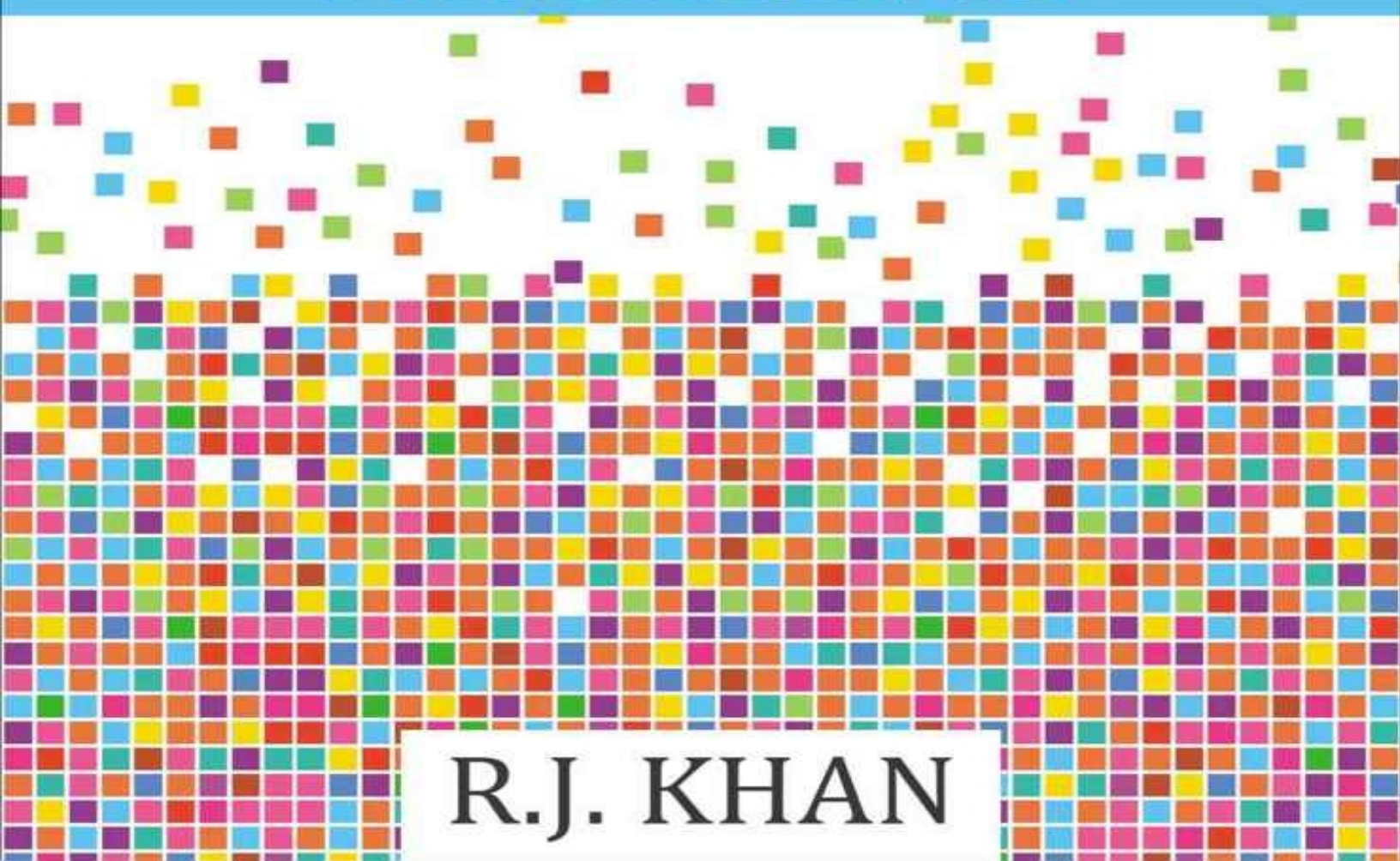
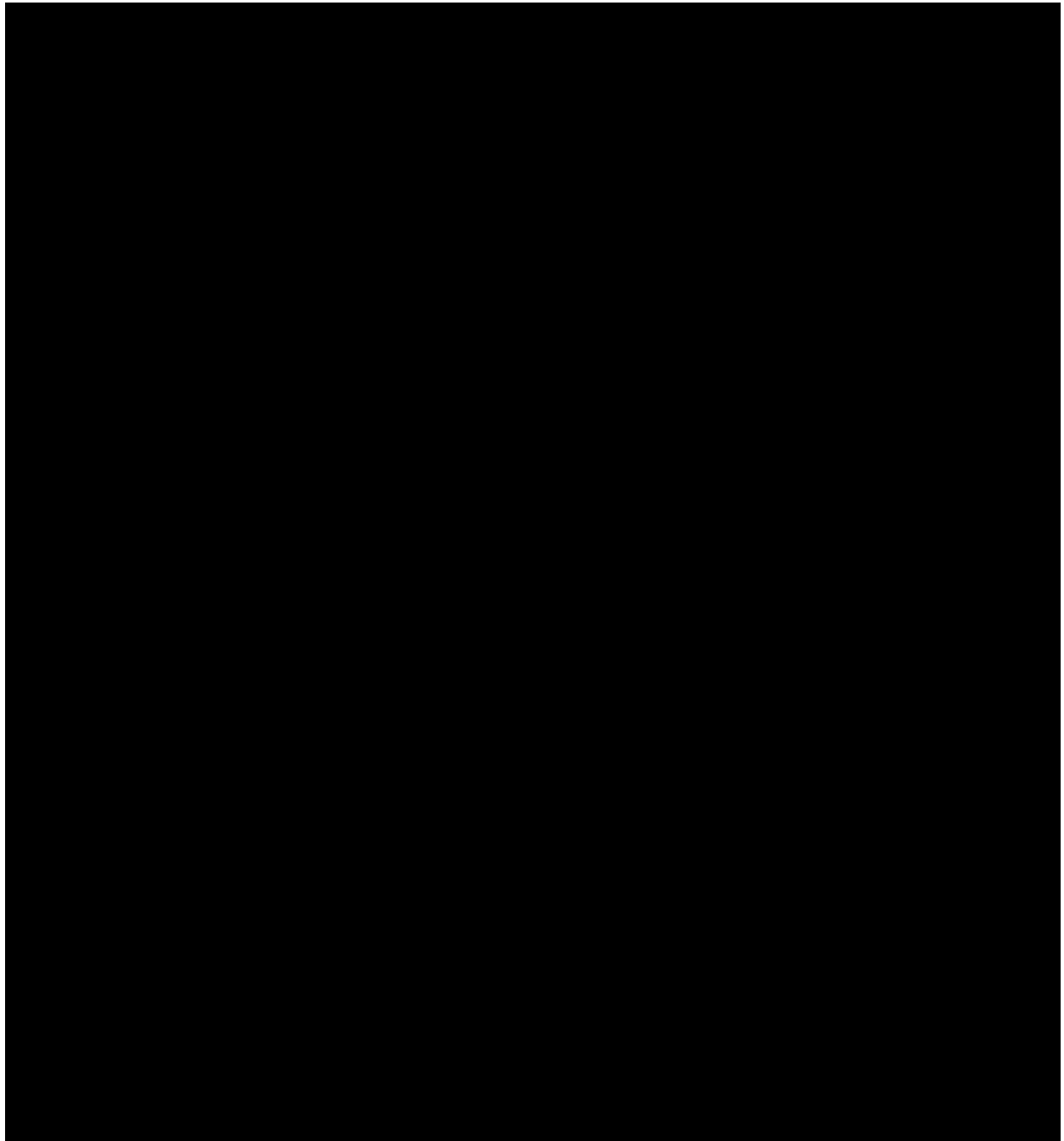


PROGRAMMING LANGUAGE QUICKSTART GUIDE

SIMPLIFIED GUIDE
FOR BEGINNERS



R.J. KHAN



Programming Language

Quick Start Guide

Simplified Guide for Beginners

By:

R.J. KHAN



Introduction

Are you looking for a simple, complete and quick reference for programming in C? If yes, you'll find this book more useful than any other book out there. The book presents the general-purpose language, step by step. The book is written following a programmer's approach and is therefore ideal for using as a quick guide and reference.

The book is highly recommended for students as their first or second programming course. It covers all the basic concepts of programming in detail like Operators, Functions, Decision making statements, Pointers, Arrays and Strings. The best thing about the book is the Source code along with output at the initial stages of book. Other than students, the book is pretty helpful for professionals as well since it has a thorough and detailed indexing of topics in each chapter.

The book allows the audience to be selective by reducing the inter-connectivity in the topics. We start with an overview of whole language, which involves brief history, basic features of C, comparison and contrast with other languages.

Then we move on to familiarizing the readers with C programs and syntax. We discuss the important components of syntax individually and in detail. This section has source code in it, which helps building the interest, and learning of the audience.

We study in detail about the Functions in C. The book discusses the built in functions as well as describes the rules and basics of writing your own functions. The chapter also includes programming examples.

Arrays, pointers and Strings are strong points of C language and hence are discussed separately in one chapter. The topics are closely related, hence comparisons and contrasts are worth the discussion.

We have an entire chapter dedicated to Binary Trees because of their importance. We discussed Abstract Data types as well as studied the examples of Stacks, Queues and Linked lists.

The book illustrates the general purpose use of programming language C,

detailed documentation of features provided by C, the working source code, along with comparison between object oriented language, C++. The differences between ANSI C

and traditional language “C” are elaborated in detail as well.

R.J. Khan

Copyright 2015 by R.J. Khan All rights reserved.

This document is geared towards providing exact and reliable information in regards to the topic and issue covered. The publication is sold with the idea that the publisher is not required to render accounting, officially permitted, or otherwise, qualified services.

If advice is necessary, legal or professional, a practiced individual in the profession should be ordered.

- From a Declaration of Principles which was accepted and approved equally by a Committee of the American Bar Association and a Committee of Publishers and Associations.

In no way is it legal to reproduce, duplicate, or transmit any part of this document in either electronic means or in printed format. Recording of this publication is strictly prohibited and any storage of this document is not allowed unless with written permission from the publisher. All rights reserved.

The information provided herein is stated to be truthful and consistent, in that any liability, in terms of inattention or otherwise, by any usage or abuse of any policies, processes, or directions contained within is the solitary and utter responsibility of the recipient reader. Under no circumstances will any legal responsibility or blame be held against the publisher for any reparation, damages, or monetary loss due to the information herein, either directly or

indirectly.

Respective authors own all copyrights not held by the publisher.

The information herein is offered for informational purposes solely, and is universal as so. The presentation of the information is without contract or any type of guarantee assurance.

The trademarks that are used are without any consent, and the publication of the trademark is without permission or backing by the trademark owner. All trademarks and brands within this book are for clarifying purposes only and are the owned by the owners themselves, not affiliated with this document.

TABLE OF CONTENTS

Chapter 1

<i>Let's C.....</i>	6
1. Background.....	6
2. ANSI C.....	7
3. Strengths of C.....	8
4. Programming in C.....	9
5. Nothing is Perfect.....	10

Chapter 2

Tokens and Syntax.....11

1. Preprocessors.....	11
2. Variables.....	14
3. Constants.....	16
4. Keywords.....	17
5. Character Constants.....	18
6. Operators and Punctuators.....	23
7. Precedence and Associativity.....	33

Chapter 3

<i>Data Types</i>	37
1. Integral Data Types.....	38
2. Floating point.....	38
3. Using ‘typedef’	39
4. Storage classes.....	39
5. Default Initializers.....	43

Chapter 4

<i>Loops and Decisions.....</i>	<i>44</i>
1. If else.....	44
2. Switch case.....	46
3. For loop.....	48
4. While loop.....	50
5. Do while loop.....	51
6. Nested Loops.....	52

Chapter 5

<i>Functions</i>	54
1. Function Definition.....	54
2. Function Prototypes	57
3. Alternative Declarations.....	58
4. Function Invocation.....	59
5. Environment.....	60
6. Recursion.....	61
7. Examples.....	62

Chapter 6

Arrays, Strings and Pointers.....66

1. 1-D Arrays.....66

2. Pointers.....68

3. Strings.....72

4. 2D-Arrays.....75

5. 3D-Arrays.....76

Chapter 7

<i>Structures</i>	77
1. Declaration.....	77
2. Memory Allocation.....	78
3. Member Access operators.....	79
4. Definition.....	80
5. Abstract Data Types.....	81

Chapter 8

<i>Binary Search Tree</i>	90
1. Composition.....	90
2. Insert Function.....	91
3. Search Function.....	92

Conclusion



Chapter 1: Let's C

This chapter gives an overview of the C programming language. We start with a brief background and discuss pros and cons of the general-purpose language. After that, a series of programs follows, and the elements of each program are carefully, line-by-line explained. We emphasize on how to use the basic input/output functions of C. As we discuss in detail the similarities between C and C++, you can fairly assume that the programs written are syntactically correct for both the languages and the concepts elaborated are also common. Of course, the C++ programmer has more capabilities using the object oriented programming concepts. We recommend that everyone reads this chapter out so that they know the basics about C and programming in C. Everyone should read this chapter even if they don't understand everything because everything is explained in detail later on, in the book but the introduction in this chapter is important.

1.1 Background:

C is a procedural programming language that was originally designed by Dennis Ritchie at Bell in 1972. UNIX operating system used it as the systems language. C was designed to overcome the shortcomings of a programming language B, which was designed by K.

Thompson while working on initial versions of UNIX. B was a programming language based on BCPL, a type less systems programming language, developed by Martin Richards in 1967. Its basic data type was the machine word, and it made heavy use of pointers and address arithmetic. It is completely different from traditional structural programming, which involved use of typed commands. Although C initially evolved from Band BCPL, but it included typing.

By the early 1980s, the language has transformed itself into what it looks like today, commonly known as the traditional C. ANSI standard C was introduced in the later part of the same decade. The committee added the void, function prototypes, a new function definition syntax, and more functionality for the preprocessor, and in general made the language definition more precise.

ANSI C maintained its standard in an amazing way that it is still used in many programs as a system language. It is one of the most commonly used programming languages of the world, and it is found in colleges and universities for learning purposes. It is mostly the first programming course offered to any IT students. C++ is an extension of ANSI standard C which incorporates the object oriented programming paradigm, which kind of revolutionized the programming world. This book describes the ANSI version of the C

language, along with some topics in C++ as well.

1.2 ANSI Standard C:

ANSI stands for "American National Standards Institute." This institute is involved in setting standards for different systems, including programming languages. Standards

are important in a sense that everyone can have his own perspective. For

example, while travelling in a train, what seems to be stationary for you, might be moving for someone outside the train. ANSI Committee X3J11 is responsible for setting the standard for the programming language C. In the late 1980s, the committee created draft standards for what is known as ANSI C or standard C. In 1990, the International Organization for Standardization (ISO) approved the standard for ANSI C as well.

Thus, ANSI C, or ANSI/ISO C, is an internationally recognized standard for programming in C. The standard specifies the form in which programs are written in C

and defines the standards for how these programs are to be interpreted. The standardization results in better portability, reliability, maintainability, and efficient execution of C language programs on different machines. All the compilers follow this standard and hence there is uniformity in interpretation of code.

1.3 Strengths of C:

The programming languages that are being used today are so powerful that nothing remains impossible. But this power depends directly on how vast a language is, which is determined by number of reserved keywords. C has fewer keywords than Pascal, but it is more powerful than it. The power of C lies in the unrestricted structure of programming in it. Although, C has its limitations like any other language but it comes as the best in its capacity. Because C is a small language, hence it is ideal to start your programming career from it. It is easy to master when compared with other languages. Once you have mastered C, you can then easily move on to the more powerful and vast languages.

