



Example:

```
/* A program to check the eligibility of a voter */
#include <stdio.h>
int main() {
intage;
ineligible:
printf("You are not eligible to vote”);//a labeled statement
printf("Enter your age”);
scanf( “%d”, &age);
if(age < 18)
      goto ineligible;
else
printf(“you are eligible to vote”);
return 0;
}
```

Output:



Summary

In this unit, I have taught you the following:

- The termination control statements in C are used to alter or stop the execution of a loop or switch statement. They include break and continue statements
- The goto statement is used to perform an unconditional jump from one part of the program to another. Its use is often discouraged in programming.

Self-Assessment Questions

1. Give two differences between break and continue termination statements.
2. Write a C program to compute the sum of all odd numbers between 100 and 100001.

Tutor Marked Assessment

Rewrite the code below by using infinite while loop and break to achieve the same purpose:

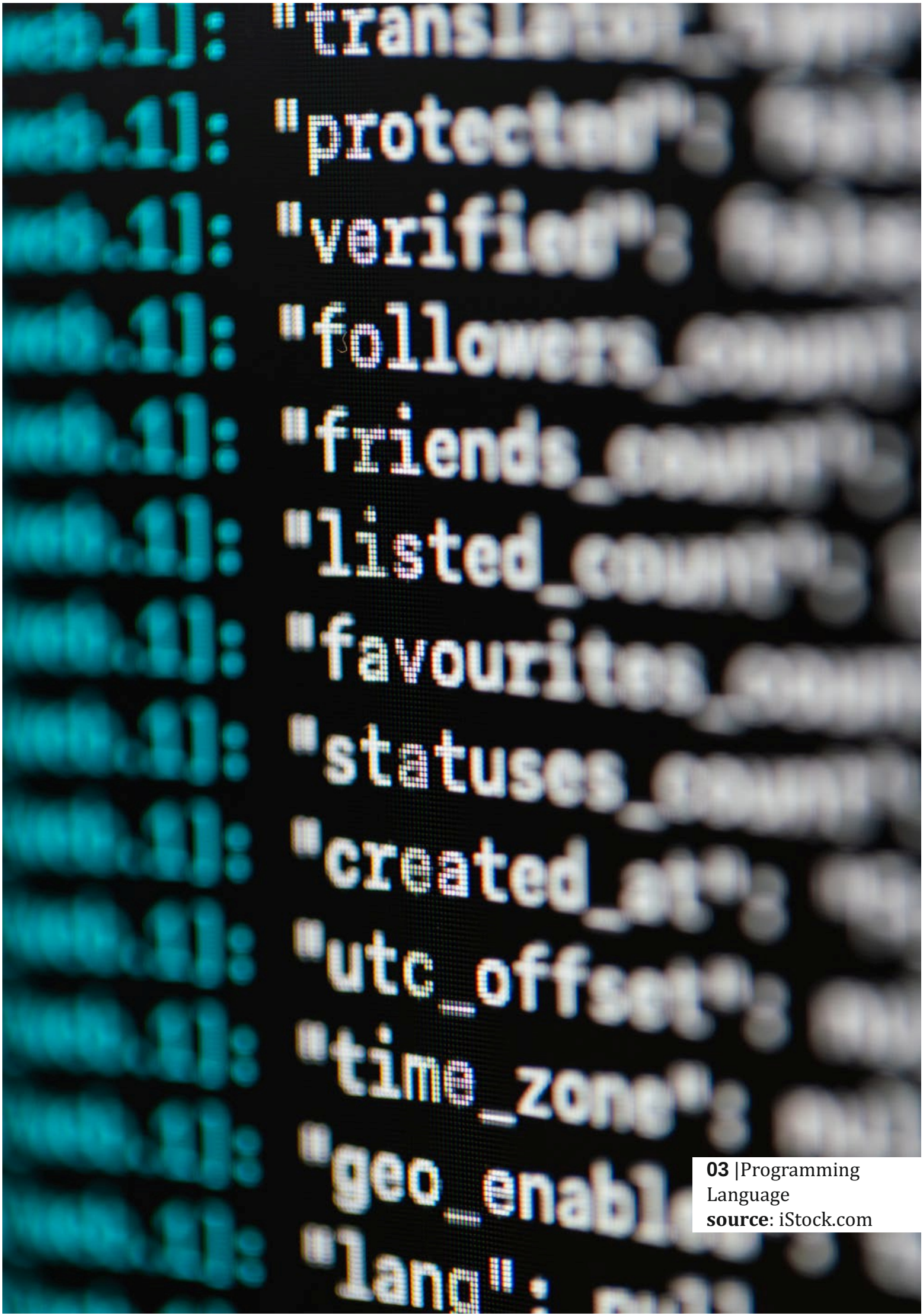
```
#include <stdio.h>
/* count the number of characters in input
string */
intmain()
{
longchN;
chN = 0;
while (getchar() != EOF)
++chN;
}
printf("%ld\n", nc);
return 0;
}
```

Further Reading

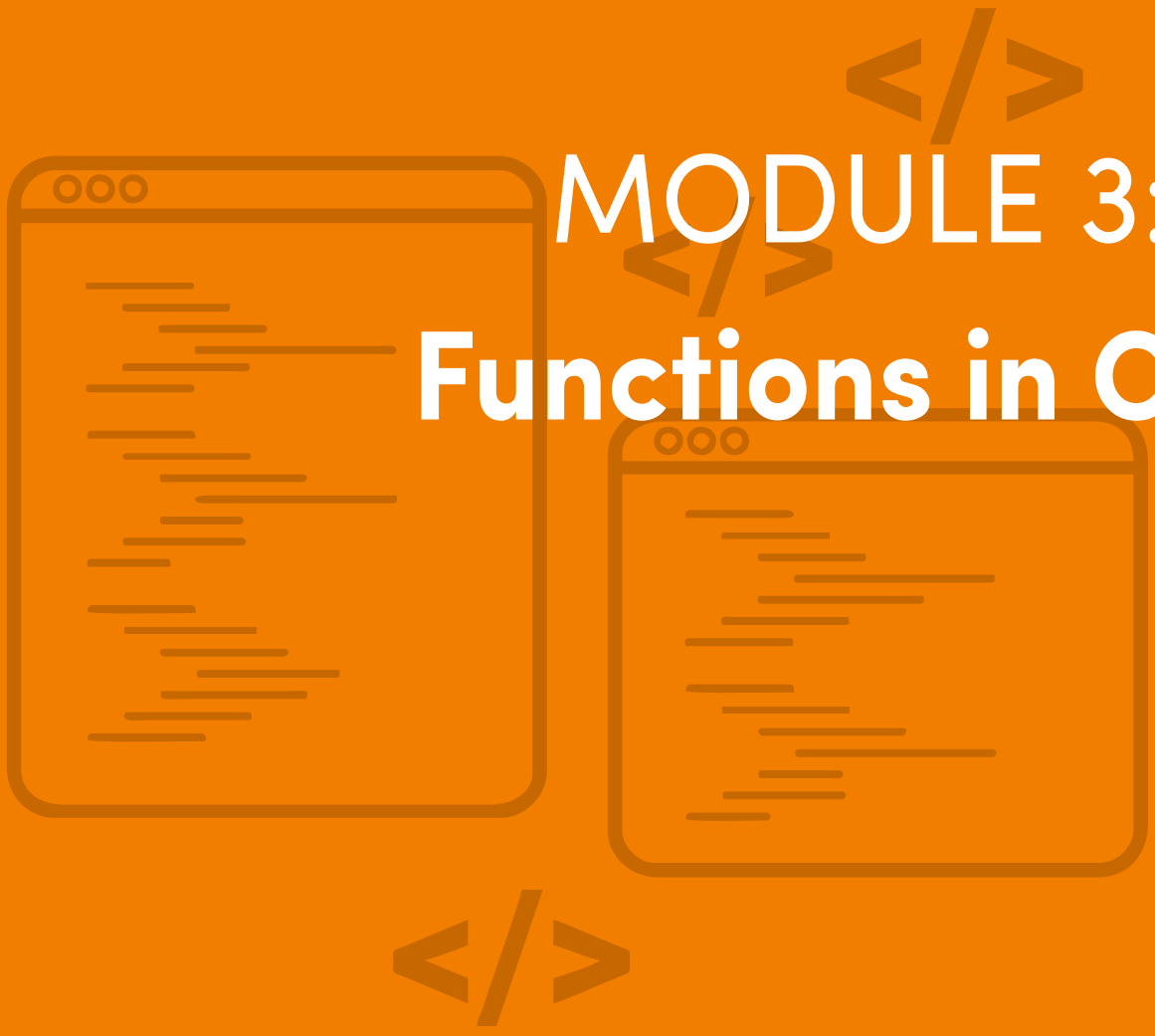
- http://www.tutorialspoint.com/cprogramming
- http://www.javatpoint.com/c-programming-language-tutorial

References

- Shola, P. B.(2002).
- Reflect Publishers, ISBN 978-047-174-X
- Learn C-Programming Language (2nd Ed).



03 |Programming Language
source: iStock.com



MODULE 3:
Functions in C

Unit 1:Function and its components

Unit 2:Function Call and Prototype

Unit 3:Multiplexing Techniques

```
nt factorial(int number);

nt main() {
int x = 6;
printf("The factorial of %d is %d\n", x, factorial(
return 0;}

nt factorial(int number) {
if (number == 1)     return (1); /* exiting condition
else  return (number * factorial(number - 1));}
```

A

Functions and components
source: guru99.com

UNIT 1

Functions and its Components

Introduction

In this unit, the central concept upon which the C programs are built; functions are explained to you. The unit introduces you to the concept of modular program design and focuses on how functions are used in C to build a modular program. It also discusses the components of a function and its type in details. This unit will lay a solid foundation for you in developing modular programs to solve complex problems.

