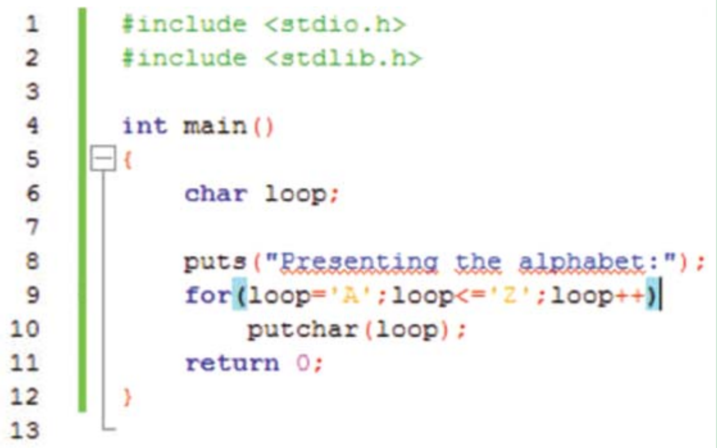
Computer Engineering

9

C Programming



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

int main()
{
    char loop;

    puts("Presenting the alphabet:");
    for(loop='A';loop<='Z';loop++)
        putchar(loop);
    return 0;
}
```

Government of Nepal
Ministry of Education, Science and Technology
Curriculum Development Centre
Sanothimi, Bhaktapur
Phone : 5639122/6634373/6635046/6630088
Website : www.moecdc.gov.np

Feedback Copy

Technical and Vocational Stream
Learning Resource Material

C Programming
(Grade 9)

Secondary Level
Computer Engineering

Government of Nepal
Ministry of Education, Science and Technology
Curriculum Development Centre
Sanothimi, Bhaktapur

Publisher : Government of Nepal

Ministry of Education, Science and Technology

Curriculum Development Centre

Sanothimi, Bhaktapur

© Publisher

Layout by Khados Sunuwar

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any other form or by any means for commercial purpose without the prior permission in writing of Curriculum Development Centre.

Preface

The curriculum and curricular materials have been developed and revised on a regular basis with the aim of making education objective-oriented, practical, relevant and job oriented. It is necessary to instill the feelings of nationalism, national integrity and democratic spirit in students and equip them with morality, discipline and self-reliance, creativity and thoughtfulness. It is essential to develop in them the linguistic and mathematical skills, knowledge of science, information and communication technology, environment, health and population and life skills. it is also necessary to bring in them the feeling of preserving and promoting arts and aesthetics, humanistic norms, values and ideals. It has become the need of the present time to make them aware of respect for ethnicity, gender, disabilities, languages, religions, cultures, regional diversity, human rights and social values so as to make them capable of playing the role of responsible citizens with applied technical and vocational knowledge and skills. This Learning Resource Material for Computer Engineering has been developed in line with the Secondary Level Computer Engineering Curriculum with an aim to facilitate the students in their study and learning on the subject by incorporating the recommendations and feedback obtained from various schools, workshops and seminars, interaction programs attended by teachers, students and parents.

In bringing out the learning resource material in this form, the contribution of the Director General of CDC Dr. Lekhnath Poudel, Pro, Dr. Subarna Shakya, Bibha Sthapit, Kumar Prasun, Dr. Romakanta Pandey, Rajendra Rokaya, Trimandir Prajapati, Satyaram Suwal is highly acknowledged. The book is written by Yogesh Parajuli and the subject matter of the book was edited by Badrinath Timalsina and Khilanath Dhamala. CDC extends sincere thanks to all those who have contributed in developing this book in this form.

This book is a supplimentary learning resource material for students and teachrs. In addition they have to make use of other relevnt materials to ensure all the learning outcomes set in the curriculum. The teachers, students and all other stakeholders are expected to make constructive comments and suggestions to make it a more useful learning resource material.