We will now briefly discuss the strong characteristics of C, which will motivate the readers to program in C.

- UNIX is a major interactive operating system on workstations, servers, and mainframes. C is the basic language used on UNIX platform. Many windowing packages, database programs, graphics libraries, and other large-application packages are written in C.

- C is portable. Code written on one machine can be easily moved to another. The programmer is provided with a standard library of functions that work independently of machines. Even if there is a system dependent code, the

programmer is given the capability to single it out using preprocessor provided by C.

- Personal Computers used C as their programming language.
- C is small but has a very powerful set of operators. Some of these operators allow the programmer to access the machine at the bit level. For example, a powerful increment operator “++” can manipulate the machine in the same way as machine code does. This results in higher efficiency as other languages require multiple statements to complete the equivalent task.
- C is modular. C uses the external functions, for which arguments are passed using call-by-value technique. Pointers are used for passing arguments using call-by-reference technique. The nesting of functions is not allowed. Using the storage class static within files provides a limited form of privacy. These features, along with tools provided by the operating system, readily support user-defined libraries of functions and modular programming.
- C laid the basis for C++ and Java. This can be clearly seen by analyzing the common attributes and functionalities of these languages. C++ and Java support the basic structure of functions, methods or routines in C++ or Java. The primitive data types are retained as well. Thus, learning C prior to learning C++ or Java is a good thing.
- Certain constructs in C are dependent on machines, implementation of these constructs is up to the programmer to support the targeted machine(s). Thus, C can support multiple machines.

1.4 Programming in C: Programs are set of instructions that are executed by a computer processor. Programs are, at the lowest level, combinations of 0s and 1s. These combinations are commonly known as machine code. The 0s and 1s are commonly known as bits or binary digits.

Machine language is closest to machine but it is away from human beings in a way that it is very difficult to remember sequences of 0s and 1s. Therefore we use low-level languages like Assembly Language and High Level languages like

C, C++ and Java.

Assembly language has short labels for certain commands to do certain tasks, for example ADD, MOV *etc.* High Level languages are the easiest way for a programmer to communicate with a machine.

C is a high level language. It is important to know that computer only understands machine language. Assembly language and High-level languages are used for aiding programmers rather than computer. The code written by humans in assembly and high-level languages is translated into its binary form so that computer can understand it.

The source code for C is written in some text editor and compiler. Common examples of such editors are Turbo C++, Borland C++ *etc.* We will be using Visual Studio 2012 for coding throughout in this book.

1.4.1 First Program in C

First of all, you need to set up your IDE (Integrated Development Environment). We recommend Borland C++. Visual Studio is for pros but is not a good choice for students because of its helpful nature. From here onwards, we assume that you have an IDE installed on your computer system and can compile a source code in C.



```
This is my first Program in C
Press any key to continue . . .
```

You need to create a new file with .c or .cpp extension. This file will have your source code, which will be compiled using your respective compiler. Let's name our file as "myFirstProgram.c".

In "myFirstProgram.c":

```
#include <iostream>
```

```
void main(void)

{

printf("This is my first Program in C\n"); }
```

Before going any further, let's just try to compile and execute this code. On doing so, you'll realize that a black screen appears for a millisecond and then disappears.

What actually happens is, that the computer runs the program and exits when the source code ends. So it exits before us seeing any of the output. To stop our program from doing it, we need to add an extra line at the end of our code.

```
#include <iostream>

void main(void)

{

printf("This is my first Program in C\n");

system("pause");

}
```

If
no
errors
are
made,
the
output

will

be

like

this:

The first line of our program includes a library in our program. This library has the functions like printf, system and similarly others which we can use in our program.

The next line declares our main method. The main method in any program is the entry point for our logic. Programs start from executing this function. The word “void” is used to indicate that the parameter under consideration is null and void.

The first void represents that the main function returns nothing. The next void inside parenthesis indicates that the main function is passed nothing at the start.

We’ll see how these two void types can be replaced by other data types as demanded by our program logic.

It is important to note that braces are used to represent blocks in our code. For now, you need to know that the body of our main function lies within braces. Anything outside these braces is not executed when main function is executed.

Inside our main function, we have two statements; printf and system. Printf takes a String, a stream of characters as input and displays it on the screen. The ‘\n’ at the end of our line in printf statement indicates the start of new line. The next line uses the system to pause the program, so that we can see the output as explained earlier.

A semi-colon ends each statement. Semi-colon is commonly known as statement terminator.

So, we stop here for now. Coding or programming isn’t a difficult thing at all.

All it demands is a little practice. We strongly believe you will be an efficient coder in C

after finishing this book.

1.5 Nothing is perfect: Like every other language, C has its shortcomings. It has a complicated syntax. For example, a common programming error is to use different operator instead of the operator `==`. Nevertheless, C is still a decent language. It doesn't restrict the programmer to access the machine. Some can consider the lifting of restrictions as an imperfection, but I personally think it has more pros than cons. C attracts programmers because of these "imperfections". Modularity in the source code is often considered critical. In C, it is actually dependent on programmers, which always prefer that there is minimal dependency between functions. A C programmer welcomes experimentation and interaction as well, depending upon certain situations.



Chapter 2: Tokens and Syntax

In this chapter, we explain the syntax and lexical elements of the C programming language. C is a high level programming language. Like other high level languages, it has rules and principles for putting together different components to create legal and working programs. These rules and principles define the syntax of the language.

The program that checks on the legality of C code is called the compiler. The compiler translates each line of code into machine language to see if there are any errors. If there is an error, the compiler will print an error message and stop. If the compiler finds no errors, then the source code is correct, and the code is translated into object code. The object code is used by loader to create an executable file.

As we discussed earlier, the preprocessors are processed prior to execution of remaining code. We can think of the preprocessor as being built into the compiler. This is what actually happens on some systems, whereas on others the preprocessor is separate. This might not concern us at this stage but we have to be aware, however, that preprocessors and compiler can both cause various errors in our programs. Let us discuss preprocessors in a bit of detail now.

2.1 Preprocessors C compiler has a preprocessor built into it. Lines that begin with a # are called preprocessors. We have so far used #define and #include. The compiler processes these preprocessors before actually compiling a code. For example, when we defined a constant using #define in section 2.2, the compiler replaced all the instances of PI with the value specified at the start before compiling the code. Such constants are usually known as Symbolic constants.

We write all the #define statements at the top of our file. It is a convention to write all the letters of a constant in capital. Following this convention enables programmers to easily identify Symbolic constants or any other constants as well. It is important to note that though, the preprocessor never changes the contents of quoted strings. For example, in the following statement, only second PI will be replaced by the value defined by preprocessor PI.

```
printf(" PI == %f\n ", PI);
```

The program becomes more readable by using preprocessors. It makes easy to follow the flow of program. More importantly, if a constant has been defined symbolically by means of the `#define` facility and used throughout a program, it is easy to change it later, if necessary. For example, the letter `c` is universally accepted to be used to represent the

speed of light, which is approximately 299792.458 km/sec. If we write the following preprocessor and then use the letter `C` throughout tens of thousands of lines of code to represent symbolically the constant 299792.458, it will be easy to change the code when a new physical experiment produces a better value for the speed of light. All the code is updated by simply changing the constant in the `#define` line. Such are the advantages of using preprocessors.

```
#define C 299792.458
```

`#include filename.h` is a preprocessing directive that causes a copy of the file `filename.h` to be included at the point in the file when it is compiled. A `#include` line can occur anywhere in a file, though it is typically at the head of the file as a convention. It is necessary to specify the filename in quotes as it indicates that it is a predefined file. The file specified in such directives, is also called a header file. These header files can contain `#define` lines and other `#include` lines. The names of header files end in `.h` extension.

The C system provides a number of standard header files. Some examples are `stdio.h`, `string.h`, and `math.h`. These files contain the declarations of functions in the standard library, macros, structure templates, and other programming elements that are commonly used. Common examples of such header files are `conio.h`, `stdio.h`, `stdlib.h`, `math.h` and others. These files contain certain system functions such as `stdio` provides the coder with functions as `printf` and `scanf`. We must include this header file before we can use these functions in our program. We, so far have used `iostream`, which is short form of input/output stream included using the `#include` directive.

Let us just see an example demonstrating how using a header file works. Following is code written in a header file named as “`myFirstHeaderFile.h`”

```
#include "iostream"
```

```
#define PI 3.14159265358
```

```
#define PI 3.14159265359
```

```
#define LENGTH 200
```

```
#define WIDTH 300
```

```
#define HEIGHT 400
```

```
#define PERIMETER 2400
```

```
#define OWNER "EMMA WATSON"
```

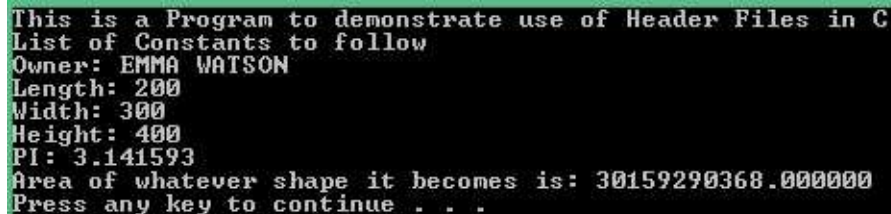
The following is the code to be written in a .cpp file which includes our header file in it.

If you are using Visual Studio, simply add the header file in the same project as the .c or .cpp) file is in.

```
#include "myFirstHeaderFile.h"
```

```
void main(void)
```

```
{
```

A screenshot of a terminal window with a black background and green text. The text displays the output of a C program, listing constants defined in a header file and the calculated area of a shape.

```
This is a Program to demonstrate use of Header Files in C
List of Constants to follow
Owner: EMMA WATSON
Length: 200
Width: 300
Height: 400
PI: 3.141593
Area of whatever shape it becomes is: 30159290368.000000
Press any key to continue . . .
```

```
float Area=PI*LENGTH*HEIGHT*HEIGHT*WIDTH;
```

```
printf("This is a Program to demonstrate use of Header Files in C\n");
```

```
printf("List of Constants to follow\n");
```

```
printf("Owner: %s\n", OWNER);
```



```
printf("Length: %d\n", LENGTH);
```

```
printf("Width: %d\n", WIDTH);
```

```
printf("Height: %d\n", HEIGHT);
```

```
printf("PI: %f\n", PI);
```

```
printf("Area of whatever shape it becomes is: %f\n", Area);
```

```
system("pause");
```

```
}
```

The output for the above program is as follows:

As you can see, the program works just as the code in the header file was written in our c file. This is just a simple example to demonstrate the use of header files. We actually use header files because programs become too complex and it's better to handle them in smaller files with lesser complexity. We divide the functionality into different categories and write code and functions for each category in a separate file. In this way, it becomes easier to digest each smaller chunk of a code.

A C program is a sequence of characters. This sequence of characters is transformed into object code by compiler. The object code is translated into a target language depending upon the machine. Target language, on most of the systems is the machine language that can be interrupted. The program must be correct in its syntax. The compiler takes each character as a token. Token is considered as the basic unit of vocabulary in the language.

In ANSI C, there are six kinds of tokens: keywords, identifiers, constants, string constants, operators and punctuators. We will discuss each of these tokens in this

chapter. The role of a compiler is to check if all the tokens can be translated into proper understandable strings according to the compiler requirements. Most compilers are very precise in their requirements.

Computers are dumb. We, the humans have the capability to understand a line more efficiently if an extra punctuation mark is used in a sentence but computers on the other hand will not be able to understand a sentence if it has an extra punctuation mark called token, it is not familiar with. Hence, the programmer must be familiar with which tokens can be used and which cannot be.

In the remaining part of this chapter, we will be covering the basic structure elements of any C program. Every large thing in the universe comprises of smaller components.

These smaller components might not be that effective individually, but assembling them in a right way certainly leads to a greater success.

The scenario mentioned above is true for the language C as well. C consists of individual components called Functions. Functions are considered as basic structural element in C

programming. The Functions themselves comprise of smaller components like Variables, constants, operators and other functions as well. We will study Functions in detail in another chapter. In this chapter, we will discuss these individual components one by one.

2.2 Variables

A variable name, also known as an identifier, consists of a sequence of letters, digits, and underscores, but may not start with a digit. These variables are usually declared at the start of program but you can declare one anywhere in your program before using it. You can store values in these variables depending upon their data types and “vary” them at any stage of your program. Identifiers should be chosen to reflect their use in the program. In this way, they serve as documentation, making the program more readable.

There is a code for everything we do in life. Similarly, we have rules for writing variable names in C. Variable names can be composed of lowercase and / or

variable name in C. variable name can be composed of uppercase and/or lowercase letters, digits and only one special character that is '_'. The first letter of a variable must be either a letter or an underscore. But, it is discouraged to start variable name with an underscore though it is legal because, variable name that starts with underscore can conflict with system names and compiler may complain. The length of a variable name has no restrictions. However, the first 31 characters of a variable are discriminated by the compiler. So, the first 31 letters of two variables in a program should be different by at least one character.

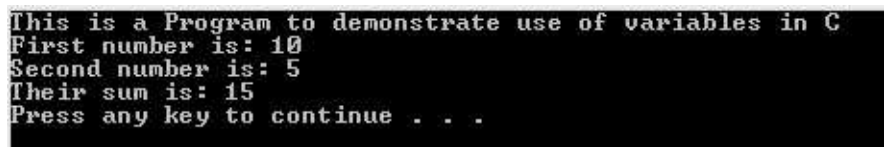
Let us get to know the use of variables by quoting some practical applications. Here is a program to calculate sum of two numbers: #include <iostream>

```
void main(void)
```

```
{
```

```
int firstNumber=10;
```

```
int secondNumber=5;
```



```
This is a Program to demonstrate use of variables in C
First number is: 10
Second number is: 5
Their sum is: 15
Press any key to continue . . .
```

```
int sum=firstNumber+secondNumber;
```

```
printf("This is a Program to demonstrate use of variables in C\n");
```

```
printf("First number is: %d\n", firstNumber);
```

```
printf("Second number is: %d\n", secondNumber);
```

```
printf("Sum of two numbers is: %d\n", sum);
```

```
printf("Their sum is: %d\n", sum);
```

```
system("pause");
```

```
}
```

The output for above program is as follows:

Some of you might be wondering why use the variables here instead of just using the numbers. That would be easy but it will have limited use as well as a bad programming approach. Let us add a few more lines to the above code to prove these two points: #include <iostream>

```
void main(void)
```

```
{
```

```
int firstNumber=10;
```

```
int secondNumber=5;
```

```
int sum=firstNumber+secondNumber;
```

```
printf("This is a Program to demonstrate use of variables in C\n");
```

```
printf("First number is: %d\n", firstNumber);
```

```
printf("Second number is: %d\n", secondNumber);
```

```
printf("Their sum is: %d\n", sum);
```

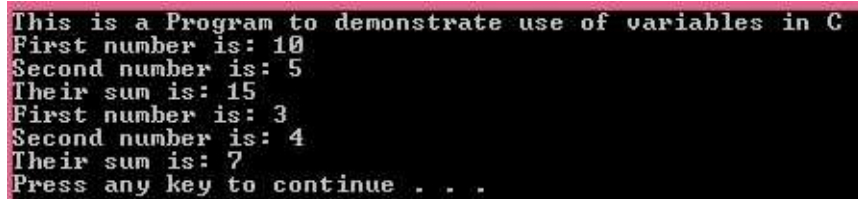
```
firstNumber=3;
```

```
secondNumber=4;
```

```
sum=firstNumber+secondNumber;
```

```
printf("First number is: %d\n", firstNumber);
```

```
printf("Second number is: %d\n", secondNumber);
```



```
This is a Program to demonstrate use of variables in C
First number is: 10
Second number is: 5
Their sum is: 15
First number is: 3
Second number is: 4
Their sum is: 7
Press any key to continue . . .
```

```
printf("Their sum is: %d\n", sum);
```

```
system("pause");
```

```
}
```

As you can see that the print statements are duplicated in the second code, but the values are changed by just changing the variables. So, we can use same code to get different output by just changing variables. This gives us a huge advantage. For example, consider a scenario that you have a list of 300 students with their obtained marks in a uniform C programming test. What would be the best way to represent total marks for this list? Would you recommend writing total marks with each entry? Absolutely not but just consider for a second that you do that and then you find out that total marks are 50

instead of hundred. You are bound to make a change i each entry. To avoid all this, you could just assign a value to a variable and use that variable in each entry. This is what variables are actually used for. Variables are a critical component of any program.