Learning Outcomes

At the end of this unit, you should be able to:

1. Define modular design in procedural programming
2. Describe the syntax and component of a function definition
3. List the various types of functions
4. Write a simple user-defined function to perform a simple computation

Modular Program Design[SAQ1]

Procedural programming involves separating source code into self-contained components that can be accessed multiple times from different locations in a complete program. This approach enables separate coding of each component and assembly of various components into a complete program. We call this approach to programming solutions *modular design*.

The modular design identifies the components of a programming project that can be developed separately. Each module consists of a set of logical constructs that are related to one another. A module may refer to other modules.

```
/* programhello.c*/
 #include   <stdio.h>//   information   about   the   printf
identifier
int main(void)         // program startup
 {
printf("This is C"); // output
return 0;            // return to operating system
 }
```

The module named hello.c starts executing at statement intmain(void), outputs a string literal and returns control to the operating system as follows:

·**main()** transfers control to a module named printf()

·**printf()** executes the detailed instructions for outputting the string literal

·**printf()** returns control to main()

We can sub-divide a programming project in different ways. We select our modules so that each one focuses on a narrower aspect of the project. Our objective is to define a set of modules that simplifies the complexity of the original problem. For example, the one shown in figure 3.1.1.

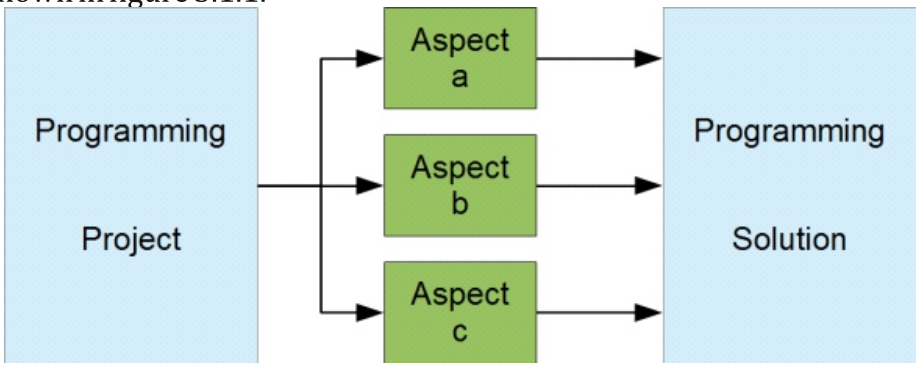


Figure 3.1.1: Modelling a Programming Solution with Modules

Some general guidelines for defining a module include:

·The module is easy to upgrade

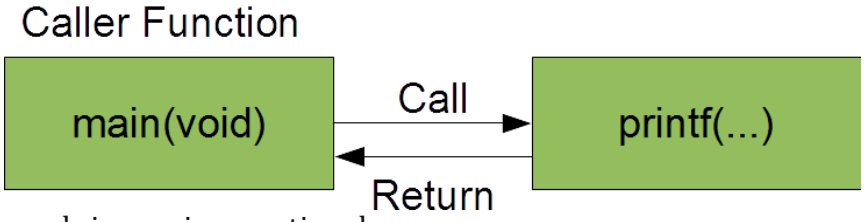
·The module contains a readable amount of code

·The module may be used as part of the solution to some other problem

Functions [SAQ2,3&4]

4 mins

The C language is a procedural programming language. It supports modular design through function syntax. A function is a group of statements that together perform a task. A function receives zero or more parameters, performs a specific task, and returns zero or one value. Functions transfer control between one another. When a function transfers control to another function, we say that it calls the other function. Once the other function completes its task and transfers control to the caller function, we say that that other function returns control to its caller.



In the example in previous sectionabove,

·the **main()** function calls the printf() function

·the **printf()** function outputs the string

·the **printf()** function returns control to its caller (main())

Characteristics of a Function

The following are some characteristics of a function in C.

·It is independent

·It is "completely" self-contained.

·It can be called at any place of your code and can be ported to another program.

·A function is invoked by its name and parameters.

·The function provides code reusability, code optimization, and readability.

Function Definition

A function consists of a header and a body. The body is the code block that contains the detailed instructions or actions to be performed by the function. The body is composed of statements enclosed in a curly bracket. The header immediately precedes the body and includes the following in order

·the type of the function's return value

·the function's name or identifier

·a parentheses-enclosed list of parameters that receive data from the caller

The syntax of a function definition in C is given below.

return_typefunction_name(type parameter1, ...,type parametern) Function Header

{

// statements

.

.

returnx; // x denotes the value returned by this function

}

Function Body

return_type specifies the type of the return value; ***type***specifies the function's parameter***function***

_name specifies the name of the function. A ***parameter*** is a variable that holds data received from the caller function. The parameter specified during the function definition is called **formal parameters.**

For example, here is the definition of a function that raises an integer number to a specified power of integer value:

```
/* Raise an integer to an integer power.c */
int power(int base, int exponent)
 {
```

```
inti, result;
result = 1;
for (i = 0; i< exponent; i++)
result = result * base;
return result;
 }
```

The function returns a value of int type. Power identifies the function. Base and exponent are the function's parameters; both are of int type. It is important to note that to return a value from a C function, you must explicitly return it with a return statement. The Syntax of the return statement is

return<expression>;

The expression can be any valid C expression that resolves to the return type defined in the function header.Type conversion (as discussed earlier) may occur if the type does not match. Also, multiple return statements can be used within a single function as the case may be (e.g., inside an "if-then-else" statement...). The return statement forms the exit point of the function.

Types of Function based on Parameters and Return

We have four different types of functions based on the way the parameters and return type are specified. These types are:

·Function with parameters and return

·Function with parametersbut no return

·Function with no parametersand return

·Function with no parameters and no return

Function with parameters and return

Did you know that the most common type of function that expects some values as parameters from its caller and as well returns a value are listed with types in the header. As a result to its caller. In the definition, the return type is specified, and the parameters are listed with the types in the header.

```
int sum ( int a, int b)
{
     return (a+b);
}
```

Example:

Function with parameters but no return

A function that does not have to return any value has no return type. We declare its return type as void and exclude any expression from the return statement. They are referred to as void functions. For example,

```
voidcountDown(int n)
 {
while (n > 0) {
printf("%d ", n);
n--;
          }
return;
 }

void sum ( int a, int b)

{

     printf("The sum of %d and %d is : %d", a, b, (a+b));

}
```

In this kind offunctions the return statement can be omitted. For example,

Function with no parameters but return

This kind of function returns a specific type but requires no data or parameters to perform its task. The parameter list is replaced with the keyword void, and a return statement is included.

```
int sum (void)

{

     inta,b;

     printf("pls enter two integers);

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

     return (a+b);

}
```

For example,

The void in the list of parameters can as well be omitted, but we must retain the parenthesis. For example, the function above can be written as:

```
int sum ()

{

     inta,b;

     printf(“pls enter two integers);

     scanf(“%d%d”, &a, &b);

     return (a+b);

}
```

Function with no parameters and no return

Functions that do not have to receive any data do not require parameters. Also, they do not return any value, thereby requiring no return type and statement. We optionally insert the keyword void between the parentheses. For example,

```
void alphabet(void)

 {

char letter = 'A';

do {

printf("%d ", letter);

letter++;

          } while (letter != 'Z')

 }
```

Note how the iteration changes the letter to the next character in the alphabet, assuming that the collating sequence arranged them contiguously.

Note that when the return type is not specified in the function header, and the keyword void

void is not used, C assumes the function to be of int type and must contain a return statement. For example, the sum function that returns an int can be written as:

```
sum ( int a, int b)

{

      return (a+b);

}
```

Types of Function based on Definition

There are two types of function based on their definition; built-in function and user-defined function.

Built-in Functions (Libraries)

These are functions that are predefined by the C language for use to aid programming tasks. They are built with the compiler and are written in the header files. To use them, appropriate header files should be included. Some header files and the built-in function then are given in the table below.

Header Files	**Functions Defined**
stdio.h	Printf(), scanf(), getchar(), putchar(), gets(), puts(), fopen(), fclose()
conio.h	Clrscr(), getch()
Ctype.h	Toupper(), tolower(), isalpha()
Math.h	Pow(), sqrt(), cos(), log()
Stdlib.h	Rand(), exit()
String.h	Strlen(), strcpy(), strupr()

User-Defined Functions

The user writes these kinds of functions at the time of programming.An example is the following function defined to be used by the programmer to perform a specific comparison.

```
int max(int a, int b){
      return  a>b ? a : b ;
}
```

The function returns the number higher in value than the other supplied.

Summary

In this unit, I have taught you the following:

- Inputs data and Output information are read and written in a sequence of characters using the stdin, stdout and stderr buffers or streams
- The character-based input/output functions are getchar(), getc(), putchar() and putc()
- Line of strings can be read and written using the string-based input/output functions gets() and puts()
- Formatted input and output tasks can be handled with the functions scanf() and printf()

Self-Assessment Questions

1. Give an accurate definition of modular design
2. Discuss the syntax and component of a function definition
3. List the types of functions based on parameters and definition
4. Write a simple function that returns the average of three values passed to it.

Tutor Marked Assessment

- Write a program that uses a function max() defined in this unit to find the largest in 6 sets of integer numbers and print it on the screen.

