CHAPTER 1

C Language Overview

This chapter describes the basic details about C programming language, how it
emerged, what are strengths of C and why we should use C.

T he C programming language is a general-purpose, high-level language that was

originally developed by Dennis M. Ritchie to develop the UNIX operating system at Bell Labs. C was originally first implemented on the DEC PDP-11 computer in 1972.
In 1978, Brian Kernighan and Dennis Ritchie produced the first publicly available description of C, now known as the K&R standard.
The UNIX operating system, the C compiler, and essentially all UNIX applications programs have been written in C. The C has now become a widely used professional language for various reasons.
• Easy to learn

• Structured language

• It produces efficient programs.

• It can handle low-level activities.

• It can be compiled on a variety of computer platforms.

Facts about C

• C was invented to write an operating system called UNIX.

• C is a successor of B language, which was introduced around 1970.

• The language was formalized in 1988 by the American National Standard Institute.
(ANSI).

• The UNIX OS was totally written in C by 1973.
• Today, C is the most widely used and popular System Programming Language.

• Most of the state-of-the-art softwares have been implemented using C.

• Today's most][popular Linux OS and RBDMS MySQL have been written in C.

Why to use C?

C was initially used for system development work, in particular the programs that make up the operating system. C was adopted as a system development language because it produces code that runs nearly as fast as code written in assembly language. Some examples of the use of C might be:
• Operating Systems

- Language Compilers

- Assemblers

- Text Editors

- Print Spoolers

- Network Drivers

- Modern Programs

- Databases

- Language Interpreters

- Utilities

C Programs

A C program can vary from 3 lines to millions of lines and it should be written into one or more text files with extension ".c"; for example, hello.c. You can use "vi", "vim" or any other text editor to write your C program into a file.
This tutorial assumes that you know how to edit a text file and how to write source code using any programming language.

C Environment Setup

This section describes how to set up your system environment before you start doing
your programming using C language.

Before you start doing programming using C programming language, you need the following two softwares available on your computer, (a) Text Editor and (b) The C Compiler.

Text Editor

This will be used to type your program. Examples of few editors include Windows Notepad, OS Edit command, Brief, Epsilon, EMACS, and vim or vi.
Name and version of text editor can vary on different operating systems. For example, Notepad will be used on Windows, and vim or vi can be used on windows as well as Linux or UNIX.
The files you create with your editor are called source files and contain program source code. The source files for C programs are typically named with the extension ".c".
Before starting your programming, make sure you have one text editor in place and you have enough experience to write a computer program, save it in a f ile, compile it and finally execute it.

The C Compiler

The source code written in source file is the human readable source for your program. It needs to be "compiled", to turn into machine language so that your CPU can actually execute the program as per instructions given.
This C programming language compiler will be used to compile your source code into final

executable program. I assume you have basic knowledge about a programming language compiler.

Most frequently used and free available compiler is GNU C/C++ compiler, otherwise you can have compilers either from HP or Solaris if you have respective Operating Systems.

Following section guides you on how to install GNU C/C++ compiler on various OS. I'm mentioning C/C++ together because GNU gcc compiler works for both C and C++ programming languages.

Installation on UNIX/Linux

If you are using Linux or UNIX, then check whether GCC is installed on your system by entering the following command from the command line:

`$ gcc -v`

If you have GNU compiler installed on your machine, then it should print a message something as follows:

```
Using built-in specs.
Target: i386-redhat-linux
Configured with: ../configure --prefix=/usr .......
Thread model: posix
gcc version 4.1.2 20080704 (Red Hat 4.1.2-46)
```

If GCC is not installed, then you will have to install it yourself using the detailed instructions available athttp://gcc.gnu.org/install/

This tutorial has been written based on Linux and all the given examples have been compiled on Cent OS flavor of Linux system.

Installation on Mac OS

If you use Mac OS X, the easiest way to obtain GCC is to download the Xcode development environment from Apple's web site and follow the simple installation instructions. Once y ou have Xcode setup, you will be able to use GNU compiler for C/C++.

Xcode is currently available at developer.apple.com/technologies/tools/.

Installation on Windows

To install GCC at Windows you need to install MinGW. To install MinGW, go to the MinGW homepage, www.mingw.org, and follow the link to the MinGW download page. Download the latest version of the MinGW installation program, which should be named MinGW-<version>.exe.

While installing MinWG, at a minimum, you must install gcc-core, gcc-g++, binutils, and the MinGW runtime, but you may wish to install more.

Add the bin subdirectory of your MinGW installation to your PATH environment variable, so that you can specify these tools on the command line by their simple names.

When the installation is complete, you will be able to run gcc, g++, ar, ranlib, dlltool, and several other GNU tools from the Windows command line.

C Program Structure

Let's look into Hello World example using C Programming Language.

B efore we study basic building blocks of the C programming language, let us look a

bare minimum C program structure so that we can take it as a reference in upcoming chapters.

C Hello World Example

A C program basically consists of the following parts:

• Preprocessor Commands

• Functions

• Variables

• Statements & Expressions

• Comments

Let us look at a simple code that would print the words "**Hello World**":

```c
#include <stdio.h>
int main()
{
/* my first program in C */
printf("Hello, World! \n");
return 0;
}
```

Let us look various parts of the above program:

1. The first line of the program #include **<stdio.h>** is a preprocessor command, which tells a C compiler to include **stdio.h** file before going to actual compilation.

2. The next line **int main()** is the main function where program execution begins.

3. The next line **/*...*/** will be ignored by the compiler and it has been put to add additional comments in the program. So such lines are called comments in the program.

4. The next line **printf(...)** is another function available in C which causes the message "Hello, World!" to be displayed on the screen.

5. The next line return 0; terminates **main()**function and returns the value 0.

Compile & Execute C Program

Let's look at how to save the source code in a file, and how to compile and run it. Following are the simple steps:

1. Open a text editor and add the above-mentioned code.

2. Save the file as **hello.c**

3. Open a command prompt and go to the directory where you saved the file.

4. Type **gcc hello.c** and press enter to compile your code.

5. If there are no errors in your code, the command prompt will take you to the next line and would generate **a.out** executable file.

6. Now, type **a.out** to execute your program.

7. You will be able to see **"Hello World"** printed on the screen

```
$ gcc hello.c
$ ./a.out
Hello, World!
```

Make sure that **gcc** compiler is in your path and that you are running it in the directory containing source file hello.c.

C Basic Syntax

This chapter will give details about all the basic syntax about C programming
language including tokens, keywords, identifiers, etc.

Y

ou have seen a basic structure of C program, so it will be easy to understand other

basic building blocks of the C programming language.

Tokens in C

A C program consists of various tokens and a token is either a keyword, an identifier, a constant, a string literal, or a symbol. For example, the following C statement consists of five tokens:

```
printf("Hello, World! \n");
```

The individual tokens are:

```
printf
(
"Hello, World! \n"
)
;
```

Semicolons ;

In C program, the semicolon is a statement terminator. That is, each individual statement must be ended with a semicolon. It indicates the end of one logical entity.
For example, following are two different statements:

```
printf("Hello, World! \n");
return 0;
```

Page 7

Comments

Comments are like helping text in your C program and they are ignored by the compiler. They start with /* and terminates with the characters */ as shown below:

```
/* my first program in C */
```

You cannot have comments within comments and they do not occur within a string or character literals.

Identifiers

A C identifier is a name used to identify a variable, function, or any other user -defined item. An identifier starts with a letter A to Z or a to z or an underscore _ followed by zero or more letters, underscores, and digits (0 to 9).
C does not allow punctuation characters such as @, $, and % within identifiers. C is a case sensitive programming language. Thus, Manpower and manpower are two different identifiers in C. Here are some examples of acceptable identifiers:

```
mohd zara abc move_name a_123
myname50 _temp j a23b9 retVal
```

Keywords

The following list shows the reserved words in C. These reserved words may not be used as constant or variable or any other identifier names.
auto else Long switch
break
enum
register
typedef
case extern return union
char
float
short
unsigned
const for signed void
continue goto sizeof volatile

default if static while
do
int
struct
_packed
double

Whitespace in C

A line containing only whitespace, possibly with a comment, is known as a blank line, and a C compiler totally ignores it.

Whitespace is the term used in C to describe blanks, tabs, newline characters and comments. Whitespace separates one part of a statement from another and enables the compiler to identify where one element in a statement, such as int, ends and the next element begins. Therefore, in the following statement:

```c
int age;
```

There must be at least one whitespace character (usually a space) between int and age for the compiler to be able to distinguish them. On the other hand, in the following statement:

```c
fruit = apples + oranges; // get the total fruit
```

No whitespace characters are necessary between fruit and =, or between = and apples, although you are free to include some if you wish for readability purpose

CHAPTER 2

C Data Types

In the C programming language, data types refer to an extensive system used for

declaring variables or functions of different types. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.
The types in C can be classified as follows:
S.N.
Types and Description
Basic Types:
1 They are arithmetic types and consists of the two types: (a) integer types and (b) floatingpoint types.
Enumerated types:
2 They are again arithmetic types and they are used to define variables that can only be assigned certain discrete integer values throughout the program.
3 **The type void:**
The type specifier *void* indicates that no value is available.
Derived types:
4 They include (a) Pointer types, (b) Array types, (c) Structure types, (d) Union types and (e) Function types.
The array types and structure types are referred to collectively as the aggregate types. The type of a function specifies the type of the function's return value. We will see basic types in the following section, whereas, other types will be covered in the upcoming chapters.

Integer Types

Following table gives you details about standard integer types with its storage sizes and value ranges:
Type
Storage size
Value range
Char
1 byte
-128 to 127 or 0 to 255
unsigned char
1 byte
0 to 255
signed char 1 byte -128 to 127
Int
2 or 4 bytes
-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int
2 or 4 bytes
0 to 65,535 or 0 to 4,294,967,295
Short
2 bytes
-32,768 to 32,767
unsigned short

2 bytes

0 to 65,535

Long

4 bytes

-2,147,483,648 to 2,147,483,647

unsigned long

4 bytes

0 to 4,294,967,295

To get the exact size of a type or a v a ria bl e on a par ticula r pla t fo rm, y ou ca n use the **sizeof** operator. The expressions **sizeof(type)** yields the storage size of the object or type in bytes. Following is an example to get the size of **int** type on any machine:

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

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

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

Floating-Point Types

Following table gives you details about standard floating-point types with storage sizes and value ranges and their precision:

Type Storage size Value range Precision

float 4 byte 1.2E-38 to 3.4E+38 6 decimal places

double 8 byte 2.3E-308 to 1.7E+308 15 decimal places

long double 10 byte 3.4E-4932 to 1.1E+4932 19 decimal places

The header file **float.h** defines macros that allow you to use these values and other details about the binary representation of real numbers in your programs. Following example will print storage space taken by a float type and its range values:

```c
#include <stdio.h>
#include <float.h>
int main()
```

```c
{
printf("Storage size for float : %d \n", sizeof(float));
printf("Minimum float positive value: %E\n", FLT_MIN );
printf("Maximum float positive value: %E\n", FLT_MAX );
printf("Precision value: %d\n", FLT_DIG );
return 0;
}
```

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

```
Storage size for float : 4
Minimum float positive value: 1.175494E-38
Maximum float positive value: 3.402823E+38
Precision value: 6
```

The void Type

The void type specifies that no value is available. It is used in three kinds of situations:

S.N. Types and Description

Function returns as void

There are various functi ons in C which do not return val ue or you can say they return void.

1 A function w ith no return value has the return type as void. For example, **void exit (int status);**

Function arguments as void

2 There are various functions in C which do not accept any parameter. A function with no parameter can accept as a void. For example, **int rand(void);**

Pointers to void

A pointer of ty pe void * represents the address of an object, but not its ty pe. For example,

3 a memory allocation function **void *malloc(size_t size);** returns a pointer to void which can be casted to

any data type.

The void type may not be understood to you at this point, so let us proceed and we will cover these concepts in the upcoming chapters.