2.3 Constants

A constant is something whose value does not change throughout the program execution. These are fixed for a purpose though. For example, using standard values for certain entities, for having not to type a long digit or value like 3.141517... or making the program easy to follow by mentioning all the constant values in one place.

Let us just go through a C program using constants:

```
#include <iostream>
```

```
#define PI 3.14159265359
```

```
void main(void)
```

```
{
```

```
int radius=10;
```

```
float Area=PI*radius*radius;
```

```
printf("This is a Program to demonstrate use of Constants in C\n");
```

```
printf("Radius of circle is: %d\n", radius);
```

```
printf("Area of circle is: %f\n", Area);
```

```
This is a Program to demonstrate use of Constants in C
Radius of circle is: 10
Area of circle is: 314.159271
Press any key to continue . . .
```

```
system("pause");
```

```
}
```

We used preprocessor #define to declare a constant PI in this program. This is a flexible and recommended way to use constants in C. There is another way to use constants as well using the keyword 'const'. Let us see how that works too: #include <iostream>

```
void main(void)
```

```
{
```

```
int radius=10;
```

```
const float PI=3.14159265359;
```

```
float Area=PI*radius*radius;
```

```
printf("This is a Program to demonstrate use of Constants in C\n");
```

```
printf("Radius of circle is: %d\n", radius);
```

```
printf("Area of circle is: %f\n", Area);
```

```
system("pause");
```

```
}
```

2.4 Keywords

Keywords are reserved words in any language. These words are already reserved for a particular functionality so programmers cannot use them. These keywords represent primitive data types, pre-built functions and other programming components.

Here is a list of keywords in C, predefined by ANSI: **Keywords**

Functionality

auto

Defines a local variable as having a local lifetime.

break

Passes control out of the compound statement.

case

Used to program a logic based decision on input through Switch.

char

Used to define a character type variable.

const

Makes variable value or pointer parameter unmodifiable.

continue

Passes control to the beginning of the loop.

default

Used in switch-case statements for defining default operation.

do

This keyword along with 'while' is used for a special kind of loop.

double

This keyword is used for defining a variable of type 'double'.

else

Used along with 'if' for defining an alternate for a certain condition.

enum

Defines a set of constants of type integer.

extern

Used to indicate that a variable is defined outside current program.

float

Used to define a floating type variable.

for

Used to program a 'for' loop.

goto

Used to move compiler in a program, unconditionally.

if

Define a decision statement.

int

Defines an integer type variable.

long

Defines a long type variable of required type, *e.g.* int.

register

To store a value in a variable into a CPU register return

return control to the previous block in CPU register short

Used to modify a variable of data type mentioned with 'short'.

signed

Used as a data type modifier

sizeof

Returns the size of the expression or type.

static

Used to save a variable even after its scope ends.

struct

To define a customized Data Structure according to requirements.

switch

Used as a decision making statement, Switch-case.

typedef

Defines a new data type.

union

Groups variables that share same storage.

unsigned

Used as a data type modifier.

void

Empty data type.

volatile

Opposite of the keyword, 'const'.

while

Used to define repetition loops.

2.5 Character Constants This section addresses the character constants which have a certain meaning in certain situations. Character literals are enclosed in single quotes, e.g., 'a' and can be stored in a simple variable of **char** type.

A character literal can be a plain character, an escape sequence, or a universal character.

A closely related topic, string constant, or a string literal is a sequence of characters enclosed in a pair of double-quote marks, such as "Khan". Interestingly, a string constant consists of multiple characters but is taken as a single token by compiler. Strings are stored as arrays of characters. It is important to note though that string constants are different from character constants. For example, 'a' and "a" are not the same. "a"

consists of two characters actually. First one is 'a' and the next one is '\0'. The second one represents NULL.

C language provides certain character constants whose meanings are predefined like we have used "\n" a number of times to indicate the start of new line. Such characters are known as Escape Sequences. Remember, these characters and these meanings are applicable when written inside double quotes as a string. Here is a list of the common escape sequence characters:

Escape

sequence

Meaning

\\

\ character

\'

' character

\"

" character

\?

? character

\a

Alert or bell

\b

Backspace

\f

Form feed

\n

Newline

\r

Carriage return

\t

Horizontal tab

\v

Vertical tab

\ooo

Octal number of one to three digits

\xhh . .

Hexadecimal number of one or more digits Then we have ASCII characters, that are listed below: **Char Dec Action**

NUL

0

Null character

SOH

1

Start of heading, = console interrupt STX

2

Start of text, maintenance mode on HP console ETX

3

End of text

EOT

4

End of transmission, not the same as ETB

ENQ

5

Enquiry, goes with ACK; old HP flow control ACK

6

Acknowledge, clears ENQ logon hand

BEL

7

Bell, rings the bell...

BS

8

Backspace, works on HP terminals/computers HT

9

Horizontal tab, move to next tab stop LF

10

Line Feed

VT

11

Vertical tab

FF

12

Form Feed, page eject

CR

13

Carriage Return

SO

14

Shift Out, alternate character set

SI

15

Shift In, resume defaultn character set DLE

16

Data link escape

DC1

17

XON, with XOFF to pause listings

DC2

18

Device control 2, block-mode flow control DC3

19

XOFF, with XON is TERM=18 flow control DC4

20

Device control 4

NAK

21

Negative acknowledge

SYN

22

Synchronous idle

ETB

23

End transmission block, not the same as EOT

CAN

24

Cancel line, MPE echoes!

EM

25

End of medium, ControlY interrupt

SUB

26

Substitute

ESC

27

Escape, next character is not echoed FS

28

File separator

GS

29

Group separator

RS

30

Record separator, block-mode terminator US

31

Unit separator

The above is list of all the control ASCII characters. Following is the list of printable ASCII characters in ASCII.

Char Dec Description SP

32 Space

!

33 Exclamation mark

"

34 Quotation mark

#

35 Cross hatch (number sign)

\$

36 Dollar sign

%

37 Percent sign

&

38 Ampersand

,

39 Closing single quote (apostrophe) (

40 Opening parentheses

)

41 Closing parentheses

*

42 Asterisk (star, multiply)

+

43 Plus

,

44 Comma

-

45 Hyphen, dash, minus

46 Period

/

47 Slant (forward slash, divide)

0

48 Zero

1

49 One

2

50 Two

3

51

Three

4

52 Four

5

53 Five

6

54 Six

7

55 Seven

8

56 Eight

9

57 Nine

:

58 Colon

;

59 Semicolon

<

60 Less than sign

=

61 Equals sign

>

62 Greater than sign

?

63 Question mark

@

64 At-sign

A

65 Uppercase A

R

⌞

66 Uppercase B

C

67 Uppercase C

D

68 Uppercase D

E

69 Uppercase E

F

70 Uppercase F

G

71

Uppercase G

H

72 Uppercase H

I

73 Uppercase I

J

74 Uppercase J

K

75 Uppercase K

L

76 Uppercase L

M

77 Uppercase M

N

78 Uppercase N

O

79 Uppercase O

P

80 Uppercase P

Q

81 Uppercase Q

R

82 Uppercase R

S

83 Uppercase S

T

84 Uppercase T

U

85 Uppercase U

V

86 Uppercase V

W

87 Uppercase W

X

88 Uppercase X

Y

89 Uppercase Y

Z

90 Uppercase Z

[

91 Opening square bracket

\

92 Reverse slant (Backslash)

]

93 Closing square bracket

^

94 Caret (Circumflex)

—

95 Underscore

96 Opening single quote

a

97 Lowercase a

b

98 Lowercase b

c

99 Lowercase c

d

100 Lowercase d

e

101 Lowercase e

f

102 Lowercase f

g

103 Lowercase g

h

104 Lowercase h

i

105 Lowercase i

j

106 Lowercase j

k

107 Lowercase k

l

108 Lowercase l

m

109 Lowercase m

n

110 Lowercase n

o

111 Lowercase o

p

112 Lowercase p

q

113 Lowercase q

r

114 Lowercase r

s

115 Lowercase s

t

116 Lowercase t

u

117 Lowercase u

v

118 Lowercase v

w

119 Lowercase w

x

120 Lowercase x

y

121 Lowercase y

z

122 Lowercase z

{

123 Opening curly brace

|

124 Vertical line

}

125 Closing curly brace

~

126 Tilde (approximate)

DEL 127 Delete (rubout), cross-hatch box

2.6 Operators and Punctuators Like every other programming language, C has

many characters with certain meanings.

Examples include the Arithmetic operators which stand for the usual arithmetic operations of addition, subtraction, multiplication, division, and modulus, respectively.

Modulus in mathematics returns the remainder after dividing a number by another. For example, the value of “a modulus b” is obtained by taking the remainder after dividing a by b. % is the character used for Modulus operation in C. Thus, 15%7 has the answer 1, and 7 % 5 results in 2.

In a program, operators can be used to separate identifiers that are involved in an expression. Remember, an expression is a statement that ends with a semicolon. It is good practice to give whitespaces between each identifier and operators to give better readability but it is not necessary. Compiler ignores Whitespaces unless they are quoted in the code. So, the following statements are identical to the compiler, X=A+B;

```
X = A + B ;
```

Context is very important in almost everything. Symbols change their meaning depending upon where and how they are used. For example consider the following two statements which use % symbol in two different contexts.

```
printf("Owner: %s\n", OWNER);
```

```
int x = 15 % 4 ;
```

The first % symbol is the start of a conversion specification, or format which is related to the identifier in the later part of the expression, whereas the second % symbol represents the modulus operator of mathematics.

Examples of punctuators include parentheses, braces, commas, and semicolons.

Consider the following code:

```
void main(void)
```

```

    {
int a, b = 2, c = 3;

a = (b * (b + c));

    }

```

The parenthesis used right after main are used as an operator. By rules, they are used to indicate that main is a function. After the parenthesis, the symbols “{” , “,” , “;” , “(” , “)” and “;” are all punctuators. Both operators and punctuators are collected by the compiler as tokens (characters), and along with white space, they serve to separate language elements like identifiers, constants, functions, keywords, operators and others.

Tokens depend on their context a lot. For example, as explained in the previous paragraph, the parentheses are sometimes used to indicate a function name while they can be used as punctuators as well. Another example is given by the expressions $C = a + b$;

```

++ a ;

a + = b ;

```

They all use + as a character, but ++ and += are increment and assignment operators respectively while the first one is a simple arithmetic operator. This dependency of symbols meaning on context leads to the short but more effective programming language C.

C language has rules of precedence and associativity in operators. It is critical in nature as we do not want one statement to result in two different outputs. These rules do not fully determine evaluation because we give user the power to achieve the result of desire by mastering these rules. Since expressions inside parenthesis are evaluated first, parentheses can be used to clarify or change the order operations are performed.

Consider the expression:

Answer = $30 + (4 * 10 / 5 - 20) ;$

The operator “(” has the highest parenthesis in this expression so we dissect it before anything else. Then we have three operators inside parenthesis. Multiplication and division have equal precedence so we perform whatever comes first in the expression. In this case, we perform $4 * 10$ and have a result of 40. We divide 40 by 5 and result is 8.

We subtract 20 from 8 and the result is -12. Now that parenthesis are closed, we compute the operations outside it. We add -12 to 30 and the result is 18.

Humans with good mathematics will always find the answer to be 18. But guys, who are not familiar with DMAS rule will have different answers. We want our computer to be good with mathematics, so we have made it familiar with the precedence of operators in the standard way so it always leads to one correct answer.

Let us now discuss various types of operators provided by language C: **2.6.1 Arithmetic and Bitwise Operators**

Operator Name

Operations

+

Add

Used to add two numbers

-

Subtract

Used to subtract two numbers

*

- - - - -

Multiply

Used to multiply two numbers

/

Divide

Used for division of two numbers

%

Modulus

Used to find remainder after division of 2 numbers ~

Negation

Binary operator, used to invert the boolean value &

Bitwise AND Used to perform logical AND on every bit |

Bitwise OR

Used to perform logical OR on every bit ^

Power

Used to compute power of a number

<<

Left Shift

Used to shift bits to the left

>>

Right Shift

Used to shift bits to the tight

Let us now see an example of using these operators in a program: #include "iostream"

```
void main(void)
```

```
{
```

```
int A = 50, B = 17;
```

```
bool Alive =false;
```

```
printf("This is a Program to demonstrate use of Arithmetic Operators in C\n");
```

```
printf("Sum: %d\n", A + B);
```

```
printf("Difference: %d\n", A - B);
```

```
printf("Product: %d\n", A * B);
```

```
printf("Division: %d\n", A / B);
```

```
printf("Modulus: %d\n", A % B);
```

```
printf("Negation: %s\n", ~Alive? "true": "false");
```

```
printf("Bitwise AND: %d\n", A & B);
```

```
printf("Bitwise OR: %d\n", A | B);
```

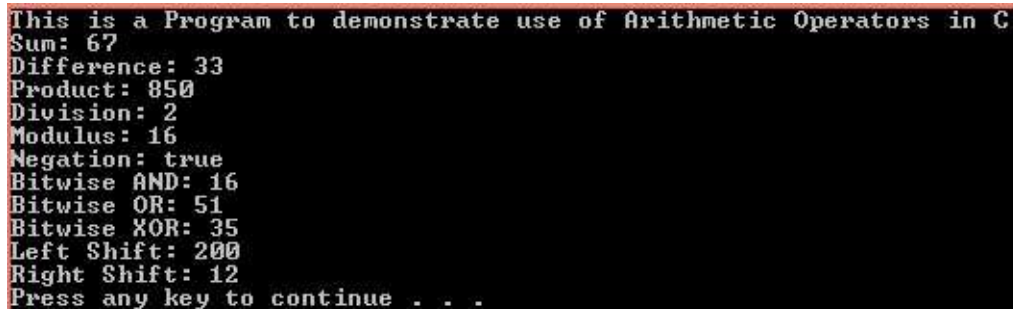
```
printf("Bitwise XOR: %d\n", A^B);
```

```
printf("Left Shift: %d\n", A<<2);
```

```
printf("Right Shift: %d\n", A>>2);
```

```
system("pause");
```

```
}
```



```
This is a Program to demonstrate use of Arithmetic Operators in C
Sum: 67
Difference: 33
Product: 850
Division: 2
Modulus: 16
Negation: true
Bitwise AND: 16
Bitwise OR: 51
Bitwise XOR: 35
Left Shift: 200
Right Shift: 12
Press any key to continue . . .
```

The first four operators are pretty common and straightforward so we'll not discuss them here. Modulus returns us the remainder after dividing one number by the other. $A \% B$ returns the remainder after dividing A by B . For instance, the example in our code divides 50 by 17, 2 is quotient while 16 is the remainder. The remainder is returned and printed. Remember, we can always assign this value by using assignment operators.