Further Reading

- http://www.javatpoint.com/c-programming-language-tutorial
- https://ict.senecacollege.ca/~ipc144/pages/content/modul.html

References

- Shola, P. B.(2002).
- Reflect Publishers, ISBN 978-047-174-X
- BalagurusamyE (2007).
- Programming in ANSI C, 4th Edition, McGraw-Hill.
- Learn C-Programming Language (2nd Ed).

```c
#include<stdio.h>

int add(int, int); // function prototype

int main()
{
    int a = 10, b = 20, c;

    c = add(a, b); // function call

    printf("Addition of %d and %d is %d\n", a, b, c);

    return 0;
}

//function definition
int add(int i, int j)
{
    return(i+j);
}
```

function call and prototypes
source: technotip.com

UNIT 2

Function Call and Prototypes

Introduction

I welcome you to the Function Call and Prototype unit. Here, I will discuss the use of Functions you learned in the previous unit to perform computations and tasks in your programs. It explains to you in details how functions are called to perform actions and return results to its caller. Communication between the caller function and the called function is also explained. Furthermore, the unit discusses function prototypes as a means of declaring a function and concludes with the placement of functions in small and large programs

Learning Outcomes

At the end of this unit, you should be able to:

1. Use appropriate function call to invoke already defined functions
2. Write a complete C program that uses functions to solve problems
3. Explain how arguments are passed to functions during a call

Function Call [SAQ1&2]

A function call transfers control from the caller to the function being called. Once the function being called has executed its instructions, it returns control to the caller. Execution continues at the point immediately following the call statement. A function is invoked (called) by its name and parameters. No two functions can have the same name and parameter types inthe C program.The communication between the function and the invoker is through the parameters and the return value. A function call takes the form:

function_name(argument,..., argument)

function_name specifies the function being called. ***argument*** specifies a value passed to the function being called. Take for example,

A function header defined as: `int func1 (int x, int y, int z)`

The Function call will look like this: `func1 (a, b, c);`

From the call, parameters x, y, and z in the function definition are initialized by the value of a, b and c, respectively.

An argument (often called actual parameter) may be a constant, a variable, or an expression (that returns a value). The number and types of arguments in a function call should match the number and types of parameters in the function header.

Example: let us write a program that uses the function max() defined in the previous unit.

```
#include <stdio.h>

int max(int a, int b){

      return a>b?a:b;

}

int main(){

      int x;

         x = max(5,8);

         x = max(x,7);
```

```
printf("The largest of the numbers is: %d", x);

return 0;

}
```

The program will print to the screen:

```
The largest of the numbers is: 8
```

```
#include <stdio.h>

int sum ()

{

      inta,b;

      printf("pls enter two integers);

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

      return (a+b);

}

int main(){

      sum();

      return 0;

      }
```

Another example is given below:

The program will prompt the user to input two numbers and print their sum to the screen.

Note the slight difference in the function call of the two programs. The first assigns the call to a variable because the function returns a value, then calls the function without assignment since it does not return a value.

Below is another example that uses the expression asan argument to a function.

```
int main(){
      inti =10, j = 5, sum;
      sum = add(add(i,j), add(i-j, i+j));
      printf ("the sum is %d", sum);
      return 0;
      }
#include <stdio.h>
int add (int a, int b)
{
      return (a+b);
}
int main(){
      inti =10, j = 5, sum;
      sum = add(i-j, i+j);
      printf ("the sum is %d", sum);
      return 0;
      }
```

It is also possible to pass a function as argument to another function. For example, if we want the previously defined add() function to be used to find the sum of four numbers just in one statement, we can rewrite the main function above as:

The program will add up 10, 5, 10—5, and 10+5 assigns the result to sum in a statement.

If the type of an argument does not match the type of the corresponding parameter, the compiler coerces (narrows or promotes) the value of the argument into a value of the type of the corresponding parameter. Consider the power() function listed in unit 1 above and the following call to it.

```
int answer;
    answer = power(2.5, 4);
```

The compiler converts the first argument (2.5) into a value of type int, as shown below:

```
int answer;
    answer = power((int)2.5, 4);
```

The C compiler evaluates the cast (coercion) of 2.5 before passing the value of type int to power and initializing the parameter to the cast result (2).

Generally, C required that all functions must be defined before they are called, as shown in the program examples above; however, this contradicts the structure of C programs that places subprogram like user defines functions after the main function. To allow us to use a function (calling it) before it is defined, C provides function prototypes as a form of function declaration.

Function Prototype [SAQ3]

A function prototype is used to declare a function which is to be defined later on in the program. The function prototype is composed of the header followed by a semicolon (;). It has the following syntax,

return_typefunction_name (type1 name1, type2 name2, ..., typennamen);

return_type specifies the type of the return value, function_name specifies the name of the function, type1…typenspecifies the type of the function's parameters, and OPTIONAL name1 …namenis an identifier holding a place for the formal parameters to be defined in the function header at definition time.

Function prototypes are placed in the global declaration section of the C program to allow us call the function in the main section before it is defined in the subprogram section of the program structure.

For example, the two programs written above in section 3.1 can be rewritten using prototypes as follow:

```
#include <stdio.h>
int sum ();
int main(){
     sum();
     return 0;
```

```
            }
        int sum ()
        {
            inta,b;
        printf("pls enter two integers);
        scanf("%d%d", &a, &b);
        return (a+b);
        }
        #include <stdio.h>
        int max(int a, int b);
        int main(){
            int x;
               x = max(5,8);
               x = max(x,7);
            printf("The largest of the numbers is: %d",
    x);
            return 0;
            }
        int max(int a, int b){
            return a>b?a:b;
        }



/* A program that finds a specified exponent of numbers
supplied by the user */
#include <stdio.h>
int a, b;
int power(int, int);
int main ()
{
    long out;
    printf("pls enter the base integer);
    scanf("%d", &a);
```

```
     printf("pls enter the exponent value);
     scanf("%d", &b);
     out = power(a,b)
     printf ("%d raised to power of %d is %ld", a,b,out);
     return 0;
}
int power(int base, int exponent)
 {
inti, result;
result = 1;
for (i = 0; i< exponent; i++)
result = result * base;
return result;
 }
```

Since the parameter name in the function prototype is optional, it can be omitted. For example, a program that uses the power function defined in section 3.2.2 of unit 1 is given below:

Run the program on your code blocks and supply two integer numbers, you will see that the program runs perfectly with the specified function prototype containing only the types in the parenthesis.

Passing Arguments to a Function

The C language passes data from a caller to a function by value. That is, it passes a copy of the value and not the value itself. The value passed is stored as the initial value in the parameters that correspond to the argument in the function call.

```
#include<stdio.h>

inttwice(int x);

int main()

{

      int x=10,y;

      y=twice(x);

      printf("%d,%d\n",x,y);

}

int twice(int x)

{

      x=x+x;

      return x;

}
```

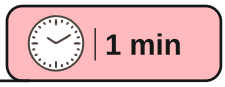
Each parameter is a variable with its memory location. We refer to the mechanism of allocating separate memory locations for parameters and using the arguments in the function call to initialize these parameters pass by value. Pass by value facilitates modular design by localizing consequences. The function being called may change the value of any of its parameters many times, but the values of the corresponding arguments in the caller remain unchanged. In other words, a function cannot change the value of an argument in the call to the function. This language feature ensures the variables in the caller are relatively secure. For example, consider the program below:

Output of the program is: 10, 20

The function twice() modifies the value of the variable x passed to it from the main function to twice its original value, but the caller does not see this modification. That is why the value of the variable x in the main, remains unchanged. This is the implication of pass by value. Furthermore, study the following program and try to predict its exact output before running it on your code blocks

```
#include <stdio.h>

int x=1;   /* global variable - bad! */

void demo(void);

int main() {

int y=2;  /* local variable to main */

printf ("\nBefore calling demo(), x = %d and y = %d.",x,y);

demo();

printf ("\nAfter calling demo(), x = %d and y = %d.\n",x,y);

return 0;

}

void demo () {

int x = 88, y =99;  /* local variables to demo */

printf ("\nWithin demo(), x = %d and y = %d.",x,y);

}
```

Placement of Functions



Functions are placed in different locations in our program based on the size of the program we are building.

For large programs

·Manage related functions in a .c file

·Write a .h file containing all the prototypes of the functions

·#include the header file in the files that uses the functions.

For example:

mymath.h:

```
int min(intx,int y);
int max(intx,int y);
```

mymath.c:

```
 int min(intx,int y)
{
 return x>y?y:x;
}
int max(intx,int y)
{
 return x>y?x:y;
}
```

Program that uses mymath functions:

```
#include <mymath.h>
      …..
```

For small programs

Use the following order in the only one file:

·All prototypes

·main() function

·Other functions

Summary

In this unit, I have taught you the following:

- A function is called by specifying the function name followed by a list of arguments enclosed in parenthesis.
- The function call causes control to be shifted to where the function being called is defined
- An argument can be a constant value, variable, expression, and even another function call.
- Function prototypes are used to declare a function before it is used.
- Functions are placed in programs based on the size and complexity of the program being built

Self-Assessment Questions

1. Describe how to call the functions countdown() and alphabets() defined in unit 1 of this module
2. Write a C program that uses the function power() defined in unit 1 of this module to evaluate the value of the polynomial 4x3 – 5x2 +10 for a value of x to be supplied by the user.
3. What do you understand by the term pass by value?

Tutor Marked Assessment

- Using code examples, differentiate between Formal parameter and Actual parameter.