2076 BS

Ministry of Education, Science and Technology
Curriculum Development Centre

Content

Unit 1 : Programming Languages **1**

1.1 Definition 1

1.2 History/Generation 1

1.3 Classification 2

1.4. Structure oriented language 4

1.5 Compiler Interpreter and Assembler 5

1.6 List of high-level programming languages 7

1.7 Difference between program and software 7

1.8 Concept of programming statement 8

1.9 Syntax and semantics errors 8

1.10 Program control structures 9

1.11 Program Design Tools 9

1.12 Algorithm 10

1.13 Flowchart 10

1.14 Pseudocode 11

Unit 2 : C-Fundamentals **13**

2.1 History of C 13

2.2 Basic Structure of C programming 14

2.3 Character sets, constants and variables 15

2.4 Keywords 16

2.5 Data types 17

2.6 Escape Sequences 20

2.7 Operators 20

Unit 3 : Input/output Statements **25**

3.1 Header file 25

3.2 Formatted input/output 25

3.3 Character Input/output 26

1.4 Program using input/output 27

Unit 4 : Control Statements **28**
4.1 Selective structures 28
4.1.1 if, if else and if else ladder 28
4.1.2 Switch and goto statement 31
4.2. Repetitive structure 34
4.2.1 While loop 34
4.2.2 Do….While Loop 36
4.2.3 For Loop 38
4.3 Nested loop 40
4.4 Break and continue statements 42
Unit 5 : Function **45**
5.1 Definition 45
5.2 Function Types 45
5.3 Programming examples of user defined functions: 46
5.4 Function call by value and call by reference 47
5.5 Return type and Non return type functions 50
5.6 Function prototyping 56
Unit 6 : Arrays and String **57**
6.1 Definition 57
6.2 Array Types 57
6.3 Single Dimensional Arrays 57
6.4 Multidimensional Arrays 60
Unit 7 : Structures **62**
7.1 Definition 62
7.2 Structure variable declaration 62
7.3 Accessing members of a structure 64
Unit 8 : Pointers in c **67**
8.1 Pointer Definition 67
8.2 Declaration of pointer variable 67
8.3 Reference operator (&) 69
8.4 Dereference operator (*) 69

Unit: 1
Programming Languages

1.1 Definition

A programming language is the set of instructions through which humans interact with computers.

These languages are used to give instructions and commands to the computer is known as programming language. E.g. C, C++, Java etc.

1.2 History/Generation

There are five generations of Programming Languages. They are:

1. First Generation Languages (1GL)

1GL uses machine code to write instruction to interact with computer. Machine code is the binary code (0 and 1) understood by computer. It is also called Machine Level Language.

2. Second Generation Languages(2GL)

2GL uses assembly language to write a program. Assembly language is made up of symbolic instructions and addresses, for e.g. ADD A. It is also called Assembly Level Language.

3. Third Generation Languages

3GL uses structured programming language to write a program. Structured programming specifies a logical structure on the program being written to make it more efficient and easier to understand and modify. Examples of 3GL are Fortran, Basic, Turbo Pascal, C, C++.

4. Fourth Generation Languages (4GL)

4GL uses Non-Procedural programming language. Non procedural languages focus on what users want to do rather than how they will be doing it. It consists of statements that are similar to statements in the human language. These are used mainly in database programming and scripting. Example of these languages include, SQL, MATLAB.

5. Fifth Generation Languages (5GL)

5GL uses Artificial Intelligence. Artificial Intelligence languages make the computer appear to communicate like a human being. Examples of fifth generation language include Mercury, OPS5, and Prolog.

1.3 Classification

Programming languages can be classified as:

1. Machine level language
2. Assembly level language
3. Structure oriented language/ procedure-oriented language
4. Object oriented language/problem oriented

Out of these language generation C belongs to the third generation so it is called middle level language as well as high level language.

Machine Level Language

The first-generation programming languages are known as machine language or low-level language. The machine level language deals with only two digits 0 and 1 which are also known as binary digits. As computer can only understand binary i.e. 0 and 1 these language deals with the level of abstraction.

The advantages of this languages are as follows:

1. They are translator free and can be directly executed by computer.
2. The execution speed of these program is very high.
3. These languages are efficient.
4. Memory utilization is efficient and possible to keep track of each bit of data.

The disadvantages of this language are as follows:

1. It is very difficult to develop programs
2. The programmer must have detail knowledge of hardware configuration
3. The programs are very difficult to understand by human beings
4. It is difficult to debug the programs.
5. The programs developed in this language are hardware dependent i.e. the program developed on one machine cannot be executed in another machine.

Assembly Level Language

Assembly language is one level above than machine language. The second-generation languages is known as assembly language. This language uses the concept of mnemonics and symbols for writing the program. The program writing was made easier by the development of assembly language.