Here, we just used the returned value to be printed once.

Negation operator is used to invert a binary number using 2's complement form. If the input is boolean, as is in our program, the result is inverted.

Then we have 5 Bitwise operators, starting with Bitwise AND operation. It performs logical AND on each bit of the numbers provided. For example, in our program, we have 50 and 17. The binary conversions of these two numbers will be 00110010 and 00010001 respectively.

00110010

00010001

00010000

The answer, after converting to decimal is 16.

Similarly, Bitwise OR operation performs OR on every bit of the numbers provided. For example, in our program 50 and 17 are operated and result in 51 as explained as follows.

00110010

00010001

00110001

Similarly, Bitwise XOR operation performs XOR on every bit of the numbers provided.

In XOR, the result is high bit only if there are odd number of high bits in the operation.

For example, in our program 50 and 17 are operated and result in 51 as explained as follows.

00110010

00010001

00100011

The result, when converted into decimal gives us 35.

Shift operators are used to shift binary digits to the left or right. By using left shift operators, the left operands value is moved left by the number of bits specified by the right operand. For example, in our program we used $A \ll 2$. It

means that bits in A must be shifted to left by two digits:

00110010

00110010<<2

11001000

Left Shift operator shifts bits to the left and add 2 zeros to the right and hence the value is changed to 200 as clear from above explanation.

By using right shift operators, the left operands value is moved right by the number of bits specified by the right operand. For example, in our program we used A>>2. It means that bits in A must be shifted to right by two digits:

00110010

00110010>>2

00001100

Right Shift operator shifts bits to the right and add 2 zeros to the left and hence the value is changed to 12 as clear from above explanation.

2.6.2 Assignment Operators Operator Equivalent Operations a = b

Equal to

Assigns the value stored in b to a

a += b

a = a + b

Adds b into a and stores result in a a -= b

a = a - b

Subtracts b from a and stores result in a a *= b

a = a * b

Stores product of a and b in a

$a /= b$

$a = a / b$

Divides a by b and stores result in a $a \%= b$

$a = a \% b$

Takes $a \% b$ and stores the result in a $a \&= b$

$a = a \& b$

Performs bitwise AND on a and b , stores in a $a |= b$

$a = a | b$

Performs bitwise OR on a and b , stores in a $a ^= b$

$a = a \wedge b$

Performs bitwise XOR on a and b , stores in a Here's a program to show the use of these assignment operators with output: `#include "iostream"`

```
void main(void)
```

```
{
```

```
int A = 10, B = 3, simpleAssignment = 0;
```

```
simpleAssignment = A + B;
```

```
printf("This is a Program to demonstrate use of Assignment Operators in C\n");
```

```
printf("Simple assignment: %d\n", simpleAssignment);
```

A += B;

printf("Plus and Equal to: %d\n", A);

A = 10, B = 3;

A -= B;

printf("Minus and Equal to: %d\n", A);

A = 10, B = 3;

A *= B;

printf("Multiply and Equal to: %d\n", A);

A = 10, B = 3;

A /= B;

printf("Modulus and equal to: %d\n", A);

A = 10, B = 3;

A %= B;

```
printf("Bitwise AND and equal to: %d\n", A );
```

```
A = 10, B = 3;
```

```
A &= B;
```

```
printf("Bitwise OR and equal to: %d\n", A );
```

```
A = 10, B = 3;
```

```
A |= B;
```

```
printf("Bitwise XOR and equal to: %d\n", A);
```

```
system("pause");
```

```
This is a Program to demonstrate use of Assignment Operators in C
Simple assignment: 13
Plus and Equal to: 13
Minus and Equal to: 7
Multiply and Equal to: 30
Modulus and equal to: 3
Bitwise AND and equal to: 1
Bitwise OR and equal to: 2
Bitwise XOR and equal to: 11
Press any key to continue . . .
```

```
}
```

2.6.3 Increment Decrement Operators Operator Name

Operations

`++a`

Prefix increment

Adds 1 to the value of a, before any assignment `--a`

Prefix decrement

Subtracts 1 to the value of a, before any assignment `a++`

Postfix increment Adds 1 to the value of a, after any assignment `a--`

Postfix decrement Subtracts 1 to the value of a, after any assignment

The increment operator `++` and the decrement operator `--` are unary operators. `++` and `-`

`-` can be applied to identifiers, but not to constants or ordinary expressions. Moreover, different effects may occur depending on whether the operators occur in prefix or postfix position.

Clearly, Increment and decrement operators can be tricky to work with. In simple situations, one can consider `++` and `--` as operators that provide concise notation for the incrementing and decrementing of a variable. In other situations, careful attention must be paid as to whether prefix or postfix position is desired. They are pretty simple once you know the purpose and use of each and the best way to make you familiar with purpose of each of these operators is the practical. So here is a program which describes usage of these operators:

```
#include "iostream"
```

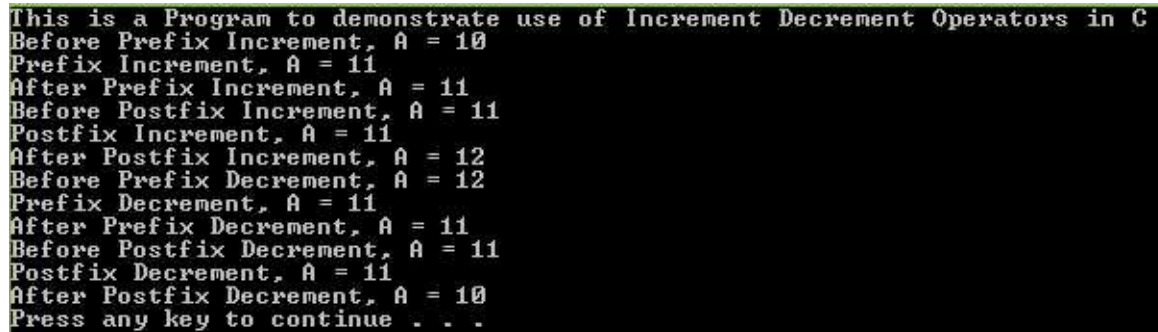
```
void main(void)
```

```
{
```

```
int A = 10;
```

```
printf("This is a Program to demonstrate use of Increment Decrement Operators  
in C\n");
```

```
printf("Before Prefix Increment, A = %d\n", A);
```

A screenshot of a terminal window with a black background and green text. The text shows the output of a C program that demonstrates the use of increment and decrement operators. It starts with a title line, followed by a series of 'Before' and 'After' statements for both prefix and postfix increment and decrement operations on a variable A. The values of A are shown as 10, 11, 12, and 10 at different points. The program ends with a 'Press any key to continue' prompt.

```
This is a Program to demonstrate use of Increment Decrement Operators in C
Before Prefix Increment, A = 10
Prefix Increment, A = 11
After Prefix Increment, A = 11
Before Postfix Increment, A = 11
Postfix Increment, A = 11
After Postfix Increment, A = 12
Before Prefix Decrement, A = 12
Prefix Decrement, A = 11
After Prefix Decrement, A = 11
Before Postfix Decrement, A = 11
Postfix Decrement, A = 11
After Postfix Decrement, A = 10
Press any key to continue . . .
```

```
printf("Prefix Increment, A = %d\n", ++A);
```

```
printf("After Prefix Increment, A = %d\n", A);
```

```
printf("Before Postfix Increment, A = %d\n", A);
```

```
printf("Postfix Increment, A = %d\n", A++);
```

```
printf("After Postfix Increment, A = %d\n", A);
```

```
printf("Before Prefix Decrement, A = %d\n", A);
```

```
printf("Prefix Decrement, A = %d\n", --A);
```

```
printf("After Prefix Decrement, A = %d\n", A);
```

```
printf("Before Postfix Decrement, A = %d\n", A);
```

```
printf("Postfix Decrement, A = %d\n", A--);
```

```
printf("After Postfix Decrement, A = %d\n", A);
```

```
system("pause");
```

```
}
```

So the initial value of A is 10. Prefix increment increments the value to 11.

So, the initial value of A is 10. Prefix increment increments the value to 11 before assignments and thus prints 11 in 3rd line of output. Fourth line just prints and shows that A has been assigned the value 11. Sixth line of output has a postfix increment which means A will be incremented after any assignments in the expression. Thus, the same value, 11 is printed and then A is incremented to 12 which can be seen in the next line.

Prefix decrement decrements the value to 11 before assignments and thus prints 11 in 9th line of output. Next line just prints and shows that A has been assigned the value 11.

12th line of output has a postfix decrement which means A will be decremented after any assignments in the expression. Thus, the same value, 11 is printed and then A is decremented to 10 which can be seen in the next line.

2.6.4 Logical Operators Logical operators are used to get a binary output depending upon certain operations between one or more operands. They are handy when developing decision statements.

```
False
False
True
True
Press any key to continue . . .
```

Operator Name

Operations

!a

Negative

Inverts the value of a

a && b

Logical AND Returns true if both a, b are true a || b

Logical OR

Returns true if a or b or both are true Following is an example illustrating the use of these logical operators: #include "iostream"

```
void main(void)
{
    bool A = true, B = false, C = true;

    printf( !A? "True\n": "False\n");

    printf( (A&&B)? "True\n": "False\n");
    printf( (A&&C)? "True\n": "False\n");
    printf( (A||B)? "True\n": "False\n");
    system("pause");
}
```

2.6.5 Comparison Operators The comparison operators are used to compare two numbers or characters. For binary and string comparisons, we have separate operators and functions so avoid comparing two strings or binary numbers using these operators. It is not recommended.

Operator Name

Operations

a == b

Equals

Returns true if contents of a equal to that of b a != b

a < b

Not equal to

Returns true if a is not equal to that of b $a \neq b$

$a \leq b$

Less than

Returns true if a is less than b

```
False
True
False
True
False
True
Press any key to continue . . .
```

$a \geq b$

Greater than

Returns true if a is greater than b

Less than or equal to

Returns true if a is less than or equal to b Greater than or equal to Returns true if a is greater than or equal to b

Following is an illustration of usage of Comparison operators in C with output.

```
#include "iostream"
```

```
void main(void)
```

```
{
```

```
int a = 10, b = 5;
```

```

printf((a==b)? "True\n": "False\n");
printf((a!=b)? "True\n": "False\n");
printf((a<b)? "True\n": "False\n");
printf((a>b)? "True\n": "False\n");
printf((a<=b)? "True\n": "False\n");
printf((a>=b)? "True\n": "False\n");
system("pause");

}

```

The output is pretty straightforward, so we trust the audience for understanding.

2.6.6 Special Operators

The ternary operator, the one we have consistently used in previous examples is quite useful. Its syntax is as follows:

Condition ? (In case of true) : (In case of false)

So, ternary operator checks a certain condition and depending upon the boolean output, selects one of the alternative cases.

Another useful operator is typecasting operator. It takes an operand and converts it into the desired data type. Its syntax is as follows: (Type) operand;

For example, (int) 3.14 yields 3.

C provides the unary operator sizeof to find the number of bytes needed to store an object. It has the same precedence and associativity as all the other unary operators. The syntax of expression is as follows:

sizeof(object)

2.7 Precedence and Associativity As we discussed earlier as well, we have rules of precedence and associativity that are used to determine how certain expressions are evaluated. Let us briefly discuss how precedence rules are used by a C compiler to compute the unique and correct answer of an expression.

The following is a table representing the precedence of operators in C. The operators at the top have highest precedence while the operators at bottom have the least precedence in an expression. The operators with higher precedence are computed first in an expression.

Category

Operator

Associativity

Postfix

() [] -> . ++ --

Left to right

Unary

+ - ! ~ ++ -- (type)* & sizeof

Right to left

Multiplicative * / %

Left to right

Additive

+ -

Left to right

Shift

<< >>

Left to right

Relational

< <= > >=

Left to right

Equality

== !=

Left to right

Bitwise AND

&

Left to right

Bitwise XOR

^

Left to right

Bitwise OR

|

Left to right

Logical AND

&&

Left to right

Logical OR

~

||

Left to right

Conditional

?:

Right to left

Assignment

= += -= *= /= %= >>= <<= &= ^= |=

Right to left

Comma

,

Left to right

Consider the following code and try to compute the results. We will discuss the output but it is better if you try it yourself before matching our results.

```
#include "iostream"
```

```
void main(void)
```

```
{
```

```
int a = 20, b = 10, c = 5;
```

```
a = b = c += 2;
```

```
printf("a= %d\n",a);
```

```
printf("b= %d\n",b);
```

```
printf("c= %d\n",c);
```

```
a = 20, b = 10, c = 5;
```

```
a = ++b -= c = 2;
```

```
printf("a= %d\n",a);
```

```
printf("b= %d\n",b);
```

```
printf("b= %d\n",c);
```

```
system("pause");
```

```
}
```

The initial values of a, b and c are 20, 10 and 5 respectively. The equivalent expression for the first expression can be: (a = (b = (c = c + 2))); The inner most parentheses assigns the value 5 to c. The other two parenthesis simply assign c to b and b to a. So, all the variables contain 7 after the execution of first

assign c to b and b to a. So, all the variables contain 7 after the execution of this assignment.

Then, we re initialize values of a, b and c with 20, 10 and 5 respectively. The equivalent expression for the second assignment would be:

```
(a=((b=b+1)=b-(c=2)));
```

The inner most parenthesis stores 2 in variable c. The next parenthesis then compute b-c, equivalent to 10-2, and results 8 which is stored in b. Then b is incremented by 1

which gives b the value 9. Then a is assigned the value of b, which is 9.

All the operators on a given line, such as * / % have equal precedence with respect to each other, but have higher precedence than all the operators that occur on the lines below them. We have associativity for all the operators in the rightmost column. For every operator, we will declare its precedence and associativity so that there is a standard to compute a mathematical expression. These rules are to be remembered by every C programmer for efficient programming.

In traditional C, the increment and decrement operators have the same precedence as the unary operators. In ANSI C, however they have the very highest pre-and left-to-right associativity as postfix operators, and they have the same pre-as the other unary operators and right-to-Left associativity as prefix operators.



Chapter 3:Data Types

Data types refer to various categories of data that is to be manipulated. C provides several default data types. We need to discuss limitations on what can be stored in each type.

The keywords signed char, unsigned char, char, signed long int, signed short int, signed int, unsigned int, unsigned short int, unsigned long int, float, double, and long double are the default data types but for programmers' ease, it is allowed to use the short forms.

The next paragraph has those legal shortened forms of data types in C.

Fundamental data types include long, short, signed char, unsigned char, char, int, float, double, long double, unsigned, unsigned short and unsigned long.

These are all keywords which means that they cannot be used as names of variables.

Usually, the keyword signed is not used. The reason is that the signed int is equivalent of int, and because shorter names are easier to type, int is used on most of the occasions.

The type char, however, is special in this regard as we'll see in the later parts of this

chapter

Let us assume that the category type is defined to be anyone of the types given in the preceding paragraph. Using this category, we can provide the syntax of declaring a variable of any type:

type identifier;

For example, this can represent `int x` or `double a` or `char alphabet` *etc.*

It is always easy to understand things when they are grouped (except cats). The default data types can also be grouped according to their functionality. The integral types can be used to hold integer values; the floating and double types are those that be used to hold real values in them. These both categories, when combined form Arithmetic data types.