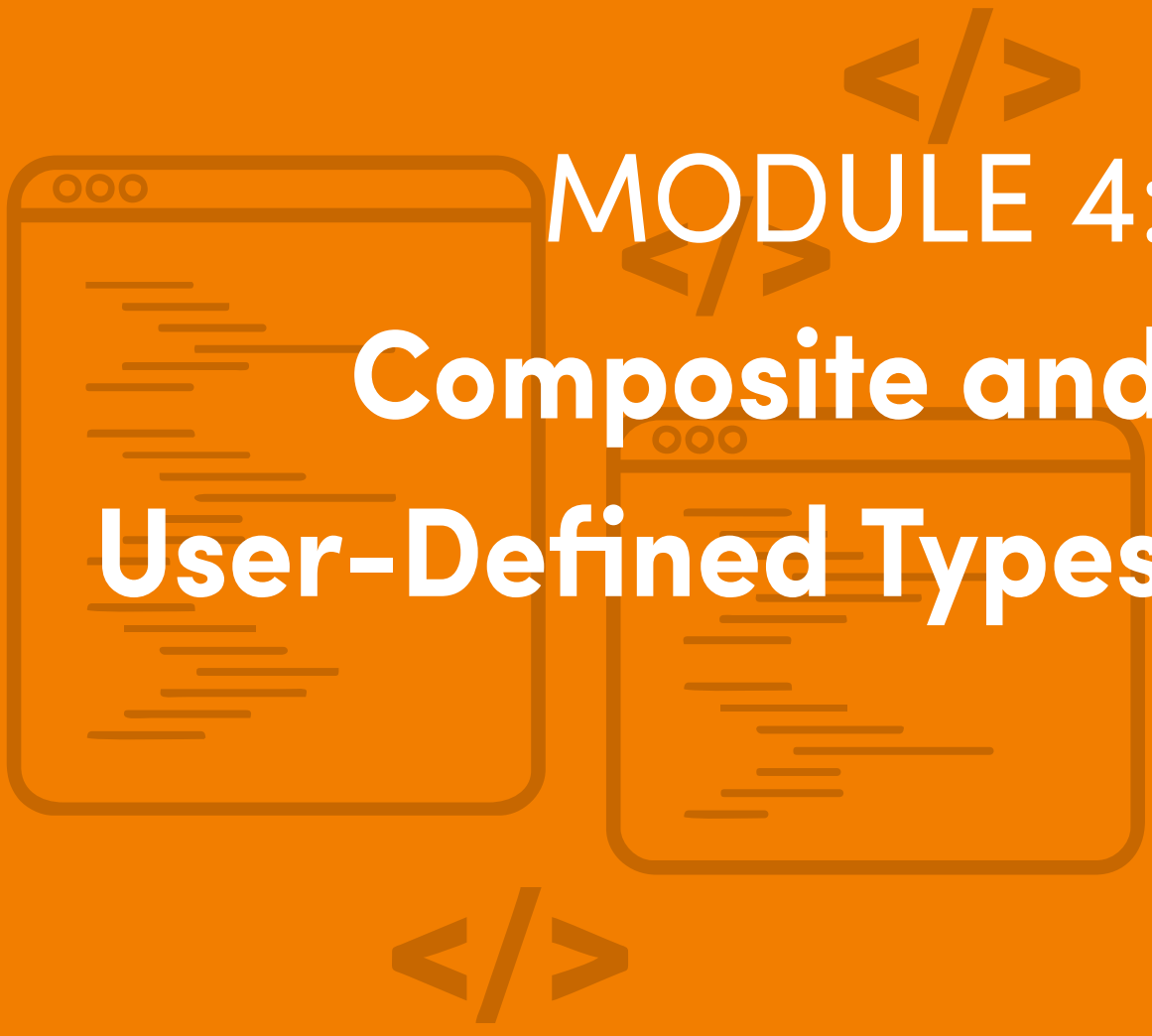
Further Reading

- http://www.javatpoint.com/c-programming-language-tutorial
- https://ict.senecacollege.ca/~ipc144/pages/content/modul.html

References

- Shola, P. B.(2002).
- Learn C-Programming Language (2nd Ed).
- Reflect Publishers, ISBN 978-047-174-X.
- Sabyasachi M. (2010).
- C programming.Techno India University.India.

04 |Codes running on a desktop
source: iStock.com



MODULE 4: Composite and User-Defined Types

Unit 1:Array and Enumeration Types

Unit 2:Pointers and Strings in C

Unit 3:Structures and Union

```
f pointers
[13] = { "", "January", "February", "March",
         "April", "May", "June", "July", "August",
         "September", "October", "November",
         "December" };

nts represent months of the year
, FEB, MAR, APR, MAY, JUN,
, SEP, OCT, NOV, DEC} months;

contain any of the 12 months
```

Array and enumeration
source: google.com

UNIT 1

Array and Enumeration

Introduction

In this unit, I will introduce you to the concept of composite datatypes in C. it discusses array and enumeration. Implementation details of each of the types are explained with practical examples.

Learning Outcomes

At the end of this unit, you should be able to:

1. Create an array in C
2. Perform simple computations using an array
3. Define and use a multidimensional array
4. Describe how arrays are passed to function

Composite Types

1 min

Composite data types are types that are composed of other fundamental or basic data types.

There are two categories of composite datatypes in C:

Derived Types: They are directly created from one or more fundamental types. Examples are Array, Pointer, Structure, and Union

User-defined Types: They are special types created by the user. Examples are Enum and Typedef

Array [SAQ1]

2 mins

An array is a group of similar objects that can be defined as a contiguous area in memory and can be referred to by a common name. It is a set of values of the same datatype. Each component of the array is referred to as an element. Array has a unique identifier for each element, called a subscriptor an index, which starts from zero (0).The syntax for declaring an array in C is:

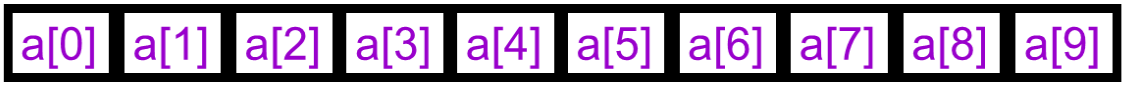
typearrayname[size];

typename is the type of the element of the array, which can be any datatype, arrayname is any valididentifier that represents the name of the array and size is the number of elements the array will contain.

For example:

```
int  a[10];
```

defines an array of ints with subscripts ranging from 0 to 9. There are 10*sizeof(int) bytes of memory reserved for this array.



Also:

```
char string [11];
```

defines a character array named string to store 10characters.string[0] to string[9] are for valid characters. String [10] for the string-terminator character, \0 (NULL). This is peculiar to strings of character. To access any element of the array, the array name followed by the subscript or index number of the element is used put in square bracket. For example, to access the fifth element of the array string declared above, we use:

```
string[4] //since the index starts from 0
```

So the following statement assigns 'c' to the fifth element:

Initializing an Array

Initialization of arrays can be done by a comma-separated list in a curly bracket following its definition.

For example:

```
int array [4] = { 100, 200, 300, 400 };
```

This is equivalent to:

```
int array [4];

      array[0] = 100;

      array[1] = 200;

      array[2] = 300;

      array[3] = 400;
```

You can also let the compiler figure out the array size for you:

```
int array[] = {100, 200, 300, 400};

#include <stdio.h>

int main() {

float expenses[12]={10.3, 9, 7.5, 4.3, 10.5, 7.5, 7.5, 8,
9.9, 10.2, 11.5, 7.8};

intcount,month;

float total;

for (month=0, total=0.0; month < 12; month++)

  {

          total+=expenses[month];

  }
```

A simple example of using array initializer and using loop to access the elements of an array in our program is given below:

```
for (count=0; count < 12; count++)

printf ("Month %d = %.2f K$\n", count+1, expenses[count]);

printf("Total = %.2f K$,   Average = %.2f K$\n", total,
total/12);

return 0;

}
```

Run the code on your code blocks and study its output.

It is important to note that the array name, when used alone, refers to the address of the first element of the array. So when it is printed, it displays an int value of the address of the array (the address of the first element, not its value). For example,

```
#include <stdio.h>

int main(){

inti;

int a[3] = { 1, 2, 3 };

printf("a ? %d\n", a);

printf("a[0]? %d\na[1]? %d\na[2]? %d\n", a[0], a[1],a
[2]);

return 0;

}
```

The first printf() function in the program prints the address of the array a.

C compilers do not introduce code that checks whether an element's index is within the bounds of its array. It is our responsibility as programmers to ensure that our code does not include index values that point to elements outside the memory allocated for an array.

Multi-Dimensional Array [SAQ3&4]

Arrays in C can have virtually as many dimensions as you want.The definition is accomplished by adding additional subscripts when it is defined.To identify an element of a two-dimensional array, we use two pairs of brackets. The index within the left pair identifies the row, while the index within the right pair identifies the column:

array[row][column]

Indexing is 0-based for both rows and columns.

Definition

The definition of a two-dimensional array takes the form

type identifier[r][c] = init;

Where r is the number of rows in the array, and c is the number of columns. r and c are integer constants or constant integer expressions. The total number of elements in the array is r * c. init is a braces-enclosed, comma-separated list of initial values. The assignment operator,together with init, is optional. If we add an initialization list, we may omit the value of r. If r * c exceeds the number of initial values, the compiler initializes the remaining elements to 0. If we omit the initialization list, we must specify r. We must always specify c.

For example:

```
int  a [4] [3] ;
```

defines a two-dimensional array where a is an array of int[3]. Basically a 2 dimensional array is an array of arrays. A 2-dimensional array can be initialized as follows:

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

Also can be done by:

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

It is equivalent to:

```
a[0][0] = 1;

a[0][1] = 2;

a[0][2] = 3;

a[1][0] = 4;

...

a[3][2] = 12;
```

The C language stores the elements of a two-dimensional array in row-major order: the first row, column-element by column-element, then the second row, column-element by column-element, then the third row, etc..

For example, the elements of the array

```
int a[4][5];
```

are stored as follows:

```
a[0][0] a[0][1] a[0][2] a[0][3] a[0][4]
 a[1][0] a[1][1] a[1][2] a[1][3] a[1][4]
 a[2][0] a[2][1] a[2][2] a[2][3] a[2][4]
 a[3][0] a[3][1] a[3][2] a[3][3] a[3][4]
```

Some programming languages store two-dimensional arrays in column-major order.

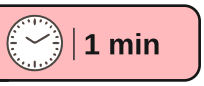
```
#include <stdio.h>

#include <stdlib.h>

int main ()  {

     int random1[8][8];

     int a, b;

     for (a = 0; a < 8; a++)

          for (b = 0; b < 8; b++)

               random1[a][b] = rand()%2;
```

A two-dimensional array is very useful in computation that involves matrices. A 2-dimensional array can be accessed using nested loop statements. For example, the following program:

```
     for (a = 0; a < 8; a++)

     {

          for (b = 0; b < 8; b++)

          printf ("%c " , random1[a][b] ? 'x' : 'o');

          printf("\n");

     }

return 0;

}
```

The program creates a 2-dimensional array of 8 by 8 and randomly fills the elements with 1 or 0 based on the remainder of a random number divided by 2. It then prints the result out using character 'x' for 1 and 'o' for 0. The program uses nested for loop twice to achieve this task. You should run the program on your code clocks to familiarize yourself with this technique.