C Variables

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in C has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C is case-sensitive. Based on the basic types explained in previous chapter, there will be the following basic variable types:

Type
Description
Char
Typically a single octet(one byte). This is an integer type.
Int
The most natural size of integer for the machine.
Float
A single-precision floating point value.
Double
A double-precision floating point value.
Void
Represents the absence of type.

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

Variable Definition in C:

A variable definition means to tell the compiler where and how much to create the storage for the variable. A variable definition specifies a data type and contains a list of one or more variables of that type as follows:

```
type variable_list;
```

Here, **type** must be a valid C data type including char, w_char, int, float, double, bool or any user - defined object, etc., and **variable_list** may consist of one or more identifier names separated by commas. Some valid declarations are shown here:

```
int i, j, k;
char c, ch;
float f, salary;
double d;
```

The line **int i, j, k;** both declares and defines the variables i, j and k; which instructs the compiler to create variables named i, j and k of type int.

Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows:

```
type variable_name = value;
```

Some examples are:

```
extern int d = 3, f = 5; // declaration of d and f.
int d = 3, f = 5; // definition and initializing d and f.
byte z = 22; // definition and initializes z.
char x = 'x'; // the variable x has the value 'x'.
```

For definition without an initializer: variables with static storage duration are implicitly initialized with NULL (all bytes have the value 0); the initial value of all other variables is undefined.

Variable Declaration in C:

A variable declaration provides assurance to the compiler that there is one variable existing with the given type and name so that compiler proceed for further compilation without needing complete detail about the variable. A variable declaration has its meaning at the time of compilation only, compiler needs actual variable declaration at the time of linking of the program. A variable declaration is useful when you are using multiple files and you define your variable in one of the files, which will be available at the time of linking of the program. You will use extern keyword to declare a variable at any place. Though you can declare a variable multiple times in your C program but it can be defined only once in a file, a function or a block of code.

Example

Try the following example, where variables have been declared at the top, but they have been defined and initialized inside the main function:

```c
#include <stdio.h>
// Variable definition:
extern int a, b;
extern int c;
extern float f;
int main ()
{
// Variable
definition: int a, b;
int c;
float f;
// actual initialization
a =10;
```
Page 14
```c
b =20;
c = a + b;
printf("value of c : %d \n", c);
f = 70.0/3.0;
printf("value of f : %f \n", f);
return 0;
}
```
When the above code is compiled and executed, it produces the following result:
```
value of c : 30
value of f : 23.333334
```
Same concept applies on function declaration where you provide a function name at the time of its declaration and its actual definition can be given anywhere else. For example:
```c
// function declaration
int func();
int main()
{
// function call
int i = func();
}
// function definition
int func()
{
return 0;
}
```

Lvalues and Rvalues in C

There are two kinds of expressions in C:

1. **lvalue:** An expression that is an **lvalue** may appear as either the left-hand or right-hand side of an assignment.

2. **rvalue:** An expression that is an **rvalue** may appear on the right- but not left-hand side of an assignment.

Variables are lvalues and so may appear on the left-hand side of an assignment. Numeric literals are rvalues and so may not be assigned and cannot appear on the left-hand side.

Following is a valid statement:
```
int g = 20;
```
But following is not a valid statement and would generate compile -time error:

C Constants and Literals

The constants refer to fixed values that the program may not alter during its

execution. These fixed values are also called **literals**.

Constants can be of any of the basic data types like an **integer constant**, **a floating constant, a character constant,** or **a string literal.** There are also enumeration constants as well.

The constants are treated just like regular variables except that their values cannot be modified after their definition.

Integer literals

An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal.

An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.

Here are some examples of integer literals:
```
212 /* Legal */
215u /* Legal */
0xFeeL /* Legal */
078 /* Illegal: 8 is not an octal digit */
032UU /* Illegal: cannot repeat a suffix */
```
Following are other examples of various types of Integer literals:
```
85 /* decimal */
0213 /* octal */
0x4b /* hexadecimal */
30 /* int */
30u /* unsigned int */
30l /* long */
30ul /* unsigned long */
```

Floating-point literals

A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.

While representing using decimal form, you must include the decimal point, the exponent, or both and while representing using exponential form, you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

Here are some examples of floating-point literals:
```
3.14159 /* Legal */
314159E-5L /* Legal */
510E /* Illegal: incomplete exponent */
210f /* Illegal: no decimal or exponent */
.e55 /* Illegal: missing integer or fraction */
```

Character constants

Character literals are enclosed in single quotes, e.g., 'x' and can be stored in a simple variable of **char** type.

A character literal can be a plain character (e.g., 'x'), an escape sequence (e.g., '\t'), or a universal character (e.g., '\u02C0').

There are certain characters in C when they are preceded by a backslash they will have

special meaning and they are used to represent like newline (\n) or tab (\t). Here, you have a list of some of such escape sequence codes:

Escape Meaning
sequence
\\
\ 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
Following is the example to show few escape sequence characters:

```c
#include <stdio.h>
int main()
{
printf("Hello\tWorld\n\n");
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Hello World
```

String literals

String literals or constants are enclosed in double quotes "". A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.

You can break a long line into multiple lines using string literals and separating them using whitespaces.

Here are some examples of string literals. All the three forms are identical strings.

```c
"hello, dear"
"hello, \
dear"
"hello, " "d" "ear"
```

Defining Constants

There are two simple ways in C to define constants:
1. Using **#define** preprocessor.
2. Using **const** keyword.

The #define Preprocessor

Following is the form to use #define preprocessor to define a constant:

```c
#define identifier value
```

Following example explains it in detail:

```
#include <stdio.h>
#define LENGTH 10
#define WIDTH 5
#define NEWLINE '\n'
int main()
{
int area;
area = LENGTH * WIDTH;
printf("value of area : %d",
area); printf("%c", NEWLINE);
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of area : 50
```

The const Keyword

You can use **const** prefix to declare constants with a specific type as follows:

```
const type variable = value;
```

Following example explains it in detail:

```
#include <stdio.h>
int main()
{
const int LENGTH = 10;
const int WIDTH = 5; const
char NEWLINE = '\n';
int area;
area = LENGTH * WIDTH;
printf("value of area : %d",
area); printf("%c", NEWLINE);
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of area : 50
```

Note that it is a good programming pract ice to def ine constants in CAPITALS.

CHAPTER 3

C Storage Classes

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program. These specifiers precede the type that they modify. There are the following storage classes, which can be used in a C Program

• auto

• register

• static

• extern

The auto Storage Class

The **auto** storage class is the default storage class for all local variables.

```
{
int mount; auto int
month;
}
```

The example above defines two variables with the same storage class, auto can only be used within functions, i.e., local variables.

The register Storage Class

The **register** storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

```
{
register int miles;
}
```

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

The static Storage Class

The **static** storage class instructs the compiler to keep a local variable in existence during the life - time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls. The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.
In C programming, when static is used on a class data member, it causes only one copy of that member to be shared by all objects of its class.

```
#include <stdio.h>
/* function declaration */ void
func(void);
static int count = 5; /* global variable */
main()
{
while(count--)
{
func();
}
```

```
return 0;
}
/* function definition */ void
fun c( void )
{
static int i = 5; /* local static variable */ i++;
printf("i is %d and count is %d\n", i, count);
}
```

You may not understand this example at this time because I have used function and global variables, which I have not explained so far. So for now, let us proceed even if you do not understand it completely. When the above code is compiled and executed, it produces the following result:

```
i is 6 and count is 4 i is 7
and count is 3 i is 8 and count
is 2 i is 9 and count is 1 i is
10 and count is 0
```

The extern Storage Class

The extern storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern', the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function, which will be used in other files also, then extern will be used in another file to give reference of defined variable or function. Just for understanding, extern is used to declare a global variable or function in another file.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

First File: main.c

```
#include <stdio.h>
int count ;
extern void write_extern();
main()
{
write_extern();
}
```

Second File: write.c

```
#include <stdio.h>
extern int count;
void write_extern(void)
{
count = 5;
printf("count is %d\n", count);
}
```

Here, extern keyword is being used to declare count in the second file where as it has its definition in the first file, main.c. Now, compile these two files as follows:

```
$gcc main.c write.c
```

This will produce **a.out** executable program, when this program is executed, it produces the following result:

5

CHAPTER 4

C Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical

manipulations. C language is rich in built-in operators and provides the following types of operators:
• Arithmetic Operators

• Relational Operators

• Logical Operators

• Bitwise Operators

• Assignment Operators

• Misc Operators
This tutorial will explain the arithmetic , relational, logical, bitwise, assignment and other operators one by one.

Arithmetic Operators

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

Operator
Description
Example
+
Adds two operands
A + B will give 30
-
Subtracts second operand from the first
A - B will give -10
*
Multiplies both operands
A * B will give 200
/
Divides numerator by de-numerator
B / A will give 2
%
Modulus Operator and remainder of after an integer division
B % A will give 0
++ Increments operator increases integer value by one A++ will give 11
--
Decrements operator decreases integer value by one
A-- will give 9

Try the following example to understand all the arithmetic operators available in C programming language:

```
#include <stdio.h>
main()
{
int a = 21; int b =
10; int c ;
c = a + b;
```

```
printf("Line 1 - Value of c is %d\n", c ); c = a -
b;
printf("Line 2 - Value of c is %d\n", c ); c = a *
b;
printf("Line 3 - Value of c is %d\n", c ); c = a /
b;
printf("Line 4 - Value of c is %d\n", c ); c = a %
b;
printf("Line 5 - Value of c is %d\n", c ); c = a++;
printf("Line 6 - Value of c is %d\n", c ); c = a--;
printf("Line 7 - Value of c is %d\n", c );
}
```
When you compile and execute the above program, it produces the following result:
```
Line 1 - Value of c is 31
Line 2 - Value of c is 11
Line 3 - Value of c is 210
Line 4 - Value of c is 2
Line 5 - Value of c is 1
Line 6 - Value of c is 21
Line 7 - Value of c is 22
```

Relational Operators

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

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

Try the following example to understand all the relational operators available in C programming language:

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

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

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

```
Line 1 - a is not equal to b
Line 2 - a is not less than b
Line 3 - a is greater than b
Line 4 - a is either less than or equal to b
Line 5 - b is either greater than or equal to b
```

Logical Operators

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

Operator
Description
Example
&&
Calle d Lo gical AN D op er ator. I f both the op er an ds are n on -z ero,
(A && B) is false.
then condition becomes true.
||
Called Logical OR Operator. If any of the two operands is non-
(A || B) is true.
zero, then condition becomes true.
Called Logical NOT Operator. Use to reverses the logical state
! of its operand. If a condition is true, then Logical NOT operator !(A && B) is true.
will make false.