Integral Data Types Floating Point Data Types char

float

signed char

double

unsigned char

long double

short

int

long

unsigned short

unsigned

unsigned long

3.1 Integral Data Types The data type `int`, the short for integer is the most used integral data type in C. This type, along the other integral types such as `char`, `short`, and `long`, is designed for working the integer values that are representable on a machine. Mathematics define integers as positive and negative whole numbers such as -11, 13, 66 *etc.*

Since machines are designed within bounds, we can have integers up to a certain limit.

Typically, an integer is stored in either 16 bits or in 32 bits. There are other possibilities, but this is what happens in most C systems. On older PCs, an integer is typically stored in 16 bits while newer PCs, workstations and mainframes support 32 bit integers.

To calculate the range we use the power rule. On a 16 bit system, we can have $2^{16} =$

65536 distinct values, half of which would be negative if we are dealing with signed numbers. Similarly on a 32 bit system, we can have $2^{32} = 4294967296$ distinct values, half of which are negative if we are dealing with signed numbers.

In C, the data type integer is considered the "natural" or "usual" type for working with integer. There are other integral types such as `char`, `short`, and `long`. The data type `short`, for example, can be used in situations where we need to use less number of bytes to store our integer, although it is not required to do so in today's advanced world as we do not have any storage issues nowadays. In a similar fashion, the type `long` might be in situations where large integer values are needed. The compiler may provide storage greater than it is required for a natural integer. Typically, `short` is stored in 2 bytes and `long` is stored in 4 bytes. Machines with 4 byte systems have equal sized `short` and `long` data types.

3.2 Floating Point Data Types C provides the three floating types: `float`, `double`, and `long double`. Floating point variables are used to store real values like 3.14, 1.33334 *etc.* A suffix can be added to a floating constant to specify its type. Any floating constant without suffix is of type `double`. The working floating type for C is `double` rather than `float`.

Integers can be used as floating constants, but they must be written with a decimal point. For example, the constants 1.0 and 2.0 are both of type double, whereas as integers, their value will be 1 and 2 respectively.

There is another notation to represent floating point numbers as well which involves exponent. For example, the number 1.234567e-3 calls for shifting the decimal point three to the left to obtain the equivalent constant 0.001234567. This becomes handy when you have to deal with a number of zeros to start right after decimal point. It is easy to remember a floating type number like this.

A floating type number consists of four parts: an integer part, a decimal point, a fractional part, and an exponential part (if any). A floating type number must contain either a decimal point or an exponential part or both. If a decimal point is present, either an integer part or fractional part or both must be present. If no decimal point is present, then there must be an integer part along with an exponential part to indicate the possible position of decimal point.

Some examples of floating constants are 3.14159, 314.15ge-2, 1333e-3 *etc.*

3.3 Using “typedef”

The C language provides the programmer to define a data type using existing data types but with his own customized names. This is achieved through typedef mechanism, which allows the programmer to use an identifier of his choice to represent certain data types. Some examples are

```
typedef int numberOfStudents ;
```

```
typedef double percentage ;
```

```
typedef short int age;
```

In each of these type definitions, the named identifiers can be used later to declare variables or functions in the same way ordinary types can be used.

```
numberOfStudents Strength;
```

The above statement declares the variable Strength to be of type

numberOfStudents, which is synonymous with integer as per our definition.

The advantage of using typedef for customized data types is in abbreviating long declarations. Another advantage is having names that reflect the intended use like numberOfStudents.

3.4 Storage Classes Variables and functions in C have two attributes: type and storage class. You must have gathered around the first attribute so far. Let us now discuss the storage class attribute.

The storage classes are used for certain utilities of variables and functions. The most common storage class is auto. There are three other storage classes defined by these keywords:

auto

extern

register

static

3.4.1 The Storage Class auto By default, C variables declared within function definitions are automatic. Thus, automatic is the most used of the four storage classes. If a compound statement starts with variable declarations, then these variables can be acted on within the scope of the

enclosing compound statement. For difference between compound statements starting with declarations and those which do not start with variable declarations, we use block terminology. A block is a compound statement with declarations.

Declarations within a block are automatically of storage class auto. We don't need to explicitly state auto for these variables. The keyword auto is rarely used. When the program control enters a block, the system allocates memory for all the automatic variables enclosed in the block. The variables are "local" in nature

are automatic variables enclosed in the block. The variables are local in nature within that block.

When the program control exits a block, the system releases the memory that was set aside for the automatic variables. As soon as the blocks are gone, gone are the variables.

That is why these variables are called local.

3.4.2 Storage Class extern The communication between blocks is critical. One way of transmitting information across your scope is using methods to pass parameters to another scope. But, this method is not applicable in certain situations. The storage class extern provides us with another mechanism for communication between different environments or scopes.

Declaration for a variable of extern class is very similar normal variable declaration except that variables are declared outside any block. These variables act as global for all the functions written after their declaration. An example follows for better understanding:

```
int globalVariable = 10;
```

```
void printMe()
```

```
{
```

```
printf("Value of globalVariable = %d\n", globalVariable); }
```

```
void main(void)
```

```
{
```

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

```
globalVariable+=10;
```

```
printMe();
```

```
}
```

Note that we could have written the following statement as traditional C did not allow it but modern ANSI compilers do not complain about this format and also consider a variable outside any scope of extern class even if extern keyword is not used: `extern int globalVariable = 10;`

Such variables cannot be of automatic or register class but they can be static if required as explained in storage class static section. The keyword `extern` is used to make compiler look for this variable anywhere in this file or in any other file for definition and use.

Another important point to remember is that external variables are not deallocated entirely through the program but they may be hidden if a similar identifier is redefined inside a scope. If this happens, the local variable is accessed, not the global variable in that scope.

As we mentioned in the start, Information can be passed into a function two ways: by use of external variables and use of the parameter mechanism. Although there are exceptions, it is preferred to use the second method for this purpose because it leads to better modularity and conciseness of code. Using external variables carelessly may often result in surprises and magic. Global variables cascade their change throughout the program, which clearly can be harmful for your logic. The message or parameter passing technique is free from these hazards since changes are localized. Programs are therefore, easier to write and maintain. All functions have external storage class. This means that we can use the keyword `extern` in function definitions and in function prototypes. For example:

`extern double sum(double a, double b);` **3.4.3 Storage Class register** The register storage class makes variables to work as high speed register. Obviously, it might have hardware based restrictions since there are a limited number of registers in physical space. Because of physical restrictions, variables of this class are automatically converted to auto type whenever compiler finds it impossible to use the variable as a memory register.

Register class is used when fast execution is critical. When speed is a concern, the programmer may choose a few variables that are frequently accessed and declare them to be of storage class register. Registers take lesser number of clock

cycles when they are accessed for use or modification than normal memory location. Since registers store integer values, hence extern variables are integer by default as well. Therefore following two declarations are equivalent: extern int variable;

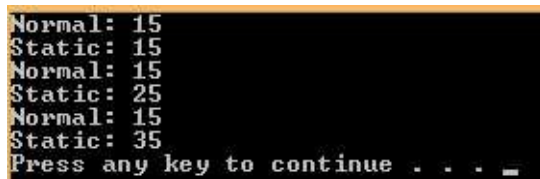
extern variable;

3.4.4 Storage Class static Static means something which keeps its state constant. Static class have two basic uses.

The primary use is similar to what the word static means: for declaring variables which retain their state when blocks are entered and exited frequently.

Remember, normal variables are de-allocated as soon as the block is closed. The second use is its use with external class declarations.

Here is an example:



```
Normal: 15
Static: 15
Normal: 15
Static: 25
Normal: 15
Static: 35
Press any key to continue . . . _
```

```
void demo()
```

```
{
```

```
int normalVariable = 5;
```

```
static int staticVariable = 5;
```

```
normalVariable += 10;
```

```

staticVariable += 10;

printf("Normal: %d\n", normalVariable);
printf("Static: %d\n", staticVariable); }

void main(void)
{

demo();

demo();

demo();

system("pause");

}

```

As you can see in the output of above program, static variable retains its previous value for second and third function call as well. The static declaration of variable is executed only the first time function is called. After that, the compiler ignores it. The next time, static variable has value 15 and 10 is added in to it and 25 is printed. The third function call uses the retained value 25 for static variable and adds 10 to it. On the other hand, normal variable is initialized again and again to 5 and 10 is added. Each of the three function calls creates a normal variable of its own while static variable is only generated the first time and then retained for future references.

Let us now discuss the static external variables. Extern and static together provide better privacy options to the programmer. As you know that global or external variables are accessible only to the functions defined after their

declaration. The static variables on

the other hand are dependent on their blocks that can be molded for use by programmers.

Let us use this facility to provide a variable that is global to a family of functions but, at the same time, is private to the file.

Another use of static is as a storage class to specify function definitions and prototypes.

This use causes the scope of the function to be restricted. Static functions are visible only within the file in which they are defined. Unlike ordinary functions, which can be accessed from other files, a static function is available throughout its own file, but no other.

3.5 Default Initializations In C, both external variables and static variables that are not assigned any values at the time of declaration are automatically initialized with 0. This default initialization is applicable on arrays, strings, pointers, structures and union.

In contrast to this, automatic and register variables are not automatically initialized by the system. This means compiler either generates an error or uses a garbage value if you access an automatic variable or register variable without initializing it.



Chapter 4: Loops & Decisions

Loops refer to programming tools for repetitive execution. This is extremely useful as it allows programmer to write codes that are concise. For example

useful as it allows programmer to write codes that are concise. For example, consider that we want to print a statement a thousand times. Instead of writing the statement a thousand times, we can just program a loop. Such are the benefits of using a loop.

C supports all the basic loops provided by other high level languages. These include for loop, while loop and do while loop. We will discuss each of these loops with example in this chapter.

Computer works on logic designed by humans and is fed to it by programmers using programming languages. Logic is based upon decisions. Decision statements are required for defining these decisions. We will discuss two types of decision statements in this chapter.

4.1 If else statement The most basic and most commonly used decision statement in any programming language is if else statement. This statement can be used for multiple scenarios. It can be used for one or any number of scenarios. Here's a program only with one if statement:

```
void main(void)

{

printf("Program starts\n");


int x=0;


printf("Enter a number and press enter: ");

scanf("%d", &x);


if(x==0)

{

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

```
printf("If executes\n"),
```

```
}
```

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

```
system("pause");
```

```
}
```

This program takes an integral input from user using scanf function. If the user enters 0, then the logical operation in if statement holds true and therefore the body of if statement is executed. If any other number is entered, the body of if statement is not executed.

Let us now see another example which involves if else statement: void main(void)

```
{
```

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

```
int x=0;
```

```
printf("Enter a number and press enter: ");
```

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

```
if(x==0)
```

```
{
```

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

```

    }

else

    {

printf("Else executes\n");

    }

printf("Program ends\n");

system("pause");

    }

```

This program has an alternate condition in the block covered by else statement. Else statement is always used with if statement. In the case, when if statement holds true, the compiler just ignores the else statement. Otherwise, when if statement is false, the body of else block is executed.

Following is an example involving multiple if else statements: void main(void)

```

    {

printf("Program starts\n");

int x=0;

printf("Enter a number and press enter: ");

scanf("%d", &x);

if(x==0)

```

```

printf("1st If executes\n");

else if(x<10)

printf("2nd If executes\n");

else if(x<20)

printf("3rd If executes\n");

else

printf("Else executes\n");

printf("Program ends\n");

system("pause");

```

4.2 Switch Case statement

Switch case is another popular decision making statement in C programming language.

The use is very similar to if else statements but instead of conditions which result in Boolean values, this decision making statement has certain cases for different values.

Let us see an example for better understanding:

```
void main(void)

{

printf("Program starts\n");

char x;

printf("Enter a small alphabet and press enter: ");
x=getchar();

switch(x)

{

case 'a': printf("You entered a..!\n"); break;

case 'b': printf("You entered b..!\n"); break;

case 'c': printf("You entered c..!\n"); break;

case 'd': printf("You entered d..!\n"); break;
```



```
default: printf("You entered something else..!\n");
```

```
}
```

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

```
system("pause");
```

```
}
```

Based on input from user which is passed as an argument in switch statement, the program executes a particular case. If it cannot find any case, the default case is executed.

An important thing to note is the break statement at the end of each case. Break statement is used to leave the current block in any situation. For example, if you are in for loop, you will ignore the code inside for loop after the break statement. Here, break statement is necessary if we do not want multiple cases to be associated with one input.

For example, see the following program: void main(void)

```
{
```

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

```
char x;
```

```
printf("Enter a small alphabet and press enter: ");
```

```
x=getchar();
```

```

switch(x)
{
case 'a': printf("You entered a..\n"); break;

case 'b': printf("You entered b..\n");

case 'c': printf("You entered c..\n"); break;

case 'd': printf("You entered d..\n"); break;

default: printf("You entered something else..\n");

}

printf("Program ends\n");

system("pause");

}

```

In the above program, if user enters character b as input, two cases are executed. Starting from case 'b', compiler executes every line of code unless it reaches a break statement or end of switch case block.

4.3 For Loop As we described earlier, loops are used for repetitive execution of statements. When we know the number of iterations, for loop is the ideal loop to use in a program.

The general format is as follows: for(initialization expression; check condition; increment/decrement){ ... }

```
increment/decrement){...}
```

Here is an example which uses for loop: void main(void)

```
{
```

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

```
char x;
```

```
for(int i=0 ; i<5 ; i++)
```

```
{
```

```
printf("Enter a small alphabet and press enter: ");
```

```
x=getchar();
```

```
switch(x)
```

```
{
```

```
case 'a': printf("You entered a..!\n"); break;
```

```
case 'b': printf("You entered b..!\n"); break;
```

```
case 'c': printf("You entered c..!\n"); break;
```

```
case 'd': printf("You entered d..!\n"); break;
```

```
default: printf("You entered something else..!\n");
```

```
}
```

```
}
```

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

```
system("pause");
```

```
}
```

The above program executes the statements within for loop five times. When compiler reaches for loop, it first of all initializes the variable involved. After that, condition is checked. If condition is true, keep going else stop. So, in the first execution cycle, I is initialized to 0 and condition is true as 0 is less than 5. After first execution cycle, increment or decrement statement at the end of for loop is executed and condition is checked. Now, increment operator gives I the value of 1, which is again less than 5 so loop keeps going. Similarly, loop goes on for three more cycles. After a total of five cycles, value of I will be 5. For 6th possible cycle, value of I is compared with 5 and the answer will be false as 5 is not less than 5. Hence, 6th cycle won't be executed and loop is terminated.

4.4 While Loop This loop uses only a condition to determine how many iterations are to be computed.

There are no (necessary) initialization and incremental change in value of a variable.

There can be initialization and change in a while loop if it is required by program logic.

The general syntax of while loop is as follows: while(condition){....}

Here is an illustration for better understanding of while loop: void main(void)

```
{
```

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

```
char x='a';
```

```
printf("Enter a small alphabet and press enter: ");
```

```
x=getchar();
```

```
while(x!='e')
```

```
{
```

```
switch(x)
```

```
{
```

```
case 'a': printf("You entered a..!\n"); break;
```

```
case 'b': printf("You entered b..!\n"); break;
```

```
case 'c': printf("You entered c..!\n"); break;
```

```
case 'd': printf("You entered d..!\n"); break;
```

```

default: printf("You entered something else..!\n");

        }

printf("Enter e to exit. Else, enter another small alphabet and press enter: ");

x=getchar ();

        }

printf ("Program ends\n");


system ("pause");