The advantages of assembly language are as follows

1. It is easy to develop, understand and modify the programs developed in assembly language than machine level language.
2. Less errors are generated in this language.
3. They are more standardized and easier to use than machine languages.

The disadvantages of assembly language are as follows

1. It is machine dependent language.
2. The programmer needs to have detail knowledge of hardware architecture.
3. This language needs additional language translator to translate the program into machine code
4. Assembly language program are still complex.

High Level Language

High level language is the modern language that uses formats that are similar to English. The purpose of developing high level languages was to enable people to write programs easily in their own native language environment. The generation of languages that belong to high level language are as follows:

1. Third generation language (structure oriented/ procedure oriented)
2. Fourth generation language (problem oriented/ object oriented)
3. Fifth generation language (Natural Language processing)

Advantage of high-level language

1. Language are user friendly
2. Easier to learn
3. Easier to maintain
4. They are problem oriented rather than machine based

Disadvantages of high-level language

1. It takes additional translation time to translate the source to machine code.
2. High level programs are comparatively slower than low level programs. Easier to maintain
3. Compared to low level programs, they are generally less memory efficient.
4. Cannot communicate directly with the hardware.

4. Structure oriented language

These languages belong to the third-generation language. These programs focus on the logic of the program rather than the hardware architecture. These languages are machine independent and the program developed on one computer can be used in another computer. E.g. of these languages are C, COBOL, ALGOL etc.

The advantages of these languages are as follows

1. It is easy to learn, develop and understand the problems.
2. The programs developed on this language can be executed on other computers.
3. Program are easy to code and debug
4. Program written are less prone to error.

The disadvantages of these language are as follows

1. The program execution is slow
2. The program needs to be converted to machine code so additional language translator is required
3. Uses computer resources less efficiently
4. The memory requirement of the program written will be more compared to 1GL and 2GL

Problem Oriented Language/ Object Oriented Language

This language is also termed as fourth generation language. Most of the languages are general purpose programming languages and can be used to develop any kinds of software for any platform. These programs developed in this language are based on real world scenario.

The advantages of these languages are as follows

1. These languages are easy to learn and use
2. The debugging and coding are made easier
3. It is more user friendly.
4. The programs developed in these languages are machine independent.

The disadvantages of these languages are as follows

1. The execution speed of the program is slower
2. It needs additional software to convert the program to machine code
3. The memory requirement of the program is high compared to other programming languages

1.5 Compiler Interpreter and Assembler

Compiler Interpreter and Assembler falls under the category of language translator.

Language Translators

A program written in high-level language is called as source code. To convert the source code into machine code, translators are needed.

A translator takes a program written in source language as input and converts it into a program in target language as output.

It also detects and reports the error during translation.

Roles of translator are

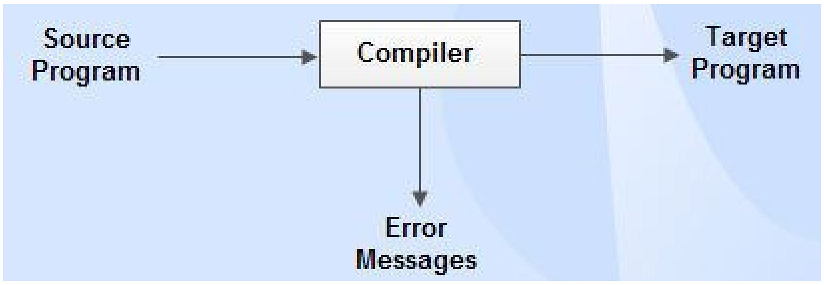
- Translating the high-level language program input into an equivalent machine language program.
- Providing diagnostic or error messages wherever the programmer violates specification or rules of the high-level language program.

Different type of translators

The different types of translator are as follows:

Compiler

Compiler is a translator which is used to convert programs in high-level language to low-level language. It translates the entire program and also reports the errors in source program encountered during the translation.

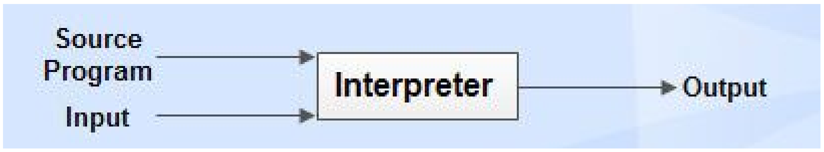


Interpreter

Interpreter is a translator which is used to convert programs in high-level language to low-level language. Interpreter translates line by line and reports the error once it encountered during the translation process.