Try the following example to understand all the logical operators available in C programming language:

```
#include <stdio.h>
main()
{
int a = 5; int b =
20; int c ;
if ( a && b )
{
printf("Line 1 - Condition is true\n" );
}
if ( a || b )
{
printf("Line 2 - Condition is true\n" );
```

```
}
/* lets change the value of a and b */ a = 0;
b = 10;
if ( a && b )
{
printf("Line 3 - Condition is true\n" );
}
else
{
printf("Line 3 - Condition is not true\n" );
}
if ( !(a && b) )
{
printf("Line 4 - Condition is true\n" );
}
}
```

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

```
Line 1 - Condition is true
Line 2 - Condition is true
Line 3 - Condition is not true
Line 4 - Condition is true
```

Bitwise Operators

Bitwise operator works on bits and performs bit-by-bit operation. The truth tables for &, |, and ^ are as follows:

P
q
p & q
p | q
p ^ q
0
0
0
0
0
0
1
0
1
1
1
1
1
1
1
0
1
0
0
1
1

Assume if A = 60; and B = 13; now in binary format they will be as follows:

A = 0011 1100
B = 0000 1101

A&B = 0000 1100
A|B = 0011 1101
A^B = 0011 0001
~A = 1100 0011

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

Operator Description Example
&

Binary AND Operator copies a bit to the
result if it exists in both operands.
(A & B) will give 12, which is 0000 1100

|
Binary OR Operator copies a bit if it
exists in either operand.
(A | B) will give 61, which is 0011 1101

Binary XOR Operator copies the bit if it
is set in one operand but not both.
^ (A ^ B) will give 49, which is 0011 0001

~
Binary Ones Complement Operator is
unary and has the effect of 'flipping' bits.
(~A) will give -60, which is 1100 0011

Binary Left Shift Operator. The left
<<
operands value is moved left by the
number of bits specified by the right
operand.
A << 2 will give 240, which is 1111 0000

Binary Right Shift Operator. The left
>>
operands value is moved right by the
number of bits specified by the right
operand.
A >> 2 will give 15, which is 0000 1111

Try the following example to understand all the bitwise operators available in C programming language:

```c
#include <stdio.h>
main()
{
unsigned in t a = 60; /* 60 = 0011 1100 */
unsigned in t b = 13; /* 13 = 0000 1101 */
int c = 0;
c = a & b; /* 12 = 0000 1100 */
printf("Lin e 1 - Value of c is %d\n", c );
c = a | b; /* 61 = 0011 1101 */
printf("Lin e 2 - Value of c is %d\n", c );
c = a ^ b; /* 49 = 0011 0001 */
printf("Lin e 3 - Value of c is %d\n", c );
c = ~a; /*-61 = 1100 0011 */
printf("Lin e 4 - Value of c is %d\n", c );
c = a << 2; /* 240 = 1111 0000 */
printf("Lin e 5 - Value of c is %d\n", c );
c = a >> 2; /* 15 = 0000 1111 */
printf("Line 6 - Value of c is %d\n", c );
}
```

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

```
Line 1 - Value of c is 12
Line 2 - Value of c is 61
Line 3 - Value of c is 49
Line 4 - Value of c is -61
Line 5 - Value of c is 240
Line 6 - Value of c is 15
```

Assignment Operators

There are following assignment operators supported by C language:

Operator
Description
Example
= Simple assignment operator, Assigns values
from right side operands to left side operand
C = A + B will assign value of A +
B into C

Add AND assignment operator, It adds right
+= op era n d to th e lef t op era n d an d assign the result C += A is equivalent to C = C + A
to left operand
Subtract AND assignment operator, It subtracts
-= right operand from the left operand and assign C -= A is equivalent to C = C - A
the result to left operand
Multiply AND assignment operator, It multiplies
*= right operand with the left operand and assign C *= A is equivalent to C = C * A
the result to left operand
Divide AND assignment operator, It divides left
/= operand with the right operand and assign the C /= A is equivalent to C = C / A
result to left operand
Modulus AND assignment operator, It takes
%= modulus using two operands and assign the C %= A is equivalent to C = C % A
result to left operand
<<=
Left shift AND assignment operator
C <<= 2 is same as C = C << 2
>>=
Right shift AND assignment operator
C >>= 2 is same as C = C >> 2
&=
Bitwise AND assignment operator
C &= 2 is same as C = C & 2
^=
bitwise exclusive OR and assignment operator
C ^= 2 is same as C = C ^ 2
|=
bitwise inclusive OR and assignment operator
C |= 2 is same as C = C | 2
Try the following example to understand all the assignment operators available in C programming
language:

```c
#include <stdio.h>
main()
{
int a = 21; int c ;
c = a;
printf("Line 1 - = Operator Example, Value of c = %d\n", c );
c += a;
printf("Line 2 - += Operator Example, Value of c = %d\n", c );
c -= a;
printf("Line 3 - -= Operator Example, Value of c = %d\n", c );
c *= a;
printf("Line 4 - *= Operator Example, Value of c = %d\n", c );
c /= a;
printf("Line 5 - /= Operator Example, Value of c = %d\n", c );
c = 200; c %= a;
printf("Line 6 - %= Operator Example, Value of c = %d\n", c );
c <<= 2;
printf("Line 7 - <<= Operator Example, Value of c = %d\n", c );
c >>= 2;
printf("Line 8 - >>= Operator Example, Value of c = %d\n", c );
c &= 2;
printf("Line 9 - &= Operator Example, Value of c = %d\n", c );
c ^= 2;
printf("Line 10 - ^= Operator Example, Value of c = %d\n", c );
c |= 2;
printf("Line 11 - |= Operator Example, Value of c = %d\n", c );
}
```

When you compile and execute the above program, it produces the following result:
Line 1 - = Operator Example, Value of c = 21
Line 2 - += Operator Example, Value of c = 42
Line 3 - -= Operator Example, Value of c = 21

```
Line 4 - *= Operator Example, Value of c = 441
Line 5 - /= Operator Example, Value of c = 21
Line 6 - %= Operator Example, Value of c = 11
Line 7 - <<= Operator Example, Value of c = 44
Line 8 - >>= Operator Example, Value of c = 11
Line 9 - &= Operator Example, Value of c = 2
Line 10 - ^= Operator Example, Value of c = 0
Line 11 - |= Operator Example, Value of c = 2
```

Misc Operators ↦sizeof & ternary

There are few other important operators including **sizeof** and **? :** supported by C Language.

Operator Description Example

sizeof()

Returns the size of an variable.

sizeof(a), where a is integer,

will return 4.

&

Returns the address of an variable.

&a; will give actual address of

the variable.

*

Pointer to a variable.

*a; will pointer to a variable.

? :

Conditional Expression

If Condition is true ? Then

value X : Otherwise value Y

Operators Precedence in C

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

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

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

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

Try the following example to understand the operator precedence available in C programming

language:

```c
#include <stdio.h>
main()
{
int a = 20;
int b = 10;
int c = 15;
int d = 5;
int e;
e = (a + b) * c / d; // ( 30 * 15 ) / 5
printf("Value of (a + b) * c / d is : %d\n", e );
e = ((a + b) * c) / d; // (30 * 15 ) / 5
printf("Val ue of (( a + b) * c) / d is : %d\n" , e );
e = (a + b) * (c / d); // (30) * (15/5)
printf("Val ue of (a + b) * (c / d) is : %d\n", e );
e = a + (b * c) / d; // 20 + (150/5)
printf("Value of a + (b * c) / d is : %d\n" , e );
return 0;
}
```

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

```
Value of (a + b) * c / d is : 90 Value of
((a + b) * c) / d is : 90 Value of (a + b) *
(c / d) is : 90 Value of a + (b * c) / d is
    : 50
```

Decision Making in C

D ecision making structures require that the programmer specify one or more

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

Following is the general form of a typical decision making structure found in most of the programming languages:

C programming language assumes any non-zero and non-null values as true, and if it is either zero or null, then it is assumed as false value. C programming language provides

if statement

An **if** statement consists of a boolean expression followed by one or more statements.

Syntax

The syntax of an if statement in C programming language is:

```
if(boolean_expression)
{
/* statement(s) will execute if the boolean expression is true */
}
```

If the **boolean** expression evaluates to **true,** then the block of code inside the **if** statement will be executed. If **boolean** expression evaluates to **false,** then the first set of code after the end of the **if** statement (after the closing curly brace) will be executed.

C programming language assumes any **non-zero** and **non-null** values as **true** and if it is either **zero** or **null** then it is assumed as **false** value.

Flow Diagram

Example

```
#include <stdio.h>
int main ()
{
/* local variable definition */ int a =
10;
/* check the boolean condition using if statement */ if( a <
20 )
{
/* if condition is true then print the following */ printf("a
is less than 20\n" );
}
printf("value of a is : %d\n", a);
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
a is less than 20;
value of a is : 10
```

if...else statement

An **if** statement can be followed by an optional **else** statement, which executes when the boolean expression is **false**.

Syntax

The syntax of an **if...else** statement in C programming language is:

```c
if(boolean_expression)
{
/* statement(s) will execute if the boolean expression is true */
}
else
{
/* statement(s) will execute if the boolean expression is false */
}
```

If the **boolean** expression evaluates to **true,** then the if block of code will be executed, otherwise **else** block of code will be executed.

C programming language assumes any **non-zero** and **non-null** values as **true** and if it is either **zero** or **null** then it is assumed as **false** value.

Flow Diagram

Example

```c
#include <stdio.h>
int main ()
{
/* local variable definition */ int a = 100;
/* check the boolean condition */ if( a < 20 )
{
/* if condition is true then print the following */ printf("a is less than 20\n" );
}
else
{
/* if condition is false then print the following */ printf("a is not less than 20\n" );
}
printf("value of a is : %d\n", a);
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
a is not less than 20;
value of a is : 100
```

The if...else if...else Statement

An **if** statement can be followed by an optional **else if...else** statement, which is very useful to test various conditions using single **if...else if** statement.

When using **if** , **else if** , **else** statements there are few points to keep in mind:

• An **if** can have zero or one else's and it must come after any **else if's**.

• An **if** can have zero to many **else if's** and they must come before the **else**.

• Once an **else if** succeeds, none of the remaining **else if's** or **else's** will be tested.

Syntax

The syntax of an **if...else if...else** statement in C programming language is:

```c
if(boolean_expression 1)
{
/* Executes when the boolean expression 1 is true */
}
else if( boolean_expression 2)
{
/* Executes when the boolean expression 2 is true */
}
else if( boolean_expression 3)
{
/* Executes when the boolean expression 3 is true */
```

```
}
else
{
/* executes when the none of the above condition is true */
}
```

Example

```c
#include <stdio.h>
int main ()
{
/* local variable definition */ int a =
100;
/* check the boolean condition */ if( a ==
10 )
{
/* if condition is true then print the following */
printf("Value of a is 10\n" );
}
else if( a == 20 )
{
/* if else if condition is true */
printf("Value of a is 20\n" );
}
else if( a == 30 )
{
/* if else if condition is true */
printf("Value of a is 30\n" );
}
else
{
/* if none of the conditions is true */ printf("None of
the values is matching\n" );
}
printf("Exact value of a is: %d\n", a );
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
None of the values is matching
Exact value of a is: 100
```

Nested if statements

It is always legal in C programming to **nest if-else** statements, which means you can use one **if** or **else if** statement inside another **if** or **else if** statement(s).

Syntax

The syntax for a **nested if** statement is as follows:

```c
if( boolean_expression 1)
{
/* Executes when the boolean expression 1 is true */
if(boolean_expression 2)
{
/* Executes when the boolean expression 2 is true */
}
}
```

You can nest else **if...else** in the similar way as you have **nested if** statement.

Example

```c
#include <stdio.h>
int main ()
{
/* local variable definition */ int a =
100;
```

```
int b = 200;
/* check the boolean condition */
if( a == 100 )
{
/* if condition is true then check the following */ if( b ==
200 )
{
/* if condition is true then print the following */
printf("Value of a is 100 and b is 200\n" );
}
}
printf("Exact value of a is : %d\n", a );
printf("Exact value of b is : %d\n", b );
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200
```

switch statement

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each switch case.

Syntax

The syntax for a **switch** statement in C programming language is as follows:

```
switch(expression){
case constant-expression :
statement(s);
break; /* optional */ case
constant-expression :
statement(s);
break; /* optional */
/* you can have any number of case statements */ default :
/* Optional */
statement(s);
}
```

The following rules apply to a **switch** statement:

• The **expression** used in a **switch** statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.

• You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.

• The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.

• When the variable being switched on is equal to a case, the statements following that case will execute until a **break** statement   is reached.

• When a **break** statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.

• Not every case needs to contain a **break**. If no **break** appears, the flow of control will *fall through* to subsequent cases until a break is reached.

• A **switch** statement can have an optional **default** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No **break** is needed in the default case.

Flow Diagram

Example

```c
#include <stdio.h>
int main ()
{
/* local variable definition */ char
grade = 'B';
switch(grade)
{
case 'A' :
printf("Excellent!\n" ); break;
case 'B' : case 'C' :
printf("Well done\n" ); break;
case 'D' :
printf("You passed\n" ); break;
case 'F' :
printf("Better try again\n" ); break;
default :
printf("Invalid grade\n" );
}
printf("Your grade is %c\n", grade );
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Well done
Your grade is B
```

Nested switch statements

It is possible to have a **switch** as part of the statement sequence of an **outer switch**. Even if the case constants of the inner and outer switch contain common values, no conflic ts will arise.