Array as Function Parameter



Remember that to pass a value to a function, we include the name of the variable as an argument of the function.

Similarly, an array can be passed to function by including the array name and the size of the array as arguments. Usually, in the function definition, the array parameter is specified by including the array name with an empty square bracket.

For example a function prototype:

```
float average (int n, float x[]);
```

can be called with:

```
avg = average (n, x);
```

Following is an example of a program that passes the array to a function.

```
#include <stdio.h>
voidinc_array(int a[ ],int size);
main()
{
      int test[3]={1,2,3};
      intary[4]={1,2,3,4};

      inti;
      inc_array(test,3);
      for(i=0;i<3;i++)
            printf("%d\n",test[i]);
      inc_array(ary,4);
      for(i=0;i<4;i++)
            printf("%d\n",ary[i]);
      return 0;
}

voidinc_array(int a[ ], int size)
{
      inti;
      for(i=0;i<size;i++)
      {
            a[i]++;
      }
}
```

Note that since the arrays in the program are passed using the name of the arrays, which is an address, then any modification to the array in the called function affects the array. This is called **pass by reference**. The printf() functions in the program will print the modified values by the function inc_array.

A practical example is a program that sorts the elements of an array and prints the sorted list.

```
#include <stdio.h>
voidmysort(int a[ ],int size)
{
inti,j,x;
for(i=0; i<size; i++)
  {
for(j=i; j>0; j--)
    {

if(a[ j ] < a[ j-1])
{  /* Change the order of a[ j ] and a[ j-1] */
          x=a[j];
          a[j]=a[j-1];
          a[j-1]=x;
      }
    }
  }
}
int main()
{
     inti;
     int tab[10] = {3,6,3,5,9,2,4,5,6,0};
     for(i=0;i<10;i++)
          printf("%d ",tab[i]);
     printf("\n");
     mysort(tab,10);
     for(i=0;i<10;i++)
          printf("%d ",tab[i]);
     printf("\n");
     return 0;
}
```

Let us Run the program and study its output.

Enumeration Type

Enumeration is a user-defined datatype in C language. It is used to assign names to the integral constants, which makes a program easy to read and maintain. The keyword "enum" is used to declare an enumeration. Here is the syntax of enum in C language,

enumenum_name{const1, const2, ... };

enum is the keyword that tells the compiler we are creating an enumeration type, enum_name is any valid identifier representing the name of the new user-defined enum we are creating, and const1.....constn are constant names that enumerate the values our enum type can assume. For example the following declaration.

```
enum week{sunday, monday, tuesday, wednesday, thursday,
friday, saturday};
```

Defines enum type week to contain constant names sunday, monday,.....saturday.

To use the enum type creates, we have to create a variable of that type. The enum keyword is also used to define the variables of an enum type. The syntax is as follows

enumenum_namevariable_name;

For example, we can create a variable of the enum type we created above using the declaration:

```
enum weekday;
```

The variable can as well be created at the point of declaration using the syntax:

enumenum_name{const1, const2, ... }Variable_name;

For example,

```
enum week{sunday, monday, tuesday, wednesday, thursday,
friday, saturday} day;
```

declares an enum type week and creates a variable day for it.

Note that the user definesenum can only be identified by the combination of the keyword enum and the enum_name. That is why it is compulsory to use the keyword enum with the enum_name during enum variable creation, as shown above.

Now the variable created can assume values within the constants provided when defining the enumeration type. for example,

```
day = wednessday;
```

assignswednessday, one of the enumerators (enum constants) to variable day.

It is also important to note that by default, the values of the enumerattin constants internally are consecutive integer values starting from zero i.e. constant1 = 0, constant2 = 1, constant3 = 2 and so on.

```
// An example program to demonstrate working
// of enum in C
#include<stdio.h>
  enum week{Mon, Tue, Wed, Thur, Fri, Sat, Sun};
  int main()
{
    enum week day;
    day = Wed;
    printf("%d",day);
    return 0;
}
```

Example:

```
Output:
2
```

Another example is given below:

```
// Another example program to demonstrate working
// of enum in C
#include<stdio.h>

enum year{Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec};

int main()
{
   inti;
   for (i=Jan; i<=Dec; i++)
      printf("%d ", i);

   return 0;
}


Output:
0 1 2 3 4 5 6 7 8 9 10 11
```

In this example, the for loop will run from i = 0 to i = 11, as initially, the value of i is Jan, which is 0, and the value of Dec is 11.

Also, we can assign values to some names in any order. All unassigned names get values equal to the value of the previous name plus one.

```
#include <stdio.h>
enum day {sunday = 1, monday, tuesday = 5,
          wednesday, thursday = 10, friday, saturday};
int main() {
    printf("%d %d %d %d %d %d %d", sunday, monday, tuesday,
            wednesday, thursday, friday, saturday);
    return 0;
}

Output:
1 2 5 6 10 11 12
```

Its also possible that two enum names are assignedthe same values. For example, in the following C program, both 'Failed' and 'Freezed' have the same value 0.

```
#include <stdio.h>
enum State {Working = 1, Failed = 0, Freezed = 0};

int main()
{
   printf("%d, %d, %d", Working, Failed, Freezed);
   return 0;
}


Output:
1, 0, 0
```

Lastly, all enum constants must be unique in their scope. For example, the following program fails in a compilation.

```
enum state  {working, failed};
enum result {failed, passed};

int main()  { return 0; }
```

```
Output:
Compile Error: 'failed' has a previous declaration as 'state
failed.'
```

Operations on enum Variable

Since enum constant values are integers internally, arithmetic operations can be performed on them as long as the operation returns integer values. For example:

```
#include <stdio.h>

enum week {Sunday, Monday, Tuesday, Wednesday, Thursday,
Friday, Saturday};
int main()
{
    // creating today variable of enum week type
enum week today;
today = Wednesday;
printf("Day %d",today+1);
return 0;
}
```

Enum variables can as well be used in arithmetic expressions. For example

```
#include <stdio.h>
enum week {Sunday, Monday, Tuesday, Wednesday, Thursday,
Friday, Saturday};

int main()
{
    // creating today variable of enum week type
enum week today;
    int yesterday;
today = Wednesday;
    yesterday = today -1;
printf("Day %d",yesterday);
return 0;
}
```

Summary

In this unit, I have taught you the following:

- An array is used to store a set of elements of the same type referenced with a name
- The elements of the array can be accessed using the array name and the index of the element to be accessed
- Array elements can be accessed automatically using loop statements
- Enumeration types are user-defined types that are used to create constant names as a list of values internally represented as integers.

Self-Assessment Questions

SAQ

1. Describe how to create an array in C
2. Create a program in C that finds the average of a set of 10 numbers specified by the user
3. What is a multidimensional array
4. Explain how arrays are passed to functions

Tutor Marked Assessment

- Discuss using suitable examples of how a multidimensional array can be accessed

Further Reading

- https://www.programiz.com/c-programming/c-enumeration
- http://www.javatpoint.com/c-programming-language-tutorial
- https://www.geeksforgeeks.org/enumeration-enum-c/

References

- BalagurusamyE (2007).
- Programming in ANSI C, 4th Edition, McGraw-Hill
- Sabyasachi M. (2010).
- C programming.Techno India University.India.
- Shola, P. B.(2002).
- Learn C-Programming Language (2nd Ed).
- Reflect Publishers, ISBN 978-047-174-X

Illustration of Strings
source: iStock.com

UNIT 2

Pointers and Strings

Introduction

In this lesson, I will discuss with you the concept of a pointer in C for memory management. It focuses on pointer declaration, use, operations, and dynamic memory allocation. It also discusses how strings are handled in C, focusing on some functions used for string processing.

Learning Outcomes

At the end of this unit, you should be able to:

1. Use the pointer to manipulate addresses of variables
2. Allocate memory dynamically to pointers at run time
3. Define and use strings in your program effectively
4. Write a simple program to search through a text

Pointer [SAQ1]

When we define a variable, the compiler (linker/loader actually) allocates a real memory address for the variable.

`int x;` will allocate 4 bytes in the main memory, which will be used to store an integer value.

When a value is assigned to a variable, the value is actually placed on the memory that was allocated.

`x=3;` will store integer 3 in the 4 bytes of memory.

When the value of a variable is used, the contents in the memory are used.

`y=x;` will read the contents in the 4 bytes of memory, and then assign it to variable y.

`&x` can get the address of x. (referencing operator &). The address can be passed to a function:

```
scanf("%d", &x);
```

The address can also be stored in a variable.A variable that holds an address is called a pointer. By definition, a pointer is a variable whose value is the address of another variable, i.e. Thedirect address of the memory location. Like any other variable, you must declare ***a pointer*** before you can use it to store any variable address. To store a variable's address, we define a pointer of the variable's type and assign the variable's address to that pointer.

The general form of a pointer variable declaration is:

Datatype *var_name;

datatype * is the type of the pointer. ***var_name*** is the name of the pointer.

For example :

```
int *ip;
double *dp;
float *fp;
char *cp;
```

The actual data type of the value of all pointers is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different datatypes is the data type of the variable that the pointer points to.

Pointer Types

The C language allows us to use pointer type for every primitive or derived type:

Type	Pointer Type
char	**char ***
short	**short ***
int	**int ***
long	**long ***
long long	**long long ***
float	**float ***
double	**double ***
long double	**long double ***
Product	**Product ***

C compilers typically store addresses in 4 bytes of memory.

Initializing Pointer

Like every other variable in C, we must initialize pointer before using them. For example:

```
int main(){

int x;

int *p;

p = &x; // initializes p with the address of x

scanf("%d", p);
```

The code snippet above will read an integer value to the address pointed to by p. The Code will generate garbage value if the value is read through p without initializing it.

Using a Pointer Variable

There are a few important operations that are required to use pointers.