It directly executes the operations specified in the source program when the input is given by the user.

It gives better error diagnostics than a compiler.



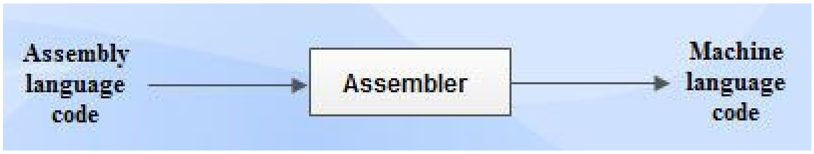
Differences between compiler and interpreter

S.N.	**Compiler**	**Interpreter**
1	Performs the translation of a program as a whole.	Performs statement by statement translation.
2	Execution is faster.	Execution is slower.
3	Requires more memory as linking is needed for the generated intermediate object code.	Memory usage is efficient as no intermediate object code is generated.
4	Debugging is hard as the error messages are generated after scanning the entire program only.	It stops translation when the first error is met. Hence, debugging is easy.

5	Programming languages like C, C++ uses compilers.	Programming languages like Python, BASIC, and Ruby uses interpreters.

Assembler

Assembler is a translator which is used to translate the assembly language code into machine language code.



1.6 List of high-level programming languages

High level language are the languages which are easy to understand and can be written in English language. The development of program is easy using this language. The example of high-level language are as follows:

- C
- C++
- JAVA
- PHP
- BASIC
- .NET

1.7 Difference between program and software

The difference between program and software is as follows

Program	Software
A program is a set of instructions written in a programming language to perform a particular task.	A software is a collection of programs to perform a multiple task.
A program only consists of coding	A software consists of coding, graphics, documentation and manuals
A program can be simply developed by	Multiple programmers work for a

a programmer	certain time to develop a software
A program is rarely used for business purpose	A software is mostly used for business purpose
E.g. A program to add two numbers	E.g. Microsoft word, calculator etc.

1.8 Concept of programming statement

A programming statement is a command that the programmer gives to the computer using a programming language

For : e.g.

```
printf("hello world");
```

The command instructs the computer to print hello world. This is a programming statement. There are many programming statements in c such as scanf(), fscanf() which will be addressed in upcoming chapters.

1.9 Syntax and semantics errors

A syntax error is like a grammatical error or an error in spelling where you made a new, unrecognizable word. So, basically you made a program the computer can't read or understand somewhere. Program doesn't execute when it has these types of errors. So,these kinds of errors are easily traceable and can be corrected by beginners of a program.

For e.g.: int a = 5 // semicolon is missing

So, compiler cannot process from here as it requires semicolon so it shows error.

A sematic error is the error in the programming logic which computer can understand but the program gives wrong output. The computer executes the program but the output is wrong. These kinds of error are complex in nature and only expert programmers are able to trace it in a huge program.

For e.g. c = a + b; for this statement we mistakenly placed c = a-b; here the error is in the logic of the program so program is right but the output produced by program is wrong.

1.10 Program control structures

Program control structures defines the flow of program. They define the way the program logic to flow in the program. There are three program control structures they are:

- Sequence
- Selection
- Iteration

Sequence: In this control structure the flow of program flows in sequential manner. The logic defines the program to flow on the line by line basis. So in these control structures the length of the program will be high and program becomes complex due to same repetition of code. This will be addressed in upcoming chapters

Selection: In this control structure the flow of program flows on the basis of selection. The program has the capability to directly jump to the function selected by user. The length of the program is reduced by the use of selection techniques in a program. The most frequently used selection procedures in C programming are switch, goto, and label.

Iteration: In this control structure the flow of a program is designed in such a way the program continuously operates until the condition given by the user is satisfied. The program keeps updating its input and output automatically as defined by the user. These techniquesreduce the length of a program and it is very efficient. The iteration is also termed as looping. The loops available in C programming are for, while, do…while.

1.11 Program Design Tools

The program design tools are the tools required to develop a program. During designing the program various tools are required to design the program such as

- Algorithm
- Flowchart
- Pseudocode

1.12 Algorithm

An algorithm is a set of steps of operations to solve a problem performing calculation, data processing, and automated reasoning tasks. An algorithm is an efficient method that can be expressed within finite amount of time and space.