Syntax

The syntax for a **nested switch** statement is as follows:

```c
switch(ch1) { case 'A':
printf("This A is part of outer switch" );
switch(ch2) {
case 'A':
printf("This A is part of inner switch" ); break;
case 'B': /* case code */
}
break;
case 'B': /* case code */
}
```

Example

```c
#include <stdio.h>
int main ()
{
/* local variable definition */ int a =
100;
int b = 200;
switch(a) { case 100:
printf("This is part of outer switch\n", a ); switch(b)
{
case 200:
printf("This is part of inner switch\n", a );
}
}
printf("Exact value of a is : %d\n", a );
printf("Exact value of b is : %d\n", b );
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
This is part of outer switch
This is part of inner switch
Exact value of a is : 100
Exact value of b is : 200
```

The ? : Operator

We have covered conditional operator **? :** in previous chapter which can be used to replace **if…else** statements. It has the following general form:

```
Exp1 ? Exp2 : Exp3;
```

Where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.

The value of a ? expression is determined like this: Exp1 is evaluated. If it is **true**, then Exp2 is evaluated and becomes the value of the entire **?** expression. If Exp1 is **false**, then Exp3 is evaluated and its value becomes the value of the expression.

CHAPTER 6

T here may be a situation, when you need to execute a block of code several number of times. In general, statements are executed sequentially: The **first** statement in a function is executed first, followed by the **second**, and **so on**.

Programming languages provide various control structures that allow for more complicated execution paths.

A **loop** statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages

C programming language provides the following types of loops to handle looping requirements.

while loop in C

A **while** loop statement in C programming language repeatedly executes a target statement as long as a given condition is **true**.

Syntax

The syntax of a **while** loop in C programming language is:

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

Here, statement(s) may be a single statement or a block of statements. The **condition** may be any expression, and **true** is any **nonzero** value. The loop iterates while the condition is **true**.

When the condition becomes **false**, program control passes to the line immediately following the loop.

Flow Diagram

Here, key point of the while loop is that the loop might not ever run. When the condition is tested and the result is **false**, the loop body will be skipped and the first statement after the while loop will be executed.

Example

```
#include <stdio.h>
int main ()
{
/* local variable definition */ int a
= 10;
/* while loop execution */ while(
a < 20 )
{
printf("value of a: %d\n", a); a++;
}
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10 value
of a: 11 value of a: 12
value of a: 13 value of
a: 14 value of a: 15
value of a: 16 value of
a: 17 value of a: 18
value of a: 19
```

for loop in C

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax

The syntax of a **for** loop in C programming language is:

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

Here is the flow of control in a for loop:

1. The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
2. Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.
3
.
After the
body
of
the for
l
oop
exec
utes,
the flow of control jumps back up
to
the
increment
statem
ent.
T
his
statem
ent
allows you to update any loop
control
variables. This statement can be left blank, as long as a semicolon appears after the condition.
4. The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates.

Flow Diagram

Example

```
#include <stdio.h>
int main ()
{
/* for loop execution */
for( int a = 10; a < 20; a = a + 1 )
{
printf("value of a: %d\n", a);
}
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10 value
of a: 11 value of a: 12
value of a: 13 value of
a: 14 value of a: 15
value of a: 16 value of
a: 17 value of a: 18
value of a: 19
```

do...while loop in C

Unlike for and while loops, which test the loop condition at the top of the loop, the **do...while** loop in C programming language checks its condition at the bottom of the loop.
A **do...while** loop is similar to a while loop, except that a **do...while** loop is guaranteed to execute at least one time.

Syntax

The syntax of a **do...while** loop in C programming language is:

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

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop execute once before the condition is teste
If the condition is **true**, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes **false**.

Flow Diagram

Example

```
#include <stdio.h>
int main ()
{
/* local variable definition */ int a
= 10;
/* do loop execution */ do
{
printf("value of a: %d\n", a); a = a
+ 1;
}while( a < 20 );
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10 value
of a: 11 value of a: 12
value of a: 13 value of
a: 14
value of a: 15 value
of a: 16 value of a: 17
value of a: 18 value of
a: 19
```

nested loops in C

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

Syntax

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

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

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

```
while(condition)
{
while(condition)
{
```

```
statement(s);
}
statement(s);
}
```
The syntax for a **nested do...while** loop statement in C programming language is as follows:
```
do
{
statement(s); do
{
statement(s); }while(
condition );
}while( condition
```
A final note on loop nesting is that you can put any type of loop inside of any other type of loop. For example, a **for loop** can be inside a while loop or vice versa.

Example

The following program uses a nested for loop to find the prime numbers from 2 to 100:
```
#include <stdio.h>
int main ()
{
/* local variable definition */ int i,
j;
for(i=2; i<100; i++) { for(j=2; j <=
(i/j); j++)
if(!(i%j)) break; // if factor found, not prime if(j >
(i/j)) printf("%d is prime\n", i);
}
return 0;
}
```
When the above code is compiled and executed, it produces the following result:
```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
97 is prime
```

break statement in C

The **break** statement in C programming language has the following two usages:
1. When the **break** statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.
2. It can be used to terminate a case in the **switch** statement (covered in the next chapter).
If you are using **nested loops** (i.e., one loop inside another loop), the **break** statement will stop

the execution of the innermost loop and start executing the next line of code after the block.

Syntax

The syntax for a **break** statement in C is as follows:

```
break;
```

Flow Diagram

Example

```c
#include <stdio.h>
int main ()
{
/* local variable definition */ int a
= 10;
/* while loop execution */ while(
a < 20 )
{
printf("value of a: %d\n", a); a++;
if( a > 15)
{
/* terminate the loop using break statement */ break;
}
}
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10 value
of a: 11 value of a: 12
value of a: 13 value of
a: 14 value of a: 15
```

continue statement in C

The **continue** statement
in C programming language
wor
ks
somew
hat
l
ike
the break statement.
Instead
of forcing termination,
however,
cont
inue
forces
the
n
ext
iteration of the loop to take place, skipping any code in between.

For the **for loop**, **continue** statement causes the conditional test and increment por tions of the loop to execute. For the **while** and **do...while** loops, **continue** statement causes the program control passes to the conditional tests.

Syntax

The syntax for a **continue** statement in C is as follows:

```
continue;
```

Flow Diagram

Example

```
#include <stdio.h>
int main ()
{
/* local variable definition */ int a = 10;
/* do loop execution */ do
{
if( a == 15)
{
/* skip the iteration */ a = a + 1;
continue;
}
printf("value of a: %d\n", a); a++;
}while( a < 20 );
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12 value
of a: 13 value of a: 14
value of a: 16 value of
a: 17 value of a: 18
value of a: 19
```

goto statement in C

A **goto** statement in C programming language provides an unconditional jump from the **goto** to a labeled statement in the same function.

NOTE: Use of **goto** statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a **goto** can be rewritten so that it doesn't need the **goto**.

Syntax

The syntax for a **goto** statement in C is as follows:

```
goto label;
..
.
label: statement;
```

Here **label** can be any plain text except C keyword and it can be set anywhere in the C program above or below to **goto** statement.

Flow Diagram

Example

```
#include <stdio.h>
int main ()
{
/* local variable definition */ int a
= 10;
/* do loop execution */
LOOP:do
{
if( a == 15)
{
/* skip the iteration */
a = a + 1; goto
LOOP;
}
printf("value of a: %d\n", a); a++;
}while( a < 20 );
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10 value
of a: 11 value of a: 12
value of a: 13 value of
a: 14 value of a: 16
value of a: 17 value of
a: 18 value of a: 19
```

The Infinite Loop

A loop becomes **infinite** loop if a condition never becomes **false**. The **for loop** is traditionally used for this purpose. Since **none** of the three expressions that form the **for loop** are required, you can make an endless loop by leaving the conditional expression empty.

```c
#include <stdio.h>
int main ()
{
for( ; ; )
{
printf("This loop will run forever.\n");
}
return 0;
}
```

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C programmers more commonly use the **for(;;)** construct to signify an infinite loop.

NOTE: You can terminate an infinite loop by pressing **Ctrl + C** keys.

CHAPTER 7

C Functions

Function is a group of statements that together perform a task. Every C program has at least

one function, which is **main(),** and all the most trivial programs can define additional functions.
You can divide up your code into separate functions. How you divide up your code among different
functions is up to you, but logically the division usually is so each function performs a specific task.
A function **declaration** tells the compiler about a function's name, return type, and parameters. A
function **definition** provides the actual body of the function.
The C standard library provides numerous built-in functions that your program can call. For
example, function **strcat()** to concatenate two strings, function **memcpy()** to copy one memory
location to another location and many more functions.
A function is known with various names like a method or a sub-routine or a procedure, etc.

Defining a Function

The general form of a **function definition** in C programming language is as follows:

```
return_type function_name( parameter list )
{
body of the function
}
```

A function definition in C programming language consists of a function header and a function body.
Here are all the parts of a function:

• **Return Type**: A function may return a value. The **return_type** is the data type of the value the
function returns. Some functions perform the desired operations without returning a value. In this

case, the return_type is the keyword **void**.

• **Function Name:** This is the actual name of the function. The function name and the parameter list
together constitute the function signature.

• **Parameters:** A parameter is like a placeholder. When a function is invoked, you pass a value to the
parameter. This value is referred to as actual parameter or argument. The
parameter list refers to the type, order, and number of the parameters of a function. Parameters
are optional; that is, a function may contain no parameters.
• **Function Body:** The function body contains a collection of statements that define what the function

does.

Example

Following is the source code for a function called max(). This function takes two parameters num1
and num2 and returns the maximum between the two:

```
/* function returning the max between two numbers */ int
max (int num1, int num2)
{
/* local variable declaration */ int
result;
if (num1 > num2) result =
num1;
```

```
else
result = num2;
return result;
}
```

Function Declarations

A **function declaration** tells the compiler about a function name and how to call the function. The actual **body of the function** can be defined separately.

A function declaration has the following par ts:

```
return_type function_name( parameter list );
```

For the above defined function max(), following is the function declaration:

```
int max(int num1, int num2);
```

Parameter names are not important in function declaration only their type is required, so following is also valid declaration:

```
int max(int, int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case you should declare the function at the top of the file calling the function.

Calling a Function

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, program control is transferred to the called function. A cal led function performs defined task, and when its return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example:

```
#include <stdio.h>
/* function declaration */ int
max(int num1, int num2);
int main ()
{
/* local variable definition */ int a
= 100;
int b = 200; int
ret;
/* calling a function to get max value */ ret =
max(a, b);
printf( "Max value is : %d\n", ret );
return 0;
}
/* function returning the max between two numbers */ int
max (int num1, int num2)
{
/* local variable declaration */ int
result;
if (num1 > num2) result =
num1;
else
result = num2;
return result;
}
```

I kept max() function along with main() function and compiled the source code. While running final executable, it would produce the following result:

```
Max value is : 200
```

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are **two ways** that arguments can be passed to a function:

Function call by value

The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

By default, C programming language uses call by value method to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function. Consider the function swap() definition as follows.

```c
/* function definition to swap the values */ void
swap(int x, int y)
{
int temp;
temp = x; /* save the value of x */ x = y;
/* put y into x */
y = temp; /* put x into y */
return;
}
```

Now, let us call the function swap() by passing actual values as in the following example:

```c
#include <stdio.h>
/* function declaration */ void
swap(int x, int y);
int main ()
{
/* local variable definition */ int a
= 100;
int b = 200;
printf("Before swap, value of a : %d\n", a );
printf("Before swap, value of b : %d\n", b );
/* calling a function to swap the values */
swap(a, b);
printf("After swap, value of a : %d\n", a );
printf("After swap, value of b : %d\n", b );
return 0;
}
```

Let us put above code in a single C file, compile and execute it, it will produce the following result:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```

Which shows that there is no change in the values though they had been changed inside the function.

Function call by reference

The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

To pass the **value by reference**, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function swap(), which exchanges the values of the two integer variables pointed to by its arguments.