·We declare a pointer variable

·Assign the address of a variable to the pointer

·Finally access the value at the address available in the pointer variable by deferencing

Deferencing a pointer means getting the value store in the memory location at the address which the pointer points to. The unary operator * which stands for 'data at address' or simply 'data at' called the dereferencing or indirection operator is used. This operator applied to a pointer's identifier evaluates to the value in the address that that pointer holds. The operator is also used when declaring a pointer.

```
/* Working with Pointers  */
#include <stdio.h>
int main(void)
{
int x;
int *p = &x; // store address of x in p
printf("Enter x : ");
scanf("%d", &x);
printf("Value stored in x : %d\n", *p);
printf("Address of x      : %x\n", p);
return 0;
}
```

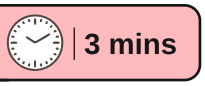
Example: The following program stores the address of variable x in pointer p and displays the value in that address using the pointerAlso take a look at these statements on accessing value pointed to by a pointer:

```
int n, m=3, *p;
p=&m;
n=*p; // the * operator returns the value of variable in the
      //address stored in P
printf("%d\n", n);
printf("%d\n",*p);
```

You can as well assign value to the variable pointed to by a pointer using the indirection operator and the pointer name at the left-hand side of an assignment statement. For example,

```
#include <stdio.h>
int main(void)
 {
     int m=3, n=100, *p;
     p=&m;
     printf("m is %d\n",*p);
     m++;
     printf("now m is %d\n",*p);
     p=&n;
     printf("n is %d\n",*p);
     *p=500;    /* *p is at the left of "="  */
     printf("now n is %d\n", n);
     return 0;
}
```

Pointer Arithmetic



C pointer is an address, which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can on numeric values. Four arithmetic operators can be used on pointers: ++, --, + and –

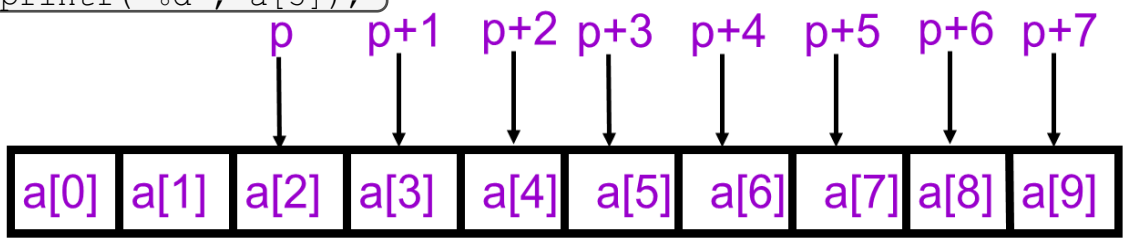
When a pointer variable points to an array element, there is a notion of adding or subtracting an integer to/from the pointer.

For example, this code:

```
int a[ 10 ], *p;
p = &a[2];
*p = 10;
*(p+1) = 10;
printf("%d", *(p+3));
```

is equivalent to:

```
int a[ 10 ], *p;
a[2] = 10;
a[3] = 10;
printf("%d", a[5]);
```



More examples:

```
int a[10], *p, *q;
p = &a[2];
q = p + 3;              /* q points to a[5] now */
p = q - 1;              /*  p points to a[4] now */
p++;                  /*  p points to a[5] now */
p--;                        /* p points to a[4] now */
*p = 123;               /*  a[4] = 123 */
*q = *p;              /*   a[5] = a[4]   */
q = p;                /*   q points to a[4] now */
scanf("%d", q)     /* scanf("%d", &a[4])  */
```

The following program increments the pointer variable to access each succeeding element of the array:

```
#include <stdio.h>
constint MAX = 3;
int main()
 {
     intvar[] = {10, 100, 200};
```

```
      inti, *ptr
      //let us have array address in our pointer
      ptr = var;
      for (i = 0; i< MAX; i++)
        {
           printf(" Address of var[%d] is %x\n", i, ptr);
           printf(" the value of var[%d] is %d\n", i, *ptr);
           // move to the next location
           ptr++;
        }
      return 0;
}
```

Pointer Comparism

1 min

If two pointers point to variables that are related to each other such as elements of the same array, then both can be meaningfully compared. Pointers may be compared by using the relational operators such as ==, < and >. Example:

```
int a[10], *p, *q , i;

p = &a[2];

q = &a[5];

i = q - p;          /* i is 3*/

i = p - q;          /* i is -3 */

a[2] = a[5] = 0;

i = *p - *q;       /* i = a[2] - a[5]  */

p < q;              /* true */

p == q;            /* false */

p != q;             /* true */
```

The following modifies the previous program in section 3.2 by using pointer comparism:

```
#include <stdio.h>

constint MAX = 3;

int main()
```

```
 {

intvar[] = {10, 100, 200};

inti, *ptr

//let us have array address in our pointer

ptr = var;

i = 0;
while(ptr<&var[MAX-1])

       {

          printf(" Address of var[%d] is %x\n", i, ptr);

          printf(" the value of var[%d] is %d\n", i, *ptr);

          // move to the next location

          ptr++;

          i++;

       }

return 0;

}
```

Run the programs on your code block and check if there is any difference in the results.

Array of Pointers

1 min

There may be a situation when we want to maintain an array that can store pointers to an int, char, or any other types available in C. Following is the declaration syntax of an array of pointers.

For example: `int *ptr[];`

Declares ptr as an array of integer pointers. Thus each element in ptr now holds a pointer to an int value. The following example makes use of 3 integers which are stored in an array of pointers:

```
#include <stdio.h>

constint MAX = 3;

int main()
```

```
 {

intvar[] = {10, 100, 200};

inti, *ptr[MAX];

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

        {

           ptr[i] = &var[i]; // assign address of integers

        }

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

        {

           printf(" the value of var[%d] is %d\n", i, *ptr[i]);

        }

return 0;

}
```

Pointers and Arrays

2 mins

Recall that the value of an array name is also an address.In fact, pointers and array names can be used interchangeably in many (but not all) cases.E.g.

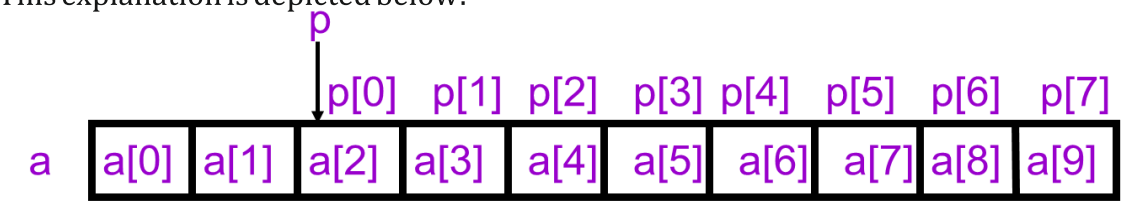
```
int n, *p;
p=&n;
  n=1;
*p = 1;
p[0] = 1;
```

The major differences are:

Array names come with valid spaces where they "point" to. And you cannot "point" the names to other places. Pointers do not point to valid space when they are created. You have to point them to some valid space (initialization). Therefore, C allows pointers to be used to access the array elements in place of the actual array name. For example:

```
int a[ 10 ], *p;
p = &a[2];
p[0] = 10;
p[1] = 10;
printf("%d", p[3]);
```

This explanation is depicted below:



Passing Pointer to a Function

C allows us to pass a pointer to a function the same way we pass an array to a function using pass by reference. To do so, simply declare the function parameters as a pointer during the function definition. Example: the following program segment sum up the values of all the elements of the array passed to it via a pointer.

```
/* Sum - sum up the ints in the given array */
int sum(int *ary, int size)
{
        inti, s;
        for(i = 0, s=0; i<size;i++){
                s+=ary[i];
        }
        return s;
}
/* In another function */
        int a[1000],x;
        ……
        x= sum(&a[100],50);
        /* This sums up a[100], a[101], …, a[149] */
```

Below is another example that uses a min-max function that returns the maximum and minimum of two numbers via pointer.

```
#include <stdio.h>
voidmin_max(int a, int b, int *min, int *max);
int main()
```

```
{
        intx,y;
        intsmall,big;
        printf("Two integers: ");
        scanf("%d %d", &x, &y);
        min_max(x,y,&small,&big);
        printf("%d <= %d", small, big);
        return 0;
}
voidmin_max(int a, int b,
                int *min, int *max){
        if(a>b){
                *max=a;
                *min=b;
        }
else{
                *max=b;
                *min=a;
        }
}
```

Dynamic Memory Allocation to Pointer [SAQ2]

 2 mins

Generally, memory can be allocated statically or dynamically. Static memory is allocated at compile-time, and their size cannot be extended during execution. Dynamic allocation, on the other hand, can be done at run time and size of memory can be extended even during execution. The memory is allocated from the heap. C language provides functions to handle dynamic memory allocation to pointers at run time. The functions are:

·**malloc():** allocates a single block of requested memory and does not initialize memory at run time. So it has garbage value initially. The syntax is given below: ***ptr = (cast-type *) malloc (byte-size);***

·**calloc():** allocates multiple blocks of requested memory, each of the same size and initializes all bytes to zero. It has the syntax: ***ptr = (cast-type *) calloc (number, byte-size);***

·**realloc():** reallocates the memory occupied by malloc() or calloc() functions. It has the syntax:

ptr = realloc (ptr, newsize);

·free(): frees the dynamically allocated memory. The syntax is given below:

free (ptr);

All the functions are defined in stdlib.h header file. Therefore,the header must be included in the program that uses the functions.