        }

```

This program keeps repeating until user enters 'e'. Thus, user is given the control of how many iterations are made by using while loop. This can be done with for loop too, but while is designed for this particular purpose. While loop is highly recommended if the number of iterations required are unknown.

4.5 Do While Loop Do while loop is almost similar to while loop except that do while always execute its first cycle without testing any condition. On the other hand, while loop always checks that the condition given is true, even for the first execution cycle.

Consider following example and compare it with the while loop example: void main(void)

```

        {

printf ("Program starts\n");


char x='a';

```

do

{

printf ("Enter a small alphabet and press enter: ");

x=getchar();

switch(x)

{

case 'a': printf("You entered a..!\n"); break;

case 'b': printf ("You entered b..!\n"); break;

case 'c': printf ("You entered c..!\n"); break;

case 'd': printf ("You entered d..!\n"); break;

default: printf ("You entered something else..!\n");

}

} while(x!='e');

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

```
system ("pause");
```

```
}
```

4.6 Nested Loops Nested loops refer to loops within loops. They are usually useful when dealing with 2-D

arrays. Here is an example in which a 2-D array is initialized using a nested for loop: void main (void)

```
{
```

```
int Two_D[10][10];
```

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

```
{
```

```
for (int j = 0; j<10; j++)
```

```
{
```

```
Two_D[i][j] = i+j;
```

```
}
```

```
}
```

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

```
{
```

```
for(int j = 0; j<10; j++)
```



```
        {  
printf("%d, ", Two_D[i][j]);  
        }  
    }  
  
system ("pause");  
}
```

Similarly, we can use nested loops for a while and do while. You can nest different loops together as well.

Decision statements and loops provide flexibility to the programmer to develop a program according to his understanding. It leads to better development techniques and encourages programmer to think of new solutions instead of just following already established solutions.



Chapter 5: Functions

A function, also known as a method or routine, is simply the programming unit which performs an operation on possible inputs and returns (if required) the outputs.

A function is an assembly of statements to perform a particular task. C is a function oriented language hence every C program is based on one or more functions. There is at least one method in any C program, called main function. Main function is the entry point into any program.

The concept of functions come from the need to divide large and complex tasks into smaller and simpler problems. It is always easy to manage smaller tasks. Functions are entirely up-to the will of programmer, but it is recommended that each function has a specific and unique tasks. Assembly of every individual function solves the basic problem. For example, in a Calculator application, there should be a function for every operation so that coding and debugging is easy.

The use of Functions give us many advantages such as reusability, modularity, readability, simplicity and maintainability.

The basic structuring element of C programs is Function. The standard C library itself, provides many built in functions such as `sqrt`, `abs` *etc.* We will see some of these functions in this chapter.

In C, we have two terminologies regarding Functions. One is Function-Declaration and the other one is Function Definition. Function Declaration means that the function is registered with compiler while Function Definition actually contains the body and functionality of the function.

5.1 Function Definition The function definition is the actual code which is executed whenever that particular function is executed.

A function definition starts with the type of output a function provides. If no value is returned, then the type should be void. The default return type is int. If the type is other than void, then a value of that type is to be returned. If there is a mismatch between the required type and actually returned type, some

conversions may be necessary. After that, we have a function name. The parameters, enclosed within parenthesis, act as a list of inputs that come with a function when it is called. The body includes the functionality core, the actual operations on the input to generate output.

Remember, Function declaration is something different which we will discuss later on. A function definition has the following general form:

```
type functionName (parameter list)

{body}
```

The first line represents the header of a function. Second line here represents the body of a function enclosed in braces by convention. The parameter list refers to a possible multiple inputs. functionName is the name of function while type represents the output of the function.

An example of a function definition is: `int sum (int a, int b)`

```
    {

sum = a + b;

return sum;

    }
```

In the program above, there are two integer inputs whose sum is returned as an output.

The first int represents the return type. Then “sum” is the name of function. A list of parameters passed or inputs (both integer type) is enclosed inside parenthesis. Then, the body is enclosed within braces. The last statement has the keyword “return”, which is used for returning an output in specified format and along with that, program control also returns to the point from where sum(a, b) are called.

The following are two ways to calculate sum of two numbers. One involves use of function and the other does not. Please see the differences to understand the utility of functions:

1. Without Function:

```
void main (void)
```

```
{
```

```
int a = 20, b = 10, c = 5;
```

```
a = b = c += 2;
```

```
printf ("a= %d\n", a);
```

```
printf ("b= %d\n", b);
```

```
printf ("c= %d\n", c);
```

```
a = 20, b = 10, c = 5;
```

```
a = ++b -= c = 2;
```

```
printf ("a= %d\n", a);
```

```
printf("b= %d\n",b);
```

```
printf("b= %d\n",c);
```

```
a = 20, b = 10, c = 5;
```

```
printf("%d",(a=((b=b+1)=b-(c=2)))));
```

```
a = 20, b = 10, c = 5;
```

```
printf("%d",((b=b+1)=b-(c=2)));
```

```
system("pause");
```

```
}
```

2. With Function:

```
int sumAndPrint(int a, int b)
```

```
{
```

```
int answer = a + b;
```

```
printf("Answer = %d\n",answer);
```

```

return answer;

    }

void main(void)

    {

int a=10, b=20, c=30, d=10, e=30, f=30, answer;

answer = sumAndPrint(a , b);


answer = sumAndPrint(c , d);


answer = sumAndPrint(e , f);


system("pause");

    }

```

One of the differences between traditional C and ANSI C is the format of function definition. Traditional C declares the type of input parameters after the parenthesis end but identifiers of parameters are written inside parenthesis. An example is

```
void functionANSI(a, b, c, d, e) int a, b, c; double d, e; { function body }
```

The order in the declaration of parameters does not matter. Since ANSI standard allows most of the features provided by traditional C, hence C compiler nowadays accept both formats for definition of functions.

5.1.1 The return statement When a return statement is executed, the current program finishes its execution and returns to the environment from which it was called. If the return statement is followed by an expression such as ++b; or a; then a value is returned to the calling environment which can be stored and used for further processing. Moreover, this value will be converted, if necessary, to the type of the function as specified in the function definition.

A function can have zero or more return statements. If there are no return statements in a function, it completes its execution at the end of braces and then return to previous environment. An illustration is followed for better understanding: void sumAndPrint(int a, int b)

```
    {  
  
    int answer = a + b;  
  
    printf("Answer = %d\n",answer);  
  
    return;  
  
    printf("Again Answer = %d\n",answer); }
```

Now, the above function prints the sum only once and returns to previous environment without executing the last statement. If there were no return statement in the code, the answer would have been printed twice. Of course, this example was just to illustrate the use of return statement, but you often come across situations in practical problems that you don't need to fully execute a function.

5.2 The Function Prototypes Function prototypes refer to Function Declarations. Functions should be declared before they are used. ANSI C provides for a new function declaration syntax called the function prototype. Function Prototypes are usually written at the top of main program.

A function prototype declares how a function is to be used by specifying the

A function prototype declares how a function is to be used by specifying the function name, input parameter list along with their data types and the return type. Input parameter list may or may not contain identifiers. An example is: `int sum (int, int);`

If someone understands the prototypes, he can use the function in his program. For example, the above function prototype tells us that we need to input two numbers and (most probably) their sum will be returned. Hence, I can write the following statement even though I have not written the sum function myself:
`Answer = sum (a, b);`

Function prototypes allow the compiler to check the code in more detail by specifying all the functions that are used in your program. As we mentioned earlier, the type of parameters may need conversion. Function prototypes make it easier for the compiler to implement these conversions. For example, if we call our sum function like this: `Answer = sum (2.5, 4.3);`

If we have used prototypes, compiler knows that sum function needs two integers rather than double and therefore either converts the input parameters into integers or generates an error of input type mismatch.

5.2.1 Function Prototypes in C++

In C++ function prototypes, the use of void in the parameter type list in both function prototypes and function definitions is optional. This is different from traditional C. For example:

```
void func (void);
```

```
void func ();
```

In C++, both of these prototypes represent same thing: there are no input parameters but in traditional C, the first prototype is illegal and second function prototype refers that there can be any number of parameters for this function. Void is not a keyword in traditional C and hence cannot be used in function declaration and definition.

5.3 Alternative Style for Declarations Some of you may have noticed that we

have not used function prototypes in the program of section 4.1. Instead, we defined the functions above the main function. This is an alternative for prototypes, acceptable by ANSI C compilers. We write the preprocessors, followed by all the function definitions and then finally our main program. This is recommended, if you have only one file to work with. For example, please refer to the examples in section 4.1.

5.4 Function Invocation This section is intended to make you familiar with how a function call works. Every C

program is structured upon one or more programs, one of which is main function. Main function is the entry point in a program so execution starts from main function. When a function is called or invoked, the program control is shifted from main program to the

```
Inside main function, before changeValue is called: Value of a = 10
Inside changeValue function: Value of a = 20
Inside main function, before afterValue is called: Value of a = 10
Press any key to continue . . .
```

called function body. Once, the execution of invoked function is completed, program control is returned to previous environment (main program in this case).

Functions must be called by providing the valid input parameters as defined by Function declaration and definition. Compiler generates an error if a mismatch is found between parameters passed and parameters required. The compiler enforces type compatibility when function prototypes are used.

"Call-by-value" is a term associated with passed parameters in C. It means that when you call a function and pass it parameters, the actual parameters are not passed to the other function. Instead their values are copied and another variable is created in the body of invoked function. This means that each argument is evaluated, and its value is used locally in its own environment. Thus, if a variable is passed to a function, the stored value of that variable in the calling environment will not be changed because the calling environment and called environment have their own copy of arguments.

Following is an example, demonstrating the concept of "call by value": void changeValue(int a)

```

        {

a = a + 10;


printf("Inside changeValue function: Value of a = %d\n", a); }


void main(void)

        {

int a = 10;


printf("Inside main function, before changeValue is called: Value of a = %d\n",
a);

changeValue(a);


printf ("Inside main function, before after changeValue is called: Value of a =
%d\n", a);

system ("pause");

        }

```

So, as you can see, the changeValue function adds 10 to the value of passed parameter but the change is not visible to the variable “a” in main program. The change is made

only to the local variable, initialized by making a copy of variable in main function in changeValue function.

The other common technique used in C is passed by reference which uses

The other common technique used in C is passed by reference which uses pointers. We will discuss pointers and this mechanism in detail in one of the following chapters.

5.5 Environment

What is an ideal way to work in a group? The ideal way to work in a group is to play an assigned role within certain limits and boundaries. Everyone has his privileges and certain limitations. If everyone pokes into each other's work without necessity, it will lead the group to a disaster. The C structure follows these basic principles for the success of every program. As we discussed earlier, C provides locality of variables in functions. At the same time, C gives the programmer to choose the scope of a variable by himself. If a programmer wants something to be shared by multiple functions, C

provides multiple options to do so. For example, use of pointers or global variables.

The basic rule of scoping is that variables are accessible only within the block of their declaration. They are unknown outside the boundaries of that block unless specified otherwise using special techniques. This is an easy rule to be followed but programmers seem to use same identifiers in different environments. For example, see the second example in Examples section at the end of this chapter. The programmer used the identifier 'a' to represent multiple variables in different scope. If there were no rules for Scoping, the compiler could not have differentiated between these identifiers. Let us see another example:

```
int globalVariable = 10;
```

```
void changeValue (int a)
```

```
{
```

```
    a = a + 10;
```

```
    printf ("Inside changeValue function: Value of localVariable = %d\n", a);
```

```
    printf ("Inside changeValue function: Value of globalVariable = %d\n",  
    globalVariable); }
```

```

void main(void)
{

int localVariable=20;


changeValue(localVariable);


printf("Inside changeValue function: Value of localVariable = %d\n",
localVariable);

for(int x=1; x<=5; x++)

{

printf ("%d,", x);

}

printf("\n");


printf ("%d,", x);


system ("pause");

}

```

The above program uses four scopes. The first one is global which means it can be accessed anywhere and changes made to it in one place are visible to everyone else. All the others are local scopes which mean they are visible only to their scope. The identifiers “local Variable”, “a” and “x” are all variables defined in local scope. The first two variables follow the general rules of scope but x violates it. This program fails to compile because variable x, declared in

the "for" loop is accessed outside its scope. The compiler is unable to identify x as an identifier outside for loop. To make this program work, just delete the second last statement in main program.

5.6 Recursion

Recursion is a technique in which a function calls itself in order to achieve a certain goal.

Recursion is a very important as it provides a way for more efficient and concise code for difficult program.

A function is said to be recursive if it calls itself within its body based on a decision statement usually. Here is an example for better illustration: void main(void)

```
    {  
  
printf("This program is never going to end..!\n"); main ();  
  
system ("pause");  
  
    }
```

Following is another example, which keeps calling itself unless a certain condition ($n \leq 1$) is met.

```
int factorial (int n)  
  
    {  
  
if (n <= 1)  
  
return 1;  
  
else
```

```

return (n * factorial(n - 1));

    }

void main(void)

    {

int f=factorial(5);

printf("Factorial = %d\n", f);

system("pause");

    }

```

5.7 Examples

5.7.1 Equalizers

```

void printTable( int );

void printHeading();

void main(void)

    {

printHeading();

int x = 1;

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

```

```

        {

printTable(x);

        }

system("pause");

        }

void printTable(int a)

        {

for(int x=1; x<=10; x++)

        {

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

        }

printf("\n");

        }

void printHeading()

        {

printf("-----Table of Tables-----\n\n"); }

```

5.7.2 Multiple Files In a file header.h:

```

void func(int a);

void funcSquare(int a);

void funcCube(int a);

```

In a file Implementation1.c:

```
void func(int a)
```

```
{
```

```
printf("I am func. I have to print: a = %d\n", a); }
```

In a file Implementation2.c

```
void funcSquare(int a)
```

```
{
```

```
printf("I am funcSquare. I have to print: a = %d\n", a*a); }
```

```
void func(int a)
```

```
{
```

```
printf("I am funcCube. I have to print: a = %d\n", a*a*a); }
```

In a file main.c

```
#include "header.h"
```

```
void main(void)
```

```
{
```

```
printf("This is our main Program. Let's start...\n");
```

```
func(5);
```

```
funcSquare(5);
```

```
funcCube(5);
```



```

printf("This is our main Program. Let's finish...\n");
system("pause");

}

```

5.7.3 Multiple Files

```

void demo()
{

int normalVariable = 5;


static int staticVariable = 5;


normalVariable += 10;


staticVariable += 10;


printf("Normal: %d\n", normalVariable);
printf("Static: %d\n", staticVariable); }

void main(void){

demo();


demo();

```

```
demo();
```

```
system("pause");
```

```
}
```

5.7.4 Pointers

```
void swap(int p, int q)
```

```
{
```

```
int tmp;
```

```
tmp = *p;
```

```
*q = tmp ;
```

```
}
```

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

```
int a = 10, b =20;
```

```
printf("a = %d, b = %d\n", a, b);
```

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

```
printf("a = %d, b = %d\n", a, b);
```

```
system("pause");
```

```
}
```

5.7.5 BubbleSort void bubble(int a[], int n)