```c
/* function definition to swap the values */ void
swa p(int *x, int *y)
{
int temp;
temp = *x; /* save the value at address x */
*x = *y; /* put y into x */
*y = temp; /* put x into y */
return;
}
```

Let us call the function swap() by passing values by reference as in the following example:

```c
#include <stdio.h>
/* function declaration */ void
swap(int *x, int *y);
```

```c
int main ()
{
/* local variable definition */ int a
= 100;
int b = 200;
printf("Before swap, value of a : %d\n", a );
printf("Before swap, value of b : %d\n", b );
/* calling a function to swap the values.
* &a indicates pointer to a ie. address of variable a and
* &b indicates pointer to b ie. address of variable b.
*/
swap(&a, &b);
printf("After swap, value of a : %d\n", a );
printf("After swap, value of b : %d\n", b );
return 0;
}
```

Let us put above code in a single C file, compile and execute it, it will produce the following result:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

Which shows that there is no change in the values though they had been changed inside the function.

CHAPTER 8

C Scope Rules

A scope in any programming is a region of the program where a defined variable can have

its existence and beyond that variable cannot be accessed. There are three places where variables can be declared in C programming language:

• Inside a function or a block which is called **local** variables,

• Outside of all functions which is called   **global** variables.

• In the definition of function parameters which is called **formal** parameters.
Let us explain what are **local** and **global** variables and **formal** parameters.

Local Variables

Variables that are declared inside a function or block are called **local variables**. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. Following is the example using **local variables**. Here all the variables a, b and c are local to **main()** function.

```c
#include <stdio.h>
int main ()
{
/* local variable declaration */ int
a, b;
int c;
/* actual initialization */
a = 10;
b = 20;
c = a + b;
printf ("value of a = %d, b = %d and c = %d\n", a, b, c);
return 0;
}
```

Global Variables

Global variables are defined outside of a function, usually on top of the program. The global variables will hold their value throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.
A **global variable** can be accessed by any function. That is, a **global variable** is available for use throughout your entire program after its declaration. Following is the example using global and local variables:

```c
#include <stdio.h>
/* global variable declaration */ int g;
int main ()
{
/* local variable declaration */ int a,
b;
/* actual initialization */
a = 10;
b = 20;
g = a + b;
printf ("value of a = %d, b = %d and g = %d\n", a, b, g);
return 0;
}
```

A program can have same name for local and global variables but value of local variable inside a function will take preference. Following is an example:

```c
#include <stdio.h>
/* global variable declaration */ int g
= 20;
int main ()
{
/* local variable declaration */ int g
= 10;
printf ("value of g = %d\n", g);
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of g = 10
```

Formal Parameters

Function parameters, so called **formal parameters**, are treated as local variables within that function and they will take preference over the global variables. Following is an example:

```c
#include <stdio.h>
/* global variable declaration */

int a = 20;
int main ()
{
/* local variable declaration in main function */

int a = 10;
int b = 20;
int c = 0;
printf ("value of a in main() =
%d\n",
a
);
c = sum( a, b);
pri
ntf
("value of c in main() =
%d\n",
c
);
ret
urn 0;
}
/* function to add two integers */
int sum(int a, int b)
{
printf ("value of a in sum() = %d\n", a);
printf ("value of b in sum() = %d\n", b);
return a + b;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a in main() = 10
value of a in sum() = 10 value
of b in sum() = 20 value of c in
main() = 30
```

Initializing Local and Global Variables

When a local variable is defined, it is not initialized by the system, you must initialize it yourself. Global variables are initialized automatically by the system when you define them as follows:

Data Type
Initial Default Value
int
0
char '\0'
float

0
double
0
pointer
NULL
It is a good programming practice to initialize variables properly otherwise, your program may produce unexpected results because uninitialized variables will take some garbage value already available at its memory location.

C Arrays

C programming language provides a data structure called the **array**, which can store

a fixed-size sequential collection of elements of the same type. An **array** is used to store a collection of data, but it is of ten more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an **index**.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows:

```
type arrayName [ arraySize ];
```

This is called a single-dimensional array. The **arraySize** must be an integer constant greater than zero and type can be any valid C data type. For example, to declare a 10-element array called balance of type double, use this statement:

```
double balance[10];
```

Now balance is a variable array which is sufficient to hold up-to 10 double numbers.

Initializing Arrays

You can initialize array in C either one by one or using a single statement as follows:

```
double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array:

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

```
double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

You will create exactly the same array as you did in the previous example.

```
balance[4] = 50.0;
```

The above statement assigns element number 5th in the array a value of 50.0. Array with 4th index will be 5th i.e. last element because all arrays have 0 as the index of their f irst element which is also called base index. Following is the pictorial representation of the same array we discussed above:

Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:

```
double salary = balance[9];
```

The above statement will take 10th element from the array and assign the value to salary variable. Following is an example which will use all the above mentioned three concepts viz. declaration, assignment and accessing arrays:

```
#include <stdio.h>
int main ()
{
int n[ 10 ]; /* n is an array of 10 integers */ int
i,j;
/* initialize elements of array n to 0 */ for (
i = 0; i < 10; i++ )
{
n[ i ] = i + 100; /* set element at location i to i + 100 */
```

```
}
/* output each array element's value */ for (j
= 0; j < 10; j++ )
{
printf("Element[%d] = %d\n", j, n[j] );
}
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

Multi-dimensional Arrays

C programming language allows **multidimensional arrays**. Here is the general form of a multidimensional array declaration:

```
type name[size1][size2]...[sizeN];
```

For example, the following declaration creates a three dimensional 5 . 10 . 4 integer array:

```
int threedim[5][10][4];
```

Two-Dimensional Arrays

The simplest form of the **multidimensional** array is the **two-dimensional** array. A twodimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size x, y you would write something as follows:

```
type arrayName [ x ][ y ];
```

Where type can be any valid C data type and **arrayName** will be a valid C identifier. A twodimensional array can be think as a table which will have x number of rows and y number of columns.

A 2-dimentional array **a**, which contains three rows and four columns can be shown as below:

Thus, every element in array a is identified by an element name of the form a[i][j], where a is the name of the array, and i and j are the subscripts that uniquely identify each element in a.

Initializing Two-Dimensional Arrays

Multidimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

```
int a[3][4] =
{
{0, 1, 2, 3}
,
/
*
initializers for row indexed
by 0
*
/
{4, 5, 6, 7}
,
/
*
initializers for row
indexed
b
y 1
*
/
```

```
{8, 9, 10,
11}
/
*
initializers for row
indexed
b
y 2
*
/
}
;
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to previous example:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Accessing Two-Dimensional Array Elements

An element in 2-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example:

```
int val = a[2][3];
```

The above statement will take 4th element from the 3rd row of the array. You can verify it in the above diagram. Let us check below program where we have used nested loop to handle a two dimensional array:

```
#include <stdio.h>
int main ()
{
/* an array with 5 rows and 2 columns*/
int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}}; int i,
j;
/* output each array element's value */ for (
i = 0; i < 5; i++ )
{
for ( j = 0; j < 2; j++ )
{
printf("a[%d][%d] = %d\n", i,j, a[i][j] );
}
}
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
a[0][0]: 0
a[0][1]: 0 a[1][0]: 1
a[1][1]: 2 a[2][0]: 2
a[2][1]: 4 a[3][0]: 3
a[3][1]: 6 a[4][0]: 4
a[4][1]: 8
```

As explained above, you can have arrays with any number of dimensions, although it is likely that most of the arrays you create will be of one or two dimensions.

Passing Arrays as Function Arguments

If you want to pass a single-dimension array as an argument in a function, you would have to declare function formal parameter in one of following three ways and all three declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received. Similar way you can pass multi-dimensional array as formal parameters.

Way-1

Formal parameters as a pointer as follows. You will study what is pointer in next chapter.

```
void myFunction(int *param)
{
.
.
.
```

```
}
```
Way-2

Formal parameters as a sized array as follows:
```c
void myFunction(int param[10])
{
.
.
.
}
```

Way-3

Formal parameters as an unsized array as follows:
```c
void myFunction(int param[])
{
.
.
.
}
```

Example

Now, consider the following function, which will take an array as an argument along with another argument and based on the passed arguments, it will return average of the numbers passed through the array as follows:
```c
double getAverage(int arr[], int size)
{
int i; double avg;
double sum;
for (i = 0; i < size; ++i)
{
sum += arr[i];
}
avg = sum / size;
return avg;
}
```
Now, let us call the above function as follows:
```c
#include <stdio.h>
/* function declaration */
double getAverage(int arr[], int size); int
main ()
{
/* an int array with 5 elements */
int balance[5] = {1000, 2, 3, 17, 50}; double
avg;
/* pass pointer to the array as an argument */ avg =
getAverage( balance, 5 ) ;
/* output the returned value */ printf(
"Average value is: %f ", avg );
return 0;
}
```
When the above code is compiled together and executed, it produces the following result:
```
Average value is: 214.400000
```
As you can see, the length of the array doesn't matter as far as the function is concerned because C performs no bounds checking for the formal parameters.

Return array from function

C programming language does not allow to return an entire array as an argument to a function. However, you can return a pointer to an array by specifying the array's name without an index. You will study pointer in next chapter so you can skip this chapter until you understand the concept of Pointers in C.

If you want to return a single-dimension array from a function, you would have to declare a function returning a pointer as in the following example:

```
int * myFunction()
{
.
.
.
}
```

Second point to remember is that C does not advocate to return the address of a local variable to outside of the function so you would have to define the **local variable** as **static variable**.

Now, consider the following function which will generate 10 random numbers and return them using an array and call this function as follows:

```
#include <stdio.h>
/* function to generate and return random numbers */ int *
getRandom( )
{
static int r[10]; int i;
/* set the seed */
srand( (unsigned)time( NULL ) ); for (
i = 0; i < 10; ++i)
{
r[i] = rand();
printf( "r[%d] = %d\n", i, r[i]);
}
return r;
}
/* main function to call above defined function */ int
mai n ()
{
/* a pointer to an int */ int
*p;
int i;
p = getRandom();
for ( i = 0; i < 10; i++ )
{
printf( "*(p + %d) : %d\n", i, *(p + i));
}
return 0;
}
```

When the above code is compiled together and executed, it produces result something as follows:

```
r[0] = 313959809 r[1] =
1759055877 r[2] = 1113101911
r[3] = 2133832223 r[4] =
2073354073 r[5] = 167288147 r[6]
= 1827471542 r[7] = 834791014
r[8] = 1901409888 r[9] =
1990469526 *(p + 0) : 313959809
*(p + 1) : 1759055877 *(p +
2) : 1113101911 *(p + 3) :
2133832223 *(p + 4) : 2073354073
*(p + 5) : 167288147 *(p + 6) :
1827471542 *(p + 7) : 834791014
*(p + 8) : 1901409888 *(p + 9) :
1990469526
```

Pointer to an Array

It is most likely that you would not understand this chapter until you are through the chapter related to Pointers in C.

So assuming you have bit understanding on **pointers** in C programming language, let us start: An array name is a constant pointer to the first element of the array. Therefore, in the declaration:

```
double balance[50];
```

balance is a pointer to &balance[0], which is the address of the first element of the array balance. Thus, the following program fragment assigns p the address of the first element of balance:

```
double *p;
double balance[10]; p =
```

balance;

It is legal to use array names as constant pointers, and vice versa. Therefore, ***(balance + 4)** is a legitimate way of accessing the data at balance[4].

Once you store the address of first element in p, you can access array elements using *p, *(p+1), *(p+2) and so on. Below is the example to show all the concepts discussed above:

```c
#include <stdio.h>
int main ()
{
/* an array with 5 elements */
double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0}; double
*p;
int i;
p = balance;
/* output each array element's value */ printf(
"Array values using pointer\n"); for ( i = 0; i <
5; i++ )
{
printf("*(p + %d) : %f\n", i, *(p + i) );
}
printf( "Array values using balance as address\n"); for (
i = 0; i < 5; i++ )
{
printf("*(balance + %d) : %f\n", i, *(balance + i) );
}
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Array values using pointer *(p +
0) : 1000.000000
*(p + 1) : 2.000000 *(p + 2)
: 3.400000 *(p + 3) : 17.000000
*(p + 4) : 50.000000
Array values using balance as address
*(balance + 0) : 1000.000000 *(balance + 1) :
2.000000
*(balance + 2) : 3.400000
*(balance + 3) : 17.000000 *(balance +
4) : 50.000000
```

In the above example, p is a pointer to double, which means it can store address of a variable of double type. Once we have address in **p**, then ***p** will give us value available at the address stored in p, as we have shown in the above example.

C Strings

The string in C programming language is actually a one-dimensional array of characters which is terminated by a null character '\0'. Thus a **null-terminated** string contains the characters that comprise the string followed by a null.

The following declaration and initialization create a string consisting of the word **"Hello"**. To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word **"Hello"**.