Example:

```
#include <stdio.h>

#include <stdlib.h>

int main(){

      int *p;= (int *) malloc( sizeof(int) );/* Allocate 4
bytes */

      scanf("%d", p);

      printf("%d", *p);

      free(p);     /*  This  returns  the  memory  to  the
system.*/

                   /* Important !!! */

      return 0;

}
```

Another example, finding prime numbers:

```
#include <stdio.h>
#include <stdlib.h>
/* Print out all prime
numbers which are less
than m */
voidprint_prime( int m )
```

```
{

inti,j;

int * ary = (int *) malloc( m * sizeof(int));



if (ary==NULL) exit -1;
for(i=0;i<m;i++)
ary[i]=1;
  /* Assume all integers between 0 and m-1 are prime */
ary[0]=ary[1]=0;
  /* Note that in fact 0 and 1 are not prime */
for(i=3;i<m;i++){
for( j=2; j<i; j++)
if(ary[ i ] &&i%j==0){
ary[i]=0;
break;
      }
  }
for(i=0;i<m;i++)
if(ary[i]) printf("%d ", i);
free(ary );
printf("\n");
}
int main() {
int m;
printf("m = ");
scanf("%d", &m);
printf("\n");
print_prime(m);
return 0;
}
```

Run the program to study the way it works

Strings [SAQ3,4 &5]

4 mins

Strings in C are simply arrays of characters. Example:

char s[10];

This is a ten (10) element array that can hold a character string consisting of 9 characters. This is because C does not know where the end of an array is at run time.By convention, C uses a NULL character '\0' to terminate all strings in its library functions

For example:

```
char  s[10] = {'u', 'n', 'I', 'x', '\0'};
```

It's the string terminator (not the size of the array) that determines the length of the string. String literals are given as a string quoted by double-quotes e.g.

```
printf("Long long ago.");
```

So a string can be initialized using the string lteral instead of the array initializerway .i.e

```
char s[10]="unix";  /* s[4] is '\0' */
char s[ ]="unix";     /* s has five elements */
```

Example:

```
charstr[]= "A message to display";
printf ("%s\n", str);
```

printf expects to receive a string as an additional parameter when it sees %s in the format string

·Can be from a character array.

·Can be another literal string.

·Can be from a character pointer

·printf knows how much to print out because of the NULL character at the end of all strings.When it finds a \0, it knows to stop

The most appropriate input/output functions to use with string are gets() and puts().

Example:

```
#include <stdio.h>
int main () {
       charlname[81], fname[81];
       int  count, id_num;

     puts  ("Enter  the  last  name,  firstname,  ID  number
     separated");
       puts ("by spaces, then press Enter \n");
       count = scanf ("%s%s%d", lname, fname,&id_num);
       printf ("%d items entered: %s %s %d\n",
             count,fname,lname,id_num);
       return 0;
}
```

Also, a string can be defined using a character pointer as follows

Char *name = "Adegunju";

Then the pointer can be used the same way we use the array of characters.

String Functions

String functions are provided in an ANSI standard string library. Access this through the include file:b#include<string.h>.

Includes functions such as;

·Computing length of string

·Copying strings

·Concatenating strings

·Comparing strings

·Converting strings to other types

This library is guaranteed to be there in any ANSI standard implementation of C.

strlen() :returns the length of a NULL terminated character string (excluding the null character) e.g.

```
Char *name = "Adegunju";
printf("the lenth of string %s is %d\n", name, strlen
(name));
```

Should output: the length of string adegunju is 8

strcpy(): Copying a string comes in the form:

char*

strcpy (char* destination, char* source);

A copy of the source is made at the destination. The source should be NULL-terminated and destination should have enough room (its length should be at least the size of source). The return value also points at the destination. E.g.

```
Char *name;
strcpy(name, adegunju);
```

strcat(): comes in the form:

char *strcat (char *str1, char *str2);

The function appends a copy of str2 to the end of str1. A pointer equal to str1 is returned. Ensure that str1 has sufficient space for the concatenated string!. If not array index out of range will be the most popular bug in your C programming career. Example:

```
#include <string.h>
#include <stdio.h>
int main() {
char str1[27] = "abc";
char str2[100];
printf("%d\n",strlen(str1));
strcpy(str2,str1);
puts(str2);
puts("\n");
strcat(str2,str1);
puts(str2);
return 0;
}
```

Comparing Strings

We can compare C strings for equality or inequality. If they are equal - they are ASCII identical. If they are unequal, the comparison function will return an int that is interpreted as:

<0 : str1 is less than str2

0: str1 is equal to str2

> 0: str1 is greater than str2

The four basic comparison functions are:

intstrcmp (char *str1, char *str2) ;

Does an ASCII comparison one char at a time until a difference is found between two chars and return value is as stated above. If both strings reach a '\0' at the same time,

they are considered equal.

intstrncmp (char *str1, char * str2, size_t n);

Compares n chars of str1 and str2 and continues until n chars are compared, or the end of str1or str2 is encountered

We also have **strcasecmp()** and **strncasecmp()** which do the same as above but ignore case in letters.

An example:

```
int main() {

char *str1 = "The first string.";

char *str2 = "The second string.";

printf("%d\n", strncmp(str1,str2,4) );

printf("%d\n", strncmp(str1,str2,5) );

      return 0;

}
```

Searching Strings

C provides us some functions to serach for a charcter or substring within a string. There are a number of searching functions:

·char * strchr (char * str, intch) ;

strchr search str until ch is found or NULL character is found instead.If found, a (non-NULL) pointer to ch is returned.Otherwise, NULL is returned instead.You can determine its location (index) in the string by subtracting the value returned from the address of the start of the string

Example:

```
#include<stdio.h>

#include<string.h>
int main() {
      charch='b', buf[80];
      strcpy(buf, "The quick brown fox");
      if (strchr(buf,ch) == NULL)
```

```
        printf ("The character %c was not found.\n",ch);
else
        printf ("The character %c was found at position
                        %d\n", ch, strchr(buf,ch)-buf+1);
return 0;
}
```

·char * strstr (char * str, char * query) ;

strstr searches struntil a sub-**str**ing query is found or a NULL character is found instead. If found, a (non-NULL) pointer to **str** is returned.Otherwise, NULL is returned instead.

Lastly C provides us special input/output functions to read from/write to strings. sscanf() and sprintf() functions. Example:

```
#include <stdio.h>
#include <string.h>
int main()
{
char result[100];
sprintf(result, "%f", (float)17/37 );
if (strstr(result, "45") != NULL)
printf("The digit sequence 45 is in 17divided by 37. \n");
return 0;
}
```

Summary

In this unit, I have taught you the following:

- An array is used to store a set of elements of the same type referenced with a name.
- The elements of the array can be accessed using the array name and the index of the element to be accessed.
- Array elements can be accessed automatically using loop statements.
- Enumeration types are user-defined types that are used to create constant names as a list of values internally represented as integers.

Self-Assessment Questions

SAQ

1. Identify how a pointer is used in C programming?
2. Describe the functions used for dynamic memory allocation to pointers in C
3. Describe a string.
4. Explain how a string is implemented in C
5. Design a simple C program to search for a substring within a 100 character long string.

Tutor Marked Assessment

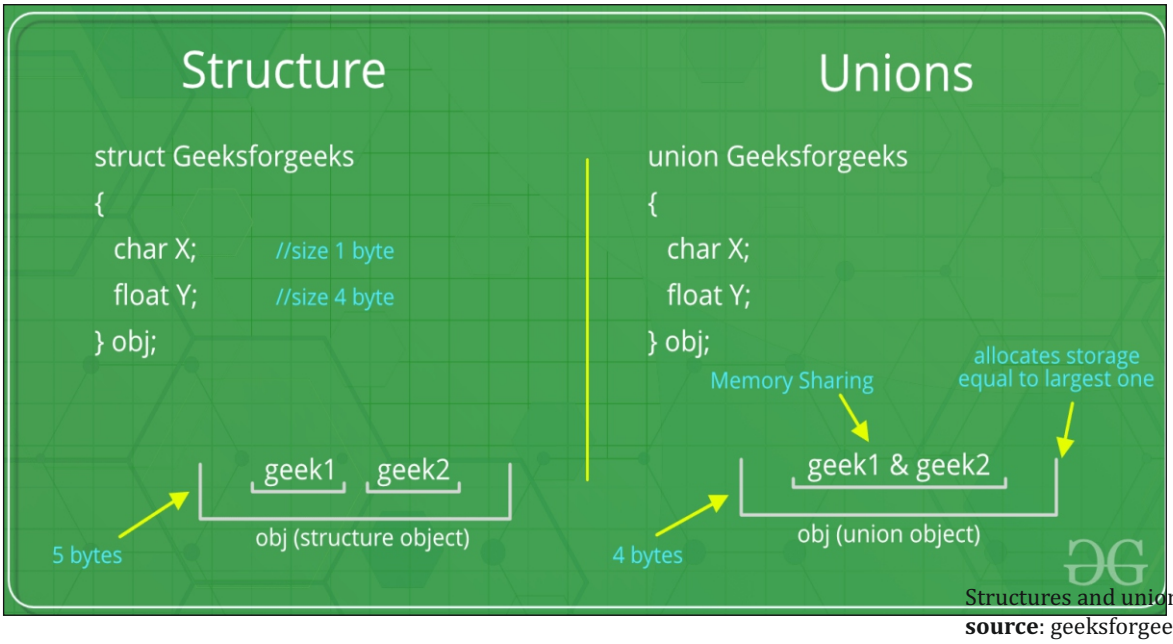
- Develop a program in C to record the biodata of a student using pointer and string manipulation.

Further Reading

- https://www.javatpoint.com/c-pointers
- http://www.javatpoint.com/c-programming-language-tutorial
- https://ict.senecacollege.ca/~ipc144/pages/content/point.html

References

- BalagurusamyE (2007).
- Programming in ANSI C, 4th Edition, McGraw-Hill
- Deitel, P. J. and Deitel, H.(2006).
- C-How to Program (5th Ed).
- Shola, P. B.(2002).
- Learn C-Programming Language (2nd Ed).
- Reflect Publishers, ISBN 978-047-174-X
- Prentice Hall, ISBN 978-0132404167



Structures and unions
source: geeksforgeeks.com

UNIT 3

Structure and Union

Introduction

In this unit, I will discuss with you, the structure and union of the user-defined type. Their implementation and usage are discussed using programming examples. Differences between the two user-defined types are highlighted to conclude the unit.