```

    {
int i, j;

for (i = 0; i < n; ++i)
    for (j = i + 1; j < n; ++j)
        if (a[j] < a[i])
            swap(&a[i], &a[j]);
    }

```



Chapter 6: Arrays, Pointers and Strings

C provides a data structure that can store multiple values of same type in it by associating each value to a subscript. It is extremely useful in common programming.

For example, you need to store marks of 50 students, instead of declaring 50 separate variables, you can simply use an array with size 50. Each element can be accessed by specifying its index or subscript.

An array is a random access data structure, where each element can be accessed directly and in constant time. A typical illustration of random access is a book - each page of the book can be open independently of others.

Arrays in C consist of contiguous memory locations which means that they are placed in memory together. The lowest member of array has the lowest address and last member of array has the highest memory address assigned to it.

In C, arrays and pointers are closely related concepts. Interestingly, an array name itself acts as a pointer, and pointers, like arrays, can be indexed. “Call by reference”

mechanism is based on pointers. This mechanism allows programmer to change the value of a local variable in one scope from another scope. C uses pointers for specifying parameters in function definitions to accomplish this. Strings are also one-dimensional arrays of characters. They will be discussed in the next chapter.

6.1 1-D Arrays

1-D arrays refer to one dimensional arrays. Such arrays can be considered as a single row or column containing a fixed number of elements. They have only one index. An array is just like a normal variable except for index. Index always start from 0 and end at (arraySize-1).

A general format for array declaration in C is as follows: **Type arrayName [arraySize];** Type refers to the data type of array, followed by identifier and in the end array size enclosed in brackets.

Arrays may be of storage class automatic, static or extern but not register. ANSI C

supports the initialization of all three types of arrays using an initializer but traditional C

only allowed it for static or extern registers.

The general format for initializing 1-D arrays is as follows:

Type identifier [size] = {list of elements separated by commas}

Here is an illustration of using a 1-D array, using initializer: void main(void)

{

```
int table[10] = { 5, 10, 15, 20, 25, 30, 35, 40, 45, 50 };
```

```
//printing elements of array
```

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

```

                                {
printf("5 x %d = %d\n",i+1,table[i]);

                                }

system("pause");

                                }

```

Here is an illustration of using a 1-D array, using a loop for initialization rather than common initializer:

```

void main(void)

                                {

int table[10];

//initialization of array

for(int i = 1; i <= 10 ; i++)

                                {

table[i-1] = i * 5;

                                }

//printing elements of array

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

                                {

printf("5 x %d = %d\n",i+1,table[i]):

```

```
system("pause");
```

```
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50
Press any key to continue . . .
```

}

Output: Same for both programs

6.2 Pointers

A simple variable in a program is stored in a certain number of bytes at a memory location, or address, in the machine. Pointers are basically variables that hold these addresses rather than actual values. These address can be accessed and hence the actual values can also be accessed via “reference” of pointers. Pointers are used in programs to memory and manipulate addresses. If i is a variable, then &i is the address or memory location where value of i is stored. This address can be stored into a pointer variable of same type. The address operator & is unary and has the same precedence and right to left associativity as the other unary operators.

Pointer variables can be declared in programs and then used to take addresses for initialization. The declaration is as follows: `int *p;`

```
int x = 5;
```

```
p = &x;
```

The above lines declare a pointer variable p of integer type which stores address of a memory location specified by another integer variable x. Following is a program which uses pointers to access other variables: void main(void)

```

    {
int *p;

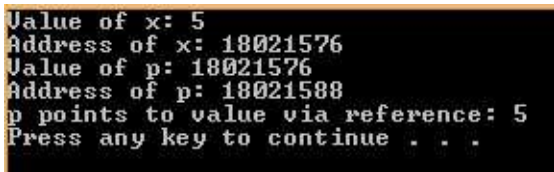
int x = 5;

p = &x;

printf("Value of x: %d\n", x);

printf("Address of x: %d\n", &x);

printf("Value of p: %d\n", p);



printf("Address of p: %d\n", &p);

printf("p points to value via reference: %d\n", *p);

system("pause");

    }

```

```
int *p;
```

```
int x = 5;
```

```
p = &x;
```

```
printf("Value of x: %d\n", x);
```

```
printf("Address of x: %d\n", &x);
```

```
printf("Value of p: %d\n", p);
```

```
Value of x: 5
Address of x: 18021576
Value of p: 18021576
Address of p: 18021588
p points to value via reference: 5
Press any key to continue . . .
```

```
printf("Address of p: %d\n", &p);
```

```
printf("p points to value via reference: %d\n", *p);
```

```
system("pause");
```

$$\}$$

NT	1	0	1	1	1	0	1	0	0	1	0	1	1	0	0	1	1
----	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Normal variable has two numbers associated with it: an address of variable, a value stored on this address. The variable name accesses the value while ampersand with variable name accesses the address. A pointer variable has three numbers associated with it: an address of pointer variable itself, a value stored in it after initialization, a value pointed by value stored in the pointer variable. The first two are accessed using variable name and ampersand variable name, just like normal variables. The third value associated with pointers is accessed using * sign with identifier.

Following is an example which uses pointers to access arrays between two functions to clarify the “call by reference” mechanism as well: void printMe(int *p)

```

{

int arr[10];


for(int i = 1; i <= 10 ; i++)

{

arr[i-1] = *(p+i-1);

}

//printing elements of array

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

{

printf("5 x %d = %d\n",i+1,arr[i]);

}

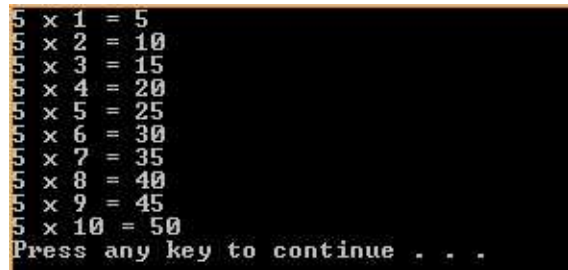
}

void main(void)
```



```
void main(void)
```

```
{
```



```
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50
Press any key to continue . . .
```

```
int table[10];
```

```
//initialization of array
```

```
for(int i = 1; i <= 10 ; i++)
```

```
{
```

```
table[i-1] = i * 5;
```

```
}
```

```
printMe(table);
```

```
system("pause");
```

```
}
```

So, we passed just the starting address of an array as a pointer to another method and we were able to access the array defined in another function. Note that the identifier representing an array, if written without an index is equivalent to the starting address of array (&identifier[0]). *(p+i) is used for indexing. If value of I is 1 and integer is of 4

1 is 1, and integer is 017

bytes, the program accesses the next integer rather than just next byte.

Now, we will see an example in which values of an identifier defined in our main function are changed after calling another function by passing a pointer to it.

```
void changeMe(int *p)
```

```
{
```

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

```
{
```

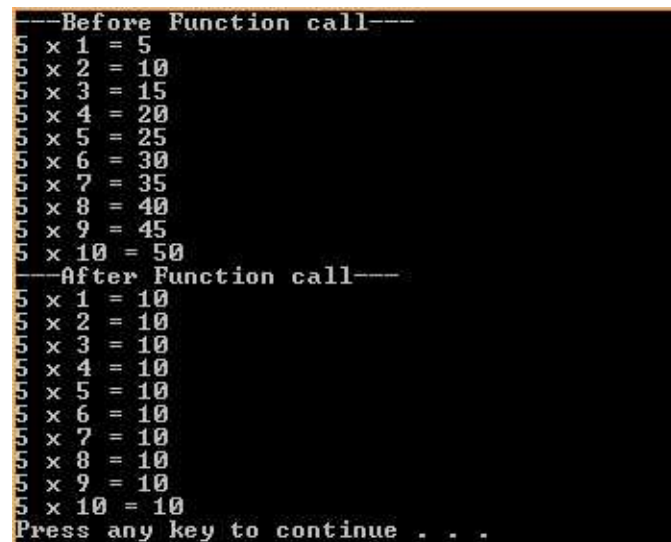
```
p[i]=10;
```

```
}
```

```
}
```

```
void main(void)
```

```
{
```



```
---Before Function call---
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50
---After Function call---
5 x 1 = 10
5 x 2 = 10
5 x 3 = 10
5 x 4 = 10
5 x 5 = 10
5 x 6 = 10
5 x 7 = 10
5 x 8 = 10
5 x 9 = 10
5 x 10 = 10
Press any key to continue . . .
```

```
int table[10];
```

```
//initialization of array
```

```
for(int i = 1; i <= 10 ; i++)
```

```
{
```

```
table[i-1] = i * 5;
```

```
}
```

```
printf("---Before Function call---\n");
```

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

```
{
```

```
printf("5 x %d = %d\n",i+1,table[i]);
```

```
}
```

```
changeMe(table);
```

```
printf("---After Function call---\n");
```

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

```
{
```

```
printf("5 x %d = %d\n",i+1,table[i]);
```

```
}
```

```
system("pause");
```

}

So, we accessed an array defined in our main function inside another function and changed all of its values to 10. The change was visible in our main program. This is what “call by reference” mechanism is used for. The mechanism is clearly possible, only due to pointers.

Although pointers and arrays are almost synonymous in terms of how they are used to access memory, as is clear from above example as well, there are differences, and these differences are subtle and important. A pointer variable can take different addresses as values which means they can be redefined at any point of time. This is not the case with arrays. An array name is an address, or pointer, that is fixed.

6.3 Strings

Strings are one-dimensional arrays of type char. You may be wondering if strings are arrays of char type, why do we need to discuss the separately. The primary difference between Strings and character arrays is that Strings can have variable length whereas char arrays have a fixed size.

As strings can have a variable length, we have a mechanism to identify the end of a string. Every string, by default ends in the null character \0. The size of a string must include the storage needed for the end-of-string character as well. It is programmer’s job to never access a character from a string that over runs the size of string.

In contrast to char arrays which are represented in single quotes, strings are initialized with double quotes. For example, "abc" is a string of size 4, with the last element being the null character \0.

Although, char constants and strings seem closely related but they are not same. For example, "a" and 'a' are not the same. “a” has two characters one of which is ‘a’ and the other one is \0.

A string is considered by the compiler as a pointer. Its value is the base address of the string. Consider the following code: char

*pointer

=

```
"abc";
```

printf(“%s %s\n”, pointer, pointer + 1); The pointer variable is assigned the starting value of string. The first string is printed starting from ‘a’. The second string moves its address by one unit (char is of 1 byte, usually) and hence starts from ‘b’. So, the output for above lines of code will be: abc bc

As strings can be considered as pointers, statements like *(“abc”+2) and “abs”[2] are legal in compilers.

Let us now see an example which uses a char array as well as a string. You can see the convenience in using strings:

```
void main(void)
```

```
{
```

```
char *s = "aString";
```

```
char characters[10] = { 'c', 'h', 'a', 'r', 'a', 'c', 't', 'e', 'r', 's'};
```

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

```
printf("%c", characters[i]);
```

```
printf("\n%s\n", s);
```

```
system("pause");
```

```
}
```

For technical reasons, it is better not to print null characters. Strings do not print null character unless we use pointer to access the last character of the string.

Now, we write a program to count number of words in a string to get better understanding of strings. The program uses a macro, `isspace()` which returns true in a case when the specified character is a space. It returns false otherwise.

```
int numberOfWords(const char *str)
```

```
{
```

```
    int count = 0;
```

```
    while (*str!= '\0')
```

```
    {
```

```
        while (isspace(*str))
```

```
        {
```

```
            ++str;
```

```
        if(*str != '\0')
```

```
        {
```

```
            count++;
```

```
        }
```

```
    }
```

```

        }

    }

return ++count;

}

```

A string or pointer to character array is passed as an argument. We consider spaces as word separator. We increment the word count variable for all the words separated by spaces. We increment it again at the again, so that the last word is also counted.

ANSI C provides the programmer with pre-defined functions to handle strings. This section demonstrates how to use some of these functions. The function prototypes for string-handling functions are given in the standard header file `string.h`. These prototypes use character pointers and strings for certain parameters. Remember, these can be used interchangeably.

Function Name

Description

`char strcat(char s1, const char *s2);` Concatenates second string to the end of first string and returns it.

`int strcmp(const char s1, const char s2);` Compares two strings. It returns a value less than 0 if first string is less than second, a value greater than 0 if first string is greater than second and returns 0 if both are equal.

`char *strcpy(char s1, const char s2);` String s1 is overwritten by String s2 and s1 is returned.

`size_t strlen(const char *s)`

Returns number of characters in string s, excluding the ending character.

These functions can be written by programmers as well. It is recommended to use these as they are more efficient because they use register class variables.

6.4 2-D Arrays

Even though array elements are stored contiguously one after the other, it is usually convenient to think of a two-dimensional array as a table of elements with rows and columns. This helps in better understanding of 2-Dimensional arrays. For example, to declare an array of twenty elements with five rows and 4 columns, we use the following statement:

```
int Two_D[5][4];
```

Nested loops are usually used for dealing with 2-Dimensional arrays.

These elements can be represented as: Two_D[0][0] Two_D[0][1]

Two_D[0][2]

Two_D[0][3]

Two_D[1][0]

Two_D[1][1]

Two_D[1][2]

Two_D[1][3]

Two_D[2][0]

Two_D[2][1]

Two_D[2][2]

Two_D[2][3]

Two_D[3][0]

Two_D[3][1]

Two_D[3][2]

Two_D[3][3]


```
Two_D[3][3]
```

```
Two_D[4][0]
```

```
Two_D[4][1]
```

```
Two_D[4][2]
```

```
Two_D[4][3]
```

6.5 3-D Arrays

Arrays of dimension higher than two work in a similar fashion. Let us describe how three-dimensional arrays work. If we declare `Int Three_D[5][3][2];`

The compiler allocates $5 \times 3 \times 2$ memory address for integer type. The base address is `Three_D[0][0][0]`.

Here is a function that adds the elements of a three dimensional array and returns the total.

```
int sum(int Three_D[5][3][2])
```

```
{
```

```
    int i, j, k, sum = 0;
```

```
    for (i = 0; i < 7; ++i)
```

```
        for (j = 0; j < 9; ++j)
```

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

```
sum += Three_D[i][j][k];
```

```
return sum;
```

```
}
```



Chapter 7: Structures

C is an easily extensible language. It can be extended using header files and standard libraries to use certain functions and macros. It can also be extended by defining data types that are constructed from the fundamental types such as an array which is a group of default data type variables.

The structure type is used to represent heterogeneous data. A structure has components, called members. Structures are designed for programmer's ease so that they can define a data type of their own which may contain more than one members. Each member can be either a default data type or another structure. Because the components of a structure can be of various types, the programmer can design them as per his requirements.

7.1 Declaration The structures allow programmer to integrate different type of variables together.

Arrays are used for storing variables of same type. As a simple example, let us define a structure that describes a student. A student has various attributes of different types such as Name, Course, Section, CGPA and ID. A student can be defined as a structure in C. We can declare the structure type for student using the keyword struct as follows: struct Student

```

{

char *Name;

int Course, ID;

char Section;

double CGPA:

};
```

This declaration creates the derived data type struct student. It is a user defined data type. The structures one defined can be used to define identifiers of this user-defined type. The tag name or identifier along with keyword struct is used to declare variables of type student:

// declare variables of type student

```
Struct Student s1, s2, s3;
```

The above declarations for struct variables allocates storage for the identifiers, which now represent a Student. An alternative way to declare these identifiers is as follows:

```
struct Student
```

```
{
```

```
char *Name;
```

```
int Course, ID;
```

```
char Section;
```

```
double CGPA;
```

```
} s1, s2, s3;
```

Another way of declaration, which is highly recommended as it shortens the declaration of identifiers of the structure, is as follows: typedef struct

```
{
```

```
char *Name;
```

```
int Course, ID;
```

```
char Section;
```

```
double CGPA;
```

```
} Student;
```

7.2 Memory Allocation Although memory is allocated to these variables at the time of declaration but these variables have been assigned nothing. No student type variable has details saved in it.

Here's how each component of a struct type variable is initialized:

```
s1.Name = "R Khan"
```

```
s1.Course = 18;
```

```
s1.ID = 786;
```

```
s1.Section = 'A';
```

```
s1.CGPA = 3.74;
```

7.3 Member Access Operators We can access a member of a struct variable in two ways. The above example uses the member access operator ".". We will now give further examples of its use and introduce the other member access operator "->". Both operators are used for same purpose. The difference lies in the caller. The dot operator is used by a structure variable while arrow is used by a pointer to structure variable.