```c
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

If you follow the rule of array initialization then you can write the above statement as follows:

```c
char greeting[] = "Hello";
```

Following is the memory presentation of above-defined string in C/C++:

Actually, you do not place the null character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print above mentioned string:

```c
#include <stdio.h>
int main ()
{
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
printf("Greeting message: %s\n", greeting );
return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Greeting message: Hello
```

C supports a wide range of functions that manipulate null -terminated strings:

S.N. Function & Purpose

1 **strcpy(s1, s2);** Copies string s2 into string s1.

2 **strcat(s1, s2);**
Concatenates string s2 onto the end of string s1.

3 **strlen(s1);** Returns the length of string s1.

4 **strcmp(s1, s2);**
Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.

5 **strchr(s1, ch);** Returns a pointer to the first occurrence of character ch in string s1.

6 **strstr(s1, s2);** Returns a pointer to the first occurrence of string s2 in string s1.

Following example makes use of few of the above-mentioned functions:

```c
#include <stdio.h> #include
<string.h>
int main ()
{
char str1[12] = "Hello"; char
str2[12] = "World"; char str3[12];
int len ;
/* copy str1 into str3 */
strcpy(str3, str1);
printf("strcpy( str3, str1) : %s\n", str3 );
/* concatenates str1 and str2 */ strcat(
str1, str2);
printf("strcat( str1, str2): %s\n", str1 );
/* total lenghth of str1 after concatenation */ len =
strlen(str1);
printf("strlen(str1) : %d\n", len );
return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

C Pointers

P ointers in C are easy and fun to learn. Some C programming tasks are performed

more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be
performed without using pointers. So it becomes necessary to learn pointers to become a
perfect C programmer. Let's start learning them in simple and easy steps.
As you know, every variable is a memory
address defined which can be accessed using
address in memory.
location and every memory location has its
ampersand (&) operator, which denotes an
Consider the following example, which will print the address of the variables defined:

```c
#include <stdio.h>
int main ()
{
int var1; char
var2[10];
printf("Address of var1 variable: %x\n", &var1 );
printf("Address of var2 variable: %x\n", &var2 );
return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var1 variable: bff5a400
Address of var2 variable: bff5a3f6
```

So you understood what is memory address and how to access it, so base of the concept is
over. Now let us see what is a pointer.

What Are Pointers?

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

```c
type *var-name;
```

Here, type is the pointer's base type; it must be a valid C data type and var-name is the name of the
pointer variable. The asterisk * you used to declare a pointer is the same asterisk that you use for
multiplication. However, in this statement the asterisk is being used to designate a variable as a
pointer. Following are the valid pointer declaration:

```c
int *ip; /* pointer to an integer */
double *dp; /* pointer to a double */
float *fp; /* pointer to a float */
char *ch /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the
same, a long hexadecimal number that represents a memory address. The only difference between
pointers of different data types is the data type of the variable or constant that the pointer points to.

How to use Pointers?

There are few important operations, which we will do with the help of pointers very frequently. **(a)** we
define a pointer variable **(b)** assign the address of a variable to a pointer and **(c)** finally access the
value at the address available in the pointer variable. This is done by using unary operator * that
returns the value of the variable located at the address specified by its operand. Following example
makes use of these operations:

```c
#include <stdio.h>
int main ()
{
int var = 20; /* actual variable declaration */
```

```
int *ip; /* pointer var iable de claratio n */
ip = &var; /* store address of var in pointer variable*/
printf("Address of var variable: %x\n", &var );
/* address stored in pointer variable */ printf("Address
stored in ip variable: %x\n", ip );
/* access the value using the pointer */
printf("Value of *ip variable: %d\n", *ip );
return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

NULL Pointers in C

It is always a good practice to assign a **NULL** value to a pointer variable in case you do not have exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned **NULL** is called a **null** pointer.

The **NULL** pointer is a constant with a value of zero defined in several standard libraries. Consider the following program:

```
#include <stdio.h>
int main ()
{
int *ptr = NULL;
printf("The value of ptr is : %x\n", &ptr );
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
The value of ptr is 0
```

On most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer you can use an if statement as follows:

```
if(ptr) /* succeeds if p is not null */
if(!ptr) /* succeeds if p is null */
```

Pointer arithmetic

As explained in main chapter, C pointer is an address, which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and -

To understand pointer arithmetic, le t us consider that **ptr** is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer:

```
ptr++
```

Now, after the above operation, the **ptr** will point to the location 1004 because each time **ptr** is incremented, it will point to the next integer location which is 4 bytes next to the current location. This operation will move the pointer to next memory location without impacting actual value at the memory location. If **ptr** points to a character whose address is 1000, then above operation will point to the location 1001 because next character will be available at 1001.

Incrementing a Pointer

We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer. The following program increments the variable pointer to access each succeeding element of the array:

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

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

When the above code is compiled and executed, it produces result something as follows:

```
Address of var[0] = bf882b30
Value of var[0] = 10
Address of var[1] = bf882b34
Value of var[1] = 100
Address of var[2] = bf882b38
Value of var[2] = 200
```

Decrementing a Pointer

The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below:

```c
#include <stdio.h>
const int MAX = 3;
int main ()
{
int var[] = {10, 100, 200}; int i,
*ptr;
/* let us have array address in pointer */ ptr =
&var[MAX-1];
for ( i = MAX; i > 0; i--)
{
printf("Address of var[%d] = %x\n", i, ptr );
printf("Value of var[%d] = %d\n", i, *ptr );
/* move to the previous location */ ptr--;
}
return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var[3] = bfedbcd8
Value of var[3] = 200
Address of var[2] = bfedbcd4
Value of var[2] = 100
Address of var[1] = bfedbcd0
Value of var[1] = 10
```

Pointer Comparisons

Pointers may be compared by using relational operators, such as ==, <, and >. If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.

The following program modifies the previous example one by incrementing the variable pointer so long as the address to which it points is either less than or equal to the address of the last element of the array, which is &var[MAX - 1]:

```c
#include <stdio.h>
const int MAX = 3;
int main ()
{
int var[] = {10, 100, 200}; int i,
*ptr;
/* let us have address of the first element in pointer */ ptr =
var;
```

```
i = 0;
while ( ptr <= &var[MAX - 1] )
{
printf("Address of var[%d] = %x\n", i, ptr );
printf("Value of var[%d] = %d\n", i, *ptr );
/* point to the previous location */ ptr++;
i++;
}
return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var[0] = bfdbcb20
Value of var[0] = 10
Address of var[1] = bfdbcb24
Value of var[1] = 100
Address of var[2] = bfdbcb28
Value of var[2] = 200
```

Array of pointers

Before we understand the concept of **arrays of pointers**, let us consider the following example, which makes use of an array of 3 integers:

```
#include <stdio.h>
const int MAX = 3;
int main ()
{
int var[] = {10, 100, 200}; int i;
for (i = 0; i < MAX; i++)
{
printf("Value of var[%d] = %d\n", i, var[i] );
}
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

There may be a situation when we want to maintain an array, which can store pointers to an int or char or any other data type available. Following is the declaration of an array of pointers to an integer:

```
int *ptr[MAX];
```

This declares ptr as an array of MAX integer pointers. Thus, each element in ptr, now holds a pointer to an int value. Following example makes use of three integers, which will be stored in an array of pointers as follows:

```
#include <stdio.h>
const int MAX = 3;
int main ()
{
int var[] = {10, 100, 200}; int i,
*ptr[MAX];
for ( i = 0; i < MAX; i++)
{
ptr[i] = &var[i]; /* assign the address of integer. */
}
for ( i = 0; i < MAX; i++)
{
printf("Value of var[%d] = %d\n", i, *ptr[i] );
}
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

You can also use an array of pointers to character to store a list of strings as follows:

```c
#include <stdio.h>
const int MAX = 4;
int main ()
{
char *names[] = {
"Zara Ali",
"Hina Ali",
"Nuha Ali",
"Sara Ali",
};
int i = 0;
for ( i = 0; i < MAX; i++)
{
printf("Value of names[%d] = %s\n", i, names[i] );
}
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of names[0] = Zara Ali
Value of names[1] = Hina Ali
Value of names[2] = Nuha Ali
Value of names[3] = Sara Ali
```

Pointer to Pointer

A pointer to a pointer is a form of **multiple indirection**, or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.

A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, following is the declaration to declare a pointer to a pointer of type int:

```c
int **var;
```

When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the example:

```c
#include <stdio.h>
int main ()
{
int var; int *ptr; int
**pptr;
var = 3000;
/* take the address of var */ ptr =
&var;
/* take the address of ptr using address of operator & */ pptr =
&ptr;
/* take the value using pptr */ printf("Value
of var = %d\n", var );
printf("Value available at *ptr = %d\n", *ptr );
printf("Value available at **pptr = %d\n", **pptr);
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of var = 3000
Value available at *ptr = 3000
Value available at **pptr = 3000
```

Passing pointers to functions

C programming language allows you to pass a pointer to a function. To do so, simply declare the function parameter as a pointer type.

Following a simple example where we pass an unsigned long pointer to a function and change the value inside the function which reflects back in the calling function:

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

```c
<time.h>
void getSeconds(unsigned long *par);
int main ()
{
unsigned long sec;
getSeconds( &sec );
/* print the actual value */ printf("Number of
seconds: %ld\n", sec );
return 0;
}
void getSeconds(unsigned long *par)
{
/* get the current number of seconds */ *par =
time( NULL );
return;
}
```

When the above code is compiled and executed, it produces the following result:

```
Number of seconds :1294450468
```

The function, which can accept a pointer, can also accept an array as shown in the following example:

```c
#include <stdio.h>
/* function declaration */
double getAverage(int *arr, int size);
int main ()
{
/* an int array with 5 elements */
int balance[5] = {1000, 2, 3, 17, 50}; double
avg;
/* pass pointer to the array as an argument */ avg =
getAverage( balance, 5 ) ;
/* output the returned value */ printf("Average
value is: %f\n", avg );
return 0;
}
double getAverage(int *arr, int size)
{
int i, sum = 0;
double avg;
for (i = 0; i < size; ++i)
{
sum += arr[i];
}
avg = (double)sum / size;
return avg;
}
```

When the above code is compiled together and executed, it produces the following result:

```
Average value is: 214.40000
```

Return pointer from functions

As we have seen in last chapter how C programming language allows to return an array from a function, similar way C allows you to **return a pointer** from a function. To do so, you would have to declare a function returning a pointer as in the following example:

```c
int * m
.
.
.
}
```

Second point to remember is that, it is not good idea to return the address of a local variable to outside of the function so you would have to define the local variable as static variable.

Now, consider the following function, which will generate 10 random numbers and returns them using an array name which represents a pointer, i.e., address of first array element.