Learning Outcomes

At the end of this unit, you should be able to:

1. Define the structure and union construct
2. Describe how structure can be used to store records
3. Implement programs that use structure and union to solve problems
4. Differentiate between structure and union

Structure [SAQ1,2&3]

It is important for you to know that Structure in C is a user-defined type used for grouping collections of data of different types into a single manageable unit. It allows us to hold a different element in an object. Each of the elements of the structure is called a member. It is similar to a class in C++ and Java. To define a structure, we use the following syntax:

structstruct_name{

datatype varname1;

datatype varname2;

.

.

datatypevarnamen;

};

Like enum type, this defines a new type ofstructstrudt_name. The datatype can be of different types. For example:

```
structcoord

{
short x ;
int y ;

};
```

This defines a new struct type structcoord composed of two members; short x and int y respectively.

The structure is not useful until a variable of that struct type is created. The syntax is as follows:

structstruct_namevariable_name;

e.g. for the structcoord defined above,

```
structcoord point;
```

creates a variable point of the type structcoord and associate unique members of the structure (x and y) to it.Like in enumeration, the definition and declaration can be combined as follows

```
structcoord {
     short x ;
int y ;
} point;
```

Accessing Members of a Structure

Members of a structure can be accessed from the structure variable using they the dot (.) operator

Generic form:

structure_var.member_name

For example:

```
point.x = 50;
point.y = 100;
```

These member names are like the public data members of a class in Java (or C++).struct_var.member_name can be used anywhere; a variable can be used. For example:

```
          printf ("%d , %d", point.x , poin.y );
          scanf("%d %d", &point.x, &point.y);
#include <stdio.h>

struct data {

float amount;

char  fname[30];

charlname[30];

}  rec;

int main () {

     struct data rec;

     printf ("Enter the donor's first and last names, \n");

     printf ("separated by a space: ");
```

Any other types can be a member of the structure, including array and pointer. Arrays within structures are the same as any other member element. For example:

```
     scanf  ("%s %s", rec.fname, rec.lname);

     printf ("\nEnter the donation amount: ");

     scanf  ("%f", &rec.amount);

     printf ("\nDonor %s %s gave $%.2f.\n",

          rec.fname,rec.lname,rec.amount);

     return 0;

}
```

Initializing a Structure

We use a structure to initialize using a comma-separated list of the initial value of the members enclosed in the curly bracket

```
struct sale {
     char  customer [20] ;
     char  item [20] ;
     int amount ;
     };
struct sale mysale = { "Acme Industries",
               "Zorgle blaster",
                  1000
                    } ;
```

Simple example:

```
struct sale {
     char  customer [20] ;
     char  item [20] ;
     int amount ;
     };
```

```
struct sale sale1, mysale = { "Acme Industries",
                "Zorgle blaster",
                   1000
                     } ;
     sale1 = mysale;
```

you should note here that a structure can be assigned to another structure directly using the assignment operator (=). For example:

```
#include <stdio.h>

struct data {

float amount;

char  fname[30];

charlname[30];

}  rec;

int main () {

     struct data rec;

     printf ("Enter the donor's first and last names, \n");

     printf ("separated by a space: ");

    scanf  ("%s %s", rec.fname, rec.lname);

    printf ("\nEnter the donation amount: ");

    scanf  ("%f", &rec.amount);

    printf ("\nDonor %s %s gave $%.2f.\n",

          rec.fname,rec.lname,rec.amount);

    return 0;

}
```

Initializing a Structure

We use a structure to initialize using a comma-separated list of the initial value of the members enclosed in the curly bracket.

```
struct sale {
      char  customer [20] ;
      char  item [20] ;
      int amount ;
      };
struct sale mysale = { "Acme Industries",
                "Zorgle blaster",
                   1000
                     } ;
```

Simple example

you should note that here that a structure can be assigned to another structure directly using the assignment operator (=). For example:

```
struct sale {
      char  customer [20] ;
      char  item [20] ;
      int amount ;
      };
struct sale sale1, mysale = { "Acme Industries",
                "Zorgle blaster",
                   1000
                     } ;
      sale1 = mysale;
```

Array of Structures

Multiple structure variables can be created using an array of structures. This is the converse of array and structure. The following example uses an array as

```
struct  entry {
      char  fname [10] ;
      char  lname [12] ;
      char phone [8] ;
} ;
struct entry list [1000];
```

member of structure and createsan array of structures to use the member.

This creates a list of 1000 identical entry(s).

Let us consider the following assignments:

```
list [1] = list [6];
strcpy (list[1].phone, list[6].phone);
list[6].phone[1] = list[3].phone[4] ;
```

program example on using array of structures is given below:

```
for (i=0; i< 4; i++) {
printf ("Name: %s %s", list[i].fname, list[i].lname);
printf ("\t\tPhone: %s\n", list[i].phone);
   }
return 0;
}
#include <stdio.h>
struct entry {
```

```
charfname [20];
charlname [20];
char phone [10];
} ;
int main() {
struct entry list[4];
inti;
for (i=0; i< 4; i++) {
printf ("\nEnter first name: ");
scanf  ("%s", list[i].fname);
printf ("Enter last name: ");
scanf  ("%s", list[i].lname);
printf ("Enter phone in 123-4567 format: ");
scanf  ("%s", list[i].phone);
   }
printf ("\n\n");
```

Now you Run the program and study its output.

Structures containing Structures

 1 min

Structures can be nested or contain variables of another struct type. For instance, we can use the coordstruct previously defined to define a rectanglestruct type as follows:

```
struct rectangle {
     structcoord topleft;
     structcoord bottomrt;
} ;
```

This describes a rectangle by using the two points necessary; members can then be accessed as thus:

```
struct rectangle mybox ;

mybox.topleft.x = 0 ;

mybox.topleft.y = 10 ;

mybox.bottomrt.x = 100 ;

mybox.bottomrt.y = 200 ;


#include <stdio.h>

structcoord {

short x;

int y;

};

struct rectangle {

structcoordtopleft;

structcoordbottomrt;

};

int main () {

     int length, width;

     long area;

     struct rectangle mybox;

     mybox.topleft.x = 0;
```

```
mybox.topleft.y = 0;

     mybox.bottomrt.x = 100;

     mybox.bottomrt.y = 50;

     width = mybox.bottomrt.x –mybox.topleft.x;
```

A program example is given below:

```
length = mybox.bottomrt.y – mybox.topleft.y;

     area   = width * length;

     printf ("The area is %ld units.\n", area);

     return 0;

}
```

Passing Structures to Functions(1min)

We pass structure by value to function. The parameter variable is a local variable, which will be assigned by the value of the argument passed. This means that the structure is copied if it is passed as anargument. Although this can be inefficient if the structure is big.In this case, it may be more efficient to pass a pointer to the structure.A struct can also be returned from a function.

Example

```
structpairInt {

     int min, max;

};

structpairIntmin_max(intx,int y)

{

     structpairInt pair;

     pair.min = (x > y) ?y : x;

     pair.max = (x > y) ?x : y;

     returnpairInt;

}
```

```
int main(){
      structpairInt result;
      result = min_max( 3, 5 );
      printf("%d<=%d", result.min, result.max);
      return 0;
}
```

Union

Like structure, a union is also a user-defined datatype that is usedto hold different types of elements. Unlike structure, union occupies the memory of the largest members-only.i.e. all elements ina union shares the memory of the largest member. It has the construct:

unionstruct_name{

datatype varname1;

datatype varname2;

.

.

datatypevarnamen;

};

Then the union variable can be created as

unionstruct_namevariable_name;

For example:

```
union car{
char name[50];
int price;
};
```

```
int main()
{
union car car1, car2, *car3;
return 0;
}
```

Definition and declaration can be combined as well:

```
union car
{
char name[50];
int price;
} car1, car2, *car3;
```

Members of a union are accessed like structures using the dot (.) operator.

Differences between Structure and Union [SAQ4]

Let's take an example to demonstrate the difference between unions and structures:

```
#include <stdio.h>
unionunionJob
{
   //defining a union
char name[32];
float salary;
intworkerNo;
} uJob;
structstructJob
{
char name[32];
```

```
float salary;

intworkerNo;

} sJob;


int main()

{

printf("size of union = %d bytes", sizeof(uJob));

printf("\nsize of structure = %d bytes", sizeof(sJob));

return 0;

}
```

Output

```
size of union = 32

size of structure = 40
```

The difference in the size is because union elements share one memory of the largest member.

Another important way in which union is different from the structure is that; you can access all members of a structure at once as sufficient memory is allocated for all members. However, it's not the case in unions. You can only access a single member of a union at one time. Let's see an example.

```
#include <stdio.h>

union Job

{

float salary;

intworkerNo;

} j;

int main()

{
```

```
j.salary = 12.3;

j.workerNo = 100;


printf("Salary = %.1f\n", j.salary);

printf("Number of workers = %d", j.workerNo);

return 0;

}
```

Output

```
Salary = 0.0

Number of workers = 100
```

Summary

In this unit, I have taught you the following:

- Structure and Union are user-defined functions that can be used to store a group of objects of different types
- Elements of the structure and union can be accessed using the dot(.) operator
- Structure and union can be composed of other types, including composite types.
- The structure is different from the union in terms of the size of memory used to store the types and the way the members are accessed.

Self-Assessment Questions

SAQ

1. Using a code example, show how to define the structure and union types in C.
2. Define a structure that can be used to store information about books in the library
3. Discuss how structures are implemented in C
4. Distinguish between structure and union using code examples.

Tutor Marked Assessment

- Construct a program in C to record student information using structures.

Activity

- Define a structure that can be used to store information about books on a shelf
- What are the differences between structure and union using code

Further Reading

- https://www.programiz.com/c-programming/c-unions
- https://ict.senecacollege.ca/~ipc144/pages/content/point.html

References

- Prentice-Hall, ISBN 978-0132404167
- Deitel, P. J. and Deitel, H.(2006).
- C-How to Program (5th Ed).
- Shola, P. B.(2002).
- Learn C-Programming Language (2nd Ed).
- Reflect Publishers, ISBN 978-047-174-X