```
typedef struct
```

```
{
```

```
    char *Name;
```

```
    int Course, ID;
```

```
    char Section;
```

```
    double CGPA;
```

```
}Student;
```

```
void printUsingPointer(Student *s) {

printf("%s %d %d %c %f\n", s -> Name, s -> ID, s -> Course, s -> Section, s ->
CGPA); }

void main(void)

{

Student s1,s2;

s1.Name = "R Khan";

s1.Course = 18;

s1.ID = 786;

s1.Section = 'A';

s1.CGPA = 3.74;

s2.Name = "Ali Khan";

s2.Course = 18;
```

```
s2.ID = 890;
```

```
s2.Section = 'A';
```

```
s2.CGPA = 3.14;
```

```
printf("%s %d %d %c %f\n", s1.Name, s1.ID, s1.Course, s1.Section, s1.CGPA);
```

```
printUsingPointer(&s2);
```

```
system("pause");
```

```
}
```

Following are a few statements and equivalent access methods are listed as well:
struct student s, *p = &s; **s.Section == 'A'; s.Name= "Khan"; s.student_id = 786;**

Expression Equivalent Expression Value s.grade

p -> grade

‘A’

s.Name

p -> Name

“Khan”

(*p).ID

p -> ID

786

*p -> Name + 1

(*(p -> Name)) + 1

D

*(p -> Name + 2)

(p -> Name)[2]

S

7.4 Definition All external and static variables, including struct variables, are initialized by the programmer are automatically initialized by the system to zero, unless specified otherwise. In traditional C, only external and static variables can be initialized. ANSI C

allows automatic variables, including structures, to be initialized as well. The syntax for initializing structures is quite similar to that for initializing arrays as structures also take values in braces.

A structure variable in a declaration can be initialized by following it with an equal sign and a list of constants contained within braces. If not enough values are used to assign all the members of the structure, the remaining members are assigned with value 0.

Some examples are:

Student s1 = {“Mashal”, 18, 123, ‘A’, 3.79}; Pair a[3][3] = { { {1.0, -10}, {20.0, 0.2}, {3.5, 0.3} }, { {4.7, -0.4}, {2.0, 0.5}, {6.0, 1.6} } }; **7.5 Abstract Data**

Types The term abstract data type (ADT) refers to a data structure together with its operations, without specifying an implementation. Suppose we wanted a new

integer type, one that could hold arbitrarily larger values than the default value. The new integer type together with its arithmetic operations will be an ADT. The idea of ADT comes from variety of requirements that keeps growing with innovative applications. Native types such as char, int, and double are already implemented by the C compiler and cannot be altered.

So, we need a mechanism to define our own Data types with certain operations.

7.5.1 Stacks An ADT stack is a container of objects that follows the last-in first-out (LIFO) principle.

It means that you can access only the latest element you have entered. A stack is a limited access data structure - elements can be added and removed from the stack only at the top. For accessing other elements, you need to remove the objects on top of them.

Only two primary operations are necessary for stack implementation: push and pop.

Push refers to adding an element to the top of stack while pop removes the item at the top of the stack. A helpful analogy is to think of a stack of dumbbells on a steel rod; you can remove only the top dumbbell, also you can add a new dumbbell on the top.

Now we develop and implement the ADT stack, one of the most useful standard data structures. The typical operations that can be implemented for a stack are push, pop, top, empty, full, and reset. The push and pop operations are discussed earlier. The empty operator tests if the stack is empty or not and results in a Boolean value. The full operator tests if the stack is full, which also returns a Boolean value. The reset operator removes all the elements of stack and initializes it.

We will use a fixed-length char array to store the contents of the stack. The top of the stack will be an integer-valued member named top. The various stack operations discussed in the previous paragraph will be implemented as functions, each of whose parameter lists includes a parameter of type pointer to stack. Using a pointer results in better efficiency by avoiding to copy large stack

values.

Here is a complete C program, which implements various functions of stack:

```
#define MAX_SIZE 50
```

```
int stack[MAX_SIZE];
```

```
void push();
```

```
int pop();
```

```
int is_empty();
```

```
int peek();
```

```
int TOP = -1;
```

```
int main()
```

```
{
```

```
int element, choice;
```

```
while(1)
```

```
{
```

```
printf("Stack Operations.\n"); printf("1. Insert into stack (Push operation).\n");  
printf("2. Delete from stack (Pop operation).\n"); printf("3. Print top element of  
stack.\n");
```

```
printf("4. Check if stack is empty.\n"); printf("5. Exit.\n");
```

```
printf("Enter your choice.\n"); scanf("%d",&choice);
```

```
switch (choice)
```

```
{
```

case 1:

```
if (TOP == MAX_SIZE - 1)
```

```
printf("Error: Overflow\n\n"); else {
```

```
printf("Enter the value to insert.\n"); scanf("%d", &element);
```

```
push(element);
```

```
}
```

```
break;
```

case 2:

```
if (TOP == -1)
```

```
printf("Error: Underflow.\n\n"); else {
```

```
element = pop();
```

```
printf("Element removed from stack is %d.\n", element); }
```

```
break;
```

case 3:

```
if (!is_empty()) {
```

```
element = peek();
```

```
printf("Element at the top of stack is %d\n\n", element); }
```

```
else
```

```
printf("Stack is empty.\n\n"); break;
```

case 4:

if (is_empty())

printf("Stack is empty.\n\n"); else

printf("Stack is not empty.\n\n"); break;

case 5:

exit(0);

}

}

}

void push(int value)

{

TOP++;

stack[TOP] = value;

}

int pop()

{

int element;

if (TOP == -1)

return TOP;

element = stack[TOP];

```

TOP--;

return element;

        }

int is_empty()

        {

if (TOP == - 1)

return 1;

else

return 0;

        }

int peek()

        {

return stack[TOP];

        }

```

7.5.2 Queues A queue is a container of objects (a linear collection) that follows the first-in first-out (FIFO) principle. An example of queues is any real life queue like in a Bank, or ATM or Airport. A new element or object is added to the back of the queue, while removal (or serving) happens from the front. In the queue only two operations are allowed. These are enqueue and dequeue functions. First one means to insert an item into the back of the queue and dequeue refers to removing an item from the front.

The difference between stacks and queues is in removing. In a stack we remove the last item that was added whereas in a queue, we remove the item that has been in the queue more than anyone else.

Here is a complete program illustrating the use of Queues: #define MAX_SIZE

50

```
int queue_array[MAX_SIZE];
```

```
int rear = - 1;
```

```
int front = - 1;
```

```
void main(void)
```

```
{
```

```
int choice;
```

```
while (1)
```

```
{
```

```
printf("1.Insert element to queue \n"); printf("2.Delete element from queue \n");  
printf("3.Display all elements of queue \n"); printf("4.Quit \n");
```

```
printf("Enter your choice : "); scanf("%d", &choice);
```

```
switch (choice)
```

```
{
```

```
case 1:
```

```
insert();
```

```
break;
```

```
case 2:
```

```
delete();
```

```
break;
```

```
case 3:
```

```
display();
```

```
break;
```

```
case 4:
```

```
exit(1);
```

```
default:
```

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

```
}
```

```
}
```

```
}
```

```
void insert()
```

```
{
```

```
int add_item;
```

```
if (rear == MAX - 1)
```

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

```
else
```

```
{
```

```
if (front == - 1)
```

```
front = 0;
```

```
printf("Inset the element in queue : "); scanf("%d", &add_item);
```

```
rear = rear + 1;
```

```
queue_array[rear] = add_item; }
```

```
}
```

```
void delete()
```

```
{
```

```
if (front == - 1 || front > rear) {
```

```
printf("Queue Underflow \n"); return ;
```

```
}
```

```
else
```

```
{
```

```
printf("Element deleted from queue is : %d\n", queue_array[front]); front = front  
+ 1;
```

```
}
```

```
}
```

```
void display()
```

```
{
```

```
int i;
```

```
if (front == - 1)
```

```
printf("Queue is empty \n");
```

```
else
```

```
{
```



```

printf("Queue is : \n");

for (i = front; i <= rear; i++) printf("%d ", queue_array[i]); printf("\n");

    }

}

```

7.5.3 Linked Lists A linked list is a sequential access data structure, where each element can be accessed only in particular order. A typical illustration of sequential access finding a particular file from an unorganized pile of files. It is like a clothes line on which the data structures hang sequentially. A head pointer addresses the first element of the list, and each element points at a successor element, with the last element having a NULL value.

Here's a code that implements a linked list. Linked list consists of adjacent nodes with each having the address of next node. Starting from head, we can traverse through the whole list by using the pointer next in each node. We can take another pointer in node which can hold address for previous node as well.

```

struct Node

{

int val;

struct Node * next;

};

typedef struct Node element;

void main()

{

element curr, head;

int i;

```

```
head = NULL;

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

    {

curr = (element *)malloc(sizeof(element)); curr->val = i;

curr->next = head;

head = curr;

    }

curr = head;

while(curr) {

printf("%d\n", curr->val); curr = curr->next ;

    }

}
```



Chapter 8: Binary Trees

A tree is a finite set of elements, often known as nodes. Every tree has a unique starting node, called the root node, whereas the remaining nodes are a disjoint collection of subtrees of this root node. Each node keeps track of its parents or/and children so that traversal can be made possible. A node with no children is called a leaf node or a terminal node.

A binary tree is a tree whose nodes can have a maximum of two children. A binary tree is a data structure which comprises of two possible sub-elements of every node, called left child and right child. If a node does not have a left or right child, they are assigned NULL values. Each link must point at a new object not pointed at or be NULL.

The general form a node in a binary tree is as follows: struct node

{

Char value;

Node *left*, right ;

};

The binary tree is a useful data structure for rapidly storing sorted data and rapidly retrieving stored data. This results in an immense increase in efficiency of searching and sorting.

8.1 Composition A binary tree is composed of parent nodes, or leaves, each of which stores data and also links to up to at most two child nodes. It is the relationship between the leaves linked to and the linking leaf, also known as the parent node, which makes the binary tree such an efficient data structure.

Generally, the left node has a lesser value than parent while right child has a greater key value than its parents. Due to these two simple rules, the efficiency of trees is enormous.

As a result, the leaves on the farthest left of the tree have the lowest values and we already know that so if we want to get the lowest value, we only need to traverse to the lowermost leaf node to the left instead of going through every element of the tree.

Similarly, the leaves on the right of the tree have the greatest values and therefore if we need to find the highest number, we just need to traverse to one side of the tree and we

can reach the destination much quicker than a program which used sorting and searching algorithms to accomplish this task.

More importantly, as each leaf connects to two other leaves, it is the beginning of a new, smaller, binary tree. Due to this reason, it is possible to easily access and insert any type data in a binary tree at any point of time, using search and insert functions recursively called on successive leaves and maintaining the order of tree as well. The newly inserted node always goes to the right place.

8.2 Insert Function The following insert function inserts elements in a tree. If tree is not created, the element creates it as well. It uses pointers to pointers in order to handle the case of a tree that might not exist. By taking a pointer to a pointer, it is possible to allocate memory if the root pointer is NULL.

```
void insert(int key, struct node **terminal) {  
  
    if( *terminal == 0 )  
  
        {  
  
            *terminal = (struct node*) malloc( sizeof( struct node ) ); (*terminal)-  
>key_value = key; (*terminal)->left = 0;  
  
            (*terminal)->right = 0;  
  
        }  
  
    else if(key < (*terminal)->key_value) {  
  
        insert( key, &(*terminal)->left ); }  
}
```

```

else if(key > (*terminal)->key_value) {
insert( key, &(*terminal)->right ); }

}

```

The insert function follows the two basic rules of a binary tree. For any current node, move left if you have a lower value to enter and move right for a greater value, until it reaches a NULL node, which it allocates memory for and initializes with the new key value. The pointers to the second last node are updated and pointers for newly initialized node are set to NULL. Once, the element has been added, the insert function will stop calling itself and return program control to normal environment for execution.

8.3 Search Function Here is a search function for a binary tree. It takes a key value and a starting node and searches the key value in the tree recursively. If leaf node is found, it is returned otherwise 0 is returned.

```

struct node search(int key, struct node terminal) {

if( terminal != 0 )

{

if(key==terminal->key_value) {

return terminal;

}

else if(key<terminal->key_value) {

return search(key, terminal->left); }

else

{

return search(key, terminal->right); }

```

}

else return 0;

}

The search function shown above recursively moves down the tree until it either reaches a node with a key value equal to the value for which the function is searching. The value being searched for may not be stored in the binary tree and hence we maintain a condition when search reaches a node with NULL value, it is concluded that the key value leaf is not there in the binary tree. It returns a pointer to the node to the previous instance of the function which called it.



Conclusion

The book is a simple, complete and quick reference for programming in C. The book presents the general purpose language, step by step. It is written following a programmer's approach and is therefore ideal for using as a quick guide and reference for people who aim to code in C.

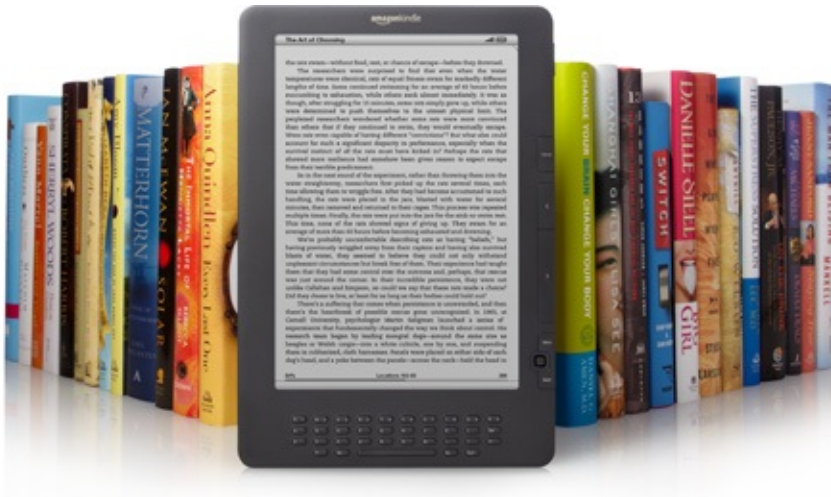
The book covers all the basic concepts of programming in detail like Operators, Functions, Pointers, Arrays, Strings, and decision making statements. The book has the flexibility to help the readers if they are interested in selective study. The chapters are written with minimum possible inter-dependency.

At the end of the book, the reader must be an efficient programmer in C and will be familiar with following concepts of C: • An overview of the language

- Lexical elements of C
- Loops
- Decision Making statements
- Arrays ()
- Strings
- Pointers
- Functions
- Abstract data types
- Stacks
- Queues
- Linked lists
- Binary Tree

If you've found the book helpful, please leave a review for it now. Also check out my other books on Kindle.

R.J. Khan



Check Out My Other Books:

C Programming Language Quick Start Guide: Simplified Guide For Beginners

SCRUM: The Fast and Simplified Guide Scrum – Agile Project Management (coming soon)