```c
#include <stdio.h> #include
<time.h>
/* function to generate and retrun random numbers. */ int *
```

```
getRandom( )
{
static int r[10]; int i;
/* set the seed */
srand( (unsigned)time( NULL ) ); for ( i =
0; i < 10; ++i)
{
r[i] = rand(); printf("%d\n",
r[i] );
}
return r;
}
/* main function to call above defined function */ int main
()
{
/* a pointer to an int */ int *p;
int i;
p = getRandom();
for ( i = 0; i < 10; i++ )
{
printf("*(p + [%d]) : %d\n", i, *(p + i) );
}
return 0;
}
```

When the above code is compiled together and executed, it produces result something as follows:

```
1523198053
1187214107
1108300978
430494959
1421301276
930971084
123250484
106932140
1604461820
149169022
*(p + [0]) : 1523198053 *(p + [1]) : 1187214107 *(p + [2]) : 1108300978 *(p + [3]) :
430494959 *(p + [4]) : 1421301276 *(p + [5]) : 930971084 *(p + [6]) : 123250484 *(p +
    [7]) : 106932140 *(p + [8]) : 1604461820 *(p + [9]) : 149169022
```

C Structures

C arrays allow you to define type of variables that can hold several data items of the

sam e
kind but structure is another user defined data type available in C programming,
which allows you to combine data items of different kinds.
Structures are used to represent a record, suppose you want to keep track of your
books in a library. You might want to track the following attributes about each book:
• Title

• Author

• Subject

• Book ID

Defining a Structure

To define a structure, you must use the **struct** statement. The **struct statement**
defines a new data type, with more than one member for your program. The format of the
struct statement is this:

```
struct [structure tag]
{
member
definition; member
definition;
...
member definition;
} [one or more structure variables];
```

The structure tag is optional and each member definition is a normal variable
definition, such as int i; or float f; or any other valid variable definition. At the end of the
structure's definition, before the final semicolon, you can specify one or more structure
variables but it is optional. Here is the way you would declare the Book structure:

```
struct Books
{
char title[50];
char author[50];
char subject[100];
int book_id;
} book;
```

Accessing Structure Members

To access any member of a structure, we use the member access operator (.). The
member access operator is coded as a period between the structure variable name and the
structure member that we wish to access. You would use **struct** keyword to define
variables of structure type. Following is the example to explain usage of structure:

```
#include
<stdio.h> #include
<string.h>
struct Books
{
char title[50];
char author[50];
char subject[100];
int book_id;
```

```c
};
int main( )
{
   struct Books Book1; /* Declare Book1 of type Book */
   struct Books Book2; /* Declare Book2 of type Book */
   /* book 1 specification */
   strcpy( Book1.title, "C
   Programming"); strcpy( Book1.author,
   "Nuha Ali");
   strcpy( Book1.subject, "C Programming
   Tutorial"); Book1.book_id = 6495407;
   /* book 2 specification */
   strcpy( Book2.title, "Telecom
   Billing"); strcpy( Book2.author, "Zara
   Ali");
   strcpy( Book2.subject, "Telecom Billing
   Tutorial"); Book2.book_id = 6495700;
   /* print Book1 info */
   printf( "Book 1 title : %s\n", Book1.title);
   printf( "Book 1 author : %s\n", Book1.author);
   printf( "Book 1 subject : %s\n", Book1.subject);
   printf( "Book 1 book_id : %d\n", Book1.book_id);
   /* print Book2 info */
   printf( "Book 2 title : %s\n", Book2.title);
   printf( "Book 2 author : %s\n", Book2.author);
   printf( "Book 2 subject : %s\n", Book2.subject);
   printf( "Book 2 book_id : %d\n", Book2.book_id);
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

Structures as Function Arguments

You can pass a structure as a function argument in very similar way as you pass any other variable or pointer. You would access structure variables in the similar way as you have accessed in the above example:

```c
#include
<stdio.h> #include
<string.h>
struct Books
{
   char title[50];
   char author[50];
   char subject[100];
   int book_id;
};
/* function declaration */
void printBook( struct Books
boo k ); int main( )
{
   struct Books Book1; /* Declare Book1 of type Book */
   struct Books Book2; /* Declare Book2 of type Book */
```

```c
/* book 1 specification */
strcpy( Book1.title, "C
Programming"); strcpy( Book1.author,
"Nuha Ali");
strcpy( Book1.subject, "C Programming
Tutorial"); Book1.book_id = 6495407;
/* book 2 specification */
strcpy( Book2.title, "Telecom
Billing"); strcpy( Book2.author, "Zara
Ali");
strcpy( Book2.subject, "Telecom Billing
Tutorial"); Book2.book_id = 6495700;
/* print Book1
info */ printBook(
Book1 );
/* Print Book2 info */
printBook( Book2 );
return 0;
}
void printBook( struct Books book )
{
printf( "Book title : %s\n", book.title);
printf( "Book author : %s\n", book.author);
printf( "Book subject : %s\n", book.subject);
printf( "Book book_id : %d\n", book.book_id);
}
```

When the above code is compiled and executed, it produces the following result:

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

Pointers to Structures

You can define pointers to structures in very similar way as you define pointer to any other variable as follows:

```c
struct Books *struct_pointer;
```

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the & operator before the structure's name as follows:

```c
struct_pointer = &Book1;
```

To access the members of a structure using a **pointer** to that **structure**, you must use the **->** operator as follows:

```c
struct_pointer->title;
```

Let us re-write above example using structure pointer, hope this will be easy for you to understand the concept:

```c
#include <stdio.h>
#include <string.h>
struct Books
{
char title[50]; char
author[50]; char
subject[100]; int
book_id;
};
/* function declaration */
void printBook( struct Books *book );
```

```c
int main( )
{
struct Books Book1; /* Declare Book1 of type Book */ struct
Books Book2; /* Declare Book2 of type Book */
/* book 1 specification */
strcpy( Book1.title, "C Programming");
strcpy( Book1.author, "Nuha Ali");
strcpy( Book1.subject, "C Programming Tutorial");
Book1.book_id = 6495407;
/* book 2 specification */
strcpy( Book2.title, "Telecom Billing");
strcpy( Book2.author, "Zara Ali");
strcpy( Book2.subject, "Telecom Billing Tutorial");
Book2.book_id = 6495700;
/* print Book1 info by passing address of Book1 */
printBook( &Book1 );
/* print Book2 info by passing address of Book2 */
printBook( &Book2 );
return 0;
}
void printBook( struct Books *book )
{
printf( "Book title : %s\n", book->title); printf(
"Book author : %s\n", book->author); printf( "Book
subject : %s\n", book->subject); printf( "Book
book_id : %d\n", book->book_id);
}
```

When the above code is compiled and executed, it produces the following result:

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

C Unions

Aunion is a special data type available in C that enables you to store different data
types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multi-purpose.

Defining a Union

To define a **union**, you must use the union statement in very similar was as you did while defining structure. The **union** statement defines a new data type, with more than one member for your program. The format of the **union statement** is as follows:

```c
union [union tag]
{
member
definition;
member
```

```
definition;
...
member definition;
} [one or more union variables];
```
The **union tag** is optional and each member definition is a normal variable definition, such as int i; or float f ; or any other valid variable definition. At the end of the union's definition, before the final **semicolon**, you can specify one or more union variables but it is optional. Here is the way you would define a union type named Data which has the three members i, f, and str:

```
union Data
{
in
t i;
fl
oat f;
ch
ar str[20];
}
data;
```

Now, a variable of Data type can store an integer, a floating-point number, or a string of characters. This means that a single variable i.e. same memory location can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.

The memory occupied by a union will be large enough to hold the largest member of the union. For example, in above example Data type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by character string. Following is the example which will display total memory size occupied by the above union:

```
#include
<stdio.h> #include
<string.h>
union Data
{
int
i; float
f;
char str[20];
};
int main( )
{
union Data data;
printf( "Memory size occupied by data : %d\n", sizeof(data));
return 0;
}
```

When the above code is compiled and executed, it produces the following result:
```
Memory size occupied by data : 20
```

Accessing Union Members

To access any member of a union, we use the member access operator **(.)**. The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use **union** keyword to define variables of union type. Following is the example to explain usage of union:

```
#include
<stdio.h> #include
<string.h>
union Data
{
int
i; float
f;
char str[20];
};
int main( )
```

```
{
union Data data;
data.i = 10;
data.f = 220.5;
strcpy( data.str, "C Programming");
printf( "data.i : %d\n",
data.i); printf( "data.f : %f\n",
data.f);
```

TUTORIALS
POI NT
Simply Easy
Learning
Pag
e 101

```
printf( "data.str : %s\n", data.str);
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

Here, we can see that values of **i** and f members of union got corrupted because final value assigned to the variable has occupied the memory location and this is the reason that the value if **str** member is getting printed very well. Now let's look into the same example once again where we will use one variable at a time which is the main purpose of having union:

```
#include
<stdio.h> #include
<string.h>
union Data
{
int
i; float
f;
char str[20];
};
int main( )
{
union Data data;
data.i = 10;
printf( "data.i : %d\n", data.i);
data.f = 220.5;
printf( "data.f : %f\n", data.f);
strcpy( data.str, "C
Programming"); printf( "data.str :
%s\n", data.str);
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
data.i : 10
data.f : 220.500000
data.str : C Programming
```

Here, all the members are getting printed very well because one member is being used at a time.

Bit Fields

Suppose your C program contains a number of **TRUE/FALSE** variables grouped in a

structure called status, as follows:

```
struct
{
unsigned int widthValidated; unsigned
int heightValidated;
} status;
```

This structure requires 8 bytes of memory space but in actual we are going to store either 0 or 1 in each of the variables. The C programming language offers a better way to utilize the memory space in such situation. If you are using such variables inside a structure then you can define the width of a variable which tells the C compiler that you are going to use only those number of bytes. For example, above structure can be re-written as follows:

```
struct
{
unsigned int widthValidated : 1; unsigned
int heightValidated : 1;
} status;
```

Now, the above structure will require 4 bytes of memory space for status variable but only 2 bits will be used to store the values. If you will use up to 32 variables each one with a width of 1 bit , then also status structure will use 4 bytes, but as soon as you will have 33 variables, then it will allocate next slot of the memory and it will start using 64 bytes. Let us check the following example to understand the concept:

```
#include <stdio.h>
#include <string.h>
/* define simple structure */
st ruct
{
unsigned int widthValidated; unsigned
int heightValidated;
} status1;
/* define a structure with bit fields */
struct
{
unsigned int widthValidated : 1;
u nsigned int heightValidated : 1;
} status2;
int main( )
{
printf( "Memory size occupied by status1 : %d\n", sizeof(status1));
printf( "Memory size occupied by status2 : %d\n", sizeof(status2));
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Memory size occupied by status1 : 8
Memory size occupied by status2 : 4
```

Bit Field Declaration

The declaration of a bit-field has the form inside a structure:

```
struct
{
type [member_name] : width ;
};
```

Below the description of variable elements of a bit field:

Elements Description

type

An integer type that determines how the bit-field's value is interpreted. The type may

be int, signed int, unsigned int.

member_na

me

The name of the bit-field.

width

The number of bits in the bit-field. The width must be less than or equal to the bit

width of the specified type.

The variables defined with a predefined width are called bit fields. A bit field can hold more than a single bit for example if you need a variable to store a value from 0 to 7 only then you can define a bit field with a width of 3 bits as follows:

```c
struct
{
unsigned int age : 3; }
Age;
```

The above structure definition instructs C compiler that age variable is going to use only 3 bits to store the value, if you will try to use more than 3 bits then it will not allow you to do so. Let us try the following example:

```c
#include <stdio.h>
#include <string.h>
struct
{
unsigned int age : 3; } Age;
int main( )
{
Age.age = 4;
printf( "Sizeof( Age ) : %d\n", sizeof(Age) );
printf( "Age.age : %d\n", Age.age );
Age.age = 7;
printf( "Age.age : %d\n", Age.age );
Age.age = 8;
printf( "Age.age : %d\n", Age.age );
return 0;
}
```

When the above code is compiled it will compile with warning and when executed, it produces the following result:

```
Sizeof( Age ) : 4
Age.age : 4
Age.age : 7
Age.age : 0
```

Typedef

The C programming language provides a keyword called typedef, which you can use

to give a type a new name. Following is an example to define a term BYTE for one -byte numbers:

```c
typedef unsigned char BYTE;
```

After this type definitions, the identifier BYTE can be used as an abbreviation for the type

unsigned char, for example:

```c
BYTE b1, b2;
```

By convention, uppercase letters are used for these definitions to remind the user that the type name is really a symbolic abbreviation, but you can use lowercase, as follows:

```c
typedef unsigned char byte;
```
You can use typedef to give a name to user defined data type as well. For example you can use typedef with structure to define a new data type and then use that data type to define structure variables directly as follows:
```c
#include <stdio.h>
#include <string.h>
typedef struct Books
{
char title[50];
char author[50]; char
subject[100]; int book_id;
} Book;
int main( )
{
Book book;
strcpy( book.title, "C Programming");
strcpy( book.author, "Nuha Ali");
strcpy( book.subject, "C Programming Tutorial");
```

typedef vs #define

The #define is a C-directive which is also used to define the aliases for various data types similar to typedef but with three differences:

• The **typedef** is limited to giving symbolic names to types only where as **#define** can be used to define alias for values as well, like you can define 1 as ONE etc.

• The **typedef** interpretation is performed by the compiler where as **#define** statements are processed by the pre-processor.

Following is a simplest usage of #define:
```c
#include <stdio.h>
#define TRUE 1 #define
FALSE 0
int main( )
{
printf( "Value of TRUE : %d\n", TRUE); printf(
"Value of FALSE : %d\n", FALSE);
return 0;
}
```
When the above code is compiled and executed, it produces the following result:
```
Value of TRUE : 1
    Value of FALSE : 0
```

Command Line Arguments

I t is possible to pass some values from the command line to your C programs when
they are executed. These values are called command line arguments and many times they are
important for your program specially when you want to control your program from outside instead of
hard coding those values inside the code.

The command line arguments are handled using main() function arguments where argc refers to the
number of arguments passed, and argv[] is a pointer ar ray which points to each argument passed to
the program. Following is a simple example which checks if there is any argument supplied from the
command line and take action accordingly:

```c
#include <stdio.h>
int main( int argc, char *argv[] )
{
if( argc == 2 )
{
printf("The argument supplied is %s\n", argv[1]);
}
else if( argc > 2 )
{
printf("Too many arguments supplied.\n");
}
else
{
printf("One argument expected.\n");
}
}
```

When the above code is compiled and executed with a single argument, it produces the following result.

```
$./a.out testing
The argument supplied is testing
```

When the above code is compiled and executed with a two arguments, it produces the following result.

```
$./a.out testing1 testing2
Too many arguments supplied.
```

When the above code is compiled and executed without passing any argument, it produces the following
result.

```
$./a.out
One argument expected
```

It should be noted that argv[0] holds the name of the program itself and argv[1] is a pointer to the first
command line argument supplied, and *argv[n] is the last argument. If no arguments are supplied, argc
will be one, otherwise and if you pass one argument then argc is set at 2.

You pass all the command line arguments separated by a space, but if argument itself has a space then
you can pass such arguments by putting them inside double quotes "" or single quotes ''. Let us re -write
above example once again where we will print program name and we also pass a command line
argument by putting inside double quotes:

```c
#include <stdio.h>
int main( int argc, char *argv[] )
{
printf("Program name %s\n", argv[0]);
if( argc == 2 )
{
printf("The argument supplied is %s\n", argv[1]);
}
else if( argc > 2 )
{
printf("Too many arguments supplied.\n");
}
else
```

```
{
printf("One argument expected.\n");
}
}
```
When the above code is compiled and executed with a single argument separated by space but inside double quotes, it produces the following result.
```
$./a.out "testing1 testing2"
Progranm name ./a.out
The argument supplied is testing1 testing2
```

Memory Management

This chapter will explain dynamic memory management in C. The C programming language provides several functions for memory allocation and management. These functions can be found in the<stdlib.h> header file.

S.N. Function and Description

void *calloc(int num, int size);

1 This function allocates an array of **num** elements each of w hose size in bytes w ill be **size**.

2 **void free(void *address);**

This function release a block of memory block specified by address.

3

void *malloc(int num);

This function allocates an array of **num** bytes and leave them initialized.

4 **void *realloc(void *address, int newsize);**

This function re-allocates memory extending it upto **newsize**.

Allocating Memory Dynamically

While doing programming, if you are aware about the size of an array, then it is easy and you can define it as an array. For example to store a name of any person, it can go max 100 characters so you can define something as follows:

```
char name[100];
```

But now let us consider a situation where you have no idea about the length of the text you need to store, for example you want to store a detailed description about a topic. Here we need to define a pointer to character without defining how much memory is required and later based on requirement we can allocate memory as shown in the below example:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
char name[100];
char *description;
strcpy(name, "Zara Ali");
/* allocate memory dynamically */ description =
malloc( 200 * sizeof(char) ); if( description == NULL
)
{
fprintf(stderr, "Error - unable to allocate required
memory\n");
}
else
{
strcpy( description, "Zara ali a DPS student in class 10th");
}
printf("Name = %s\n", name ); printf("Description:
```

```
%s\n", description );
}
```
When the above code is compiled and executed, it produces the following result.
```
Name = Zara Ali
Description: Zara ali a DPS student in class 10th
```
Same program can be written using calloc() only thing you need to replace malloc with calloc as follows:
```
calloc(200, sizeof(char));
```
So you have complete control and you can pass any size value while allocating memory unlike arrays where once you defined the size can not be changed.

Resizing and Releasing Memory

When your program comes out, operating system automatically release all the memory allocated by your program but as a good practice when you are not in need of memory anymore then you should release that memory by calling the function free().

Alternatively, you can increase or decrease the size of an allocated memory block by calling the functionrealloc(). Let us check the above program once again and make use of realloc() and free() functions:
```
#include <stdio.h> #include
<stdlib.h> #include
<string.h>
int main()
{
char name[100]; char
*description;
strcpy(name, "Zara Ali");
/* allocate memory dynamically */ description =
malloc( 30 * sizeof(char) ); if( description == NULL
)
{
fprintf(stderr, "Error - unable to allocate required
memory\n");
}
else
{
strcpy( description, "Zara ali a DPS student.");
}
/* suppose you want to store bigger description */ description =
realloc( description, 100 * sizeof(char) ); if( description == NULL
)
{
fprintf(stderr, "Error - unable to allocate required
memory\n");
}
else
{
strcat( description, "She is in class 10th");
}
printf("Name = %s\n", name ); printf("Description:
%s\n", description );
/* release memory using free() function */
free(description);
}
```
When the above code is compiled and executed, it produces the following result.
```
Name = Zara Ali
Description: Zara ali a DPS student.She is in class 10th
```
You can try above example without re-allocating extra memory and strcat() function will give an error due to lack of available memory in description.

Preprocessors

T

he C Preprocessor is not part of the compiler, but is a separate step in the

compilation process. In simplistic terms, a C Preprocessor is just a text substitution tool and they instruct compiler to do required pre-processing before actual compilation. We'll refer to the C Preprocessor as the CPP.

All preprocessor commands begin with a pound symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in first column. Following section lists down all important preprocessor directives:

Directive
Description
#define
Substitutes a preprocessor macro
#include
Inserts a particular header from another file
#undef
Undefines a preprocessor macro
#ifdef
Returns true if this macro is defined
#ifndef
Returns true if this macro is not defined
#if
Tests if a compile time condition is true
#else
The alternative for #if
#elif
#else an #if in one statement
#endif
Ends preprocessor conditional
#error
Prints error message on stderr
#pragma
Issues special commands to the compiler, using a standardized method

Preprocessors Examples

Analyze following examples to understand various directives.

```
#define MAX_ARRAY_LENGTH 20
```

This directive tells the CPP to replace instances of MAX_ARRAY_LENGTH with 20. Use #define for constants to increase readability.

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

These directives tell the CPP to get stdio.h from System Libraries and add the text to the current source file. The next line tells CPP to get myheader.h from the local directory and add the content to the current source file.

```
#undef FILE_SIZE #define
FILE_SIZE 42
```

This tells the CPP to undefine existing FILE_SIZE and define it as 42.

```
#ifndef MESSAGE
#define MESSAGE "You wish!"
#endif
```

This tells the CPP to define MESSAGE only if MESSAGE isn't already defined.

```
#ifdef DEBUG
/* Your debugging statements here */
#endif
```

This tells the CPP to do the process the statements enclosed if DEBUG is defined. This is useful if you pass the -DDEBUG flag to gcc compiler at the time of compilation. This will define DEBUG, so you can turn debugging on and off on the fly during compila tion.

Predefined Macros

ANSI C defines a number of macros. Although each one is available for your use in programming, the predefined macros should not be directly modified.

Macro Description

__DATE__

The current date as a character literal in "MMM DD YYYY" format

__TIME__ The current time as a character literal in "HH:MM:SS" format

__FILE__ This contains the current filename as a string literal.

__LINE__

This contains the current line number as a decimal constant.

__STDC__ Defined as 1 when the compiler complies with the ANSI standard.

Let's try the following example:

```c
#include <stdio.h>
main()
{
printf("File :%s\n", __FILE__ );
printf("Date :%s\n", __DATE__ );
printf("Time :%s\n", __TIME__ );
printf("Line :%d\n", __LINE__ );
printf("ANSI :%d\n", __STDC__ );
}
```

When the above code in a file test.c is compiled and executed, it produces the following result:

```
File :test.c
Date :Jun 2 2012
Time :03:36:24
Line :8
ANSI :1
```

Preprocessor Operators

The C preprocessor offers following operators to help you in creating macros:

Macro Continuation (\)

A macro usually must be contained on a single line. The macro continuation operator is used to continue a macro that is too long for a single line. For example:

```c
#define message_for(a, b) \
printf(#a " and " #b ": We love you!\n")
```

Stringize (#)

The stringize or number-sign operator (**'#'**), when used within a macro definition, converts a macro parameter into a string constant. This operator may be used only in a macro that has a specified argument or parameter list. For example:

```c
#include <stdio.h>
#define message_for(a, b) \
printf(#a " and " #b ": We love you!\n")
int main(void)
{
message_for(Carole, Debra); return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Carole and Debra: We love you!
```

Token Pasting (##)

The token-pasting operator (**##**) within a macro definition combines two arguments. It permits two separate tokens in the macro definition to be joined into a single token. For example:

```c
#include <stdio.h>
#define tokenpaster(n) printf ("token" #n " = %d", token##n)
int main(void)
{
int token34 = 40;
tokenpaster(34); return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
token34 = 40
```

How it happened, because this example results in the following actual output from the preprocessor:

```
printf ("token34 = %d", token34);
```

This example shows the concatenation of **token##n** into token34 and here we have used both stringize and token-pasting.

The defined() Operator

The preprocessor **defined** operator is used in constant expressions to determine if an identifier is defined using **#define**. If the specified identifier is defined, the value is true (non-zero). If the symbol is not defined, the value is false (zero). The **defined** operator is specified as follows:

```c
#include <stdio.h>
#if !defined (MESSAGE)
#define MESSAGE "You wish!" #endif
int main(void)
{
printf("Here is the message: %s\n", MESSAGE); return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Here is the message: You wish!
```

Parameterized Macros

One of the powerful functions of the CPP is the ability to simula te functions using parameterized macros. For example, we might have some code to square a number as follows:

```c
int square(int x) { return x
* x;
}
```

We can rewrite above code using a macro as follows:

```c
#define square(x) ((x) * (x))
```

Macros with arguments must be defined using the **#define** directive before they can be used. The argument list is enclosed in parentheses and must immediately follow the macro name. Spaces are not allowed between and macro name and open parenthesis. For example:

```c
#include <stdio.h>
#define MAX(x,y) ((x) > (y) ? (x) : (y))
int main(void)
{
printf("Max between 20 and 10 is %d\n", MAX(10, 20)); return
0;
}
```

When the above code is compiled and executed, it produces the following result:

```
    Max between 20 and 10 is 20
```