An algorithm is the best way to represent the solution of a particular problem in a very simple and efficient way. If we have an algorithm for a specific problem, then we can implement it in any programming language, meaning that the **algorithm is independent from any programming languages**.

The main characteristics of algorithms are as follows −

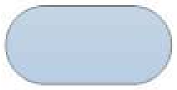
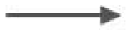
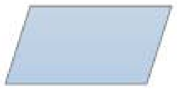
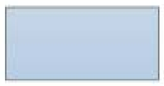
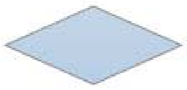
- Algorithms must have a unique name
- Algorithms should have explicitly defined set of inputs and outputs
- Algorithms are well-ordered with unambiguous operations
- Algorithms halt in a finite amount of time. Algorithms should not run for infinity, i.e., an algorithm must end at some point

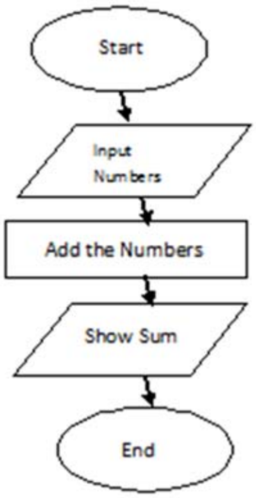
Algorithm for adding two numbers

```
Step 1:  start
Step2: input two numbers x and y
Step3: read two numbers x and y
Step4: z=x+y
Step5:  display result z
Step6: stop
```

1.13 Flowchart

A flowchart is simply a graphical representation of steps. It shows steps in a sequential order, and is widely used in presenting flow of algorithms, workflow or processes. Typically, flowchart shows the steps as boxes of various kinds, and their order by connecting them with arrows.

Symbol	Name	Function
	Start/end	An oval represents a start or end point
	Arrows	A line is a connector that shows relationships between the representative shapes
	Input/Output	A parallelogram represents input or output
	Process	A rectangle represents a process
	Decision	A diamond indicates a decision



1.14 Pseudocode

Pseudocode gives a high-level description of an algorithm without the ambiguity associated with plain text but also without the need to know the syntax of a particular programming language.

The running time can be estimated in a more general manner by using Pseudocode to represent the algorithm as a set of fundamental operations which can then be counted.

Pseudocode for finding the area of rectangle:

```
Input length breadth
Calculate area= length* breadth
Output: area
```

Unit: 2

C-Fundamentals

2.1 History of C

An ancestor of C is Basic Combined Programming Language (BCPL). Ken Thompson, a Bell Laboratory scientist, developed a version of BCPL, which he called ‘B’. Dennis Ritchie, another computer scientist, developed a version called ‘C’ in early 1970s that is modified and improved BCPL. Ritchie originally wrote the language for programming under UNIX operating system. Later Bell Laboratories rewrote UNIX entirely in C.

C belongs to the third-generation programming language and it is also termed as high level or middle level language. Out of five generations it lies in third generation.

C is a structured programming language, which means that it allows you to develop programs using well-defined control structures (you will learn about control structures in the articles to come), and provides modularity (breaking the task into multiple sub tasks that are simple enough to understand and to reuse). C is often called a **middle-level language** because it combines the best elements of low-level or machine language with high-level languages. It is one of the most popular programming languages till date due to its speed and flexibility.

Characteristics/features

Some of C's characteristics that define the language and also have led to its popularityasa programming language. This course has included these aspects :

Portability

One of the reasons of C’s popularity is its portability. We can easily transform a program written in C from one computer to another with few or no changes and compile with appropriate compilers.

Faster and efficient

C is desirable because it is faster and more efficient than comparable programs in most other high-level languages. For e.g., a program to increment a variable from

0 to 15000 takes about 50 seconds in BASIC while it takes 1 second in C.

Supports structuredprogramming

It is well suited for structured programming that means, the problem might be solved in terms of function modules or blocks. The modular structure makes program debugging, testing and maintenance easier.

Extendibility

Another property is extendibility. C is basically a collection of functions that are supported by C library. We can continuously add our own functions to C library.

Flexible

C is a flexible language. It permits us to write any complex programs with the help of its rich set of in-built functions and operators. In addition, it permits the use of low-level language. Hence it is also called “middle-level language” and therefore it is well suited for both system software and business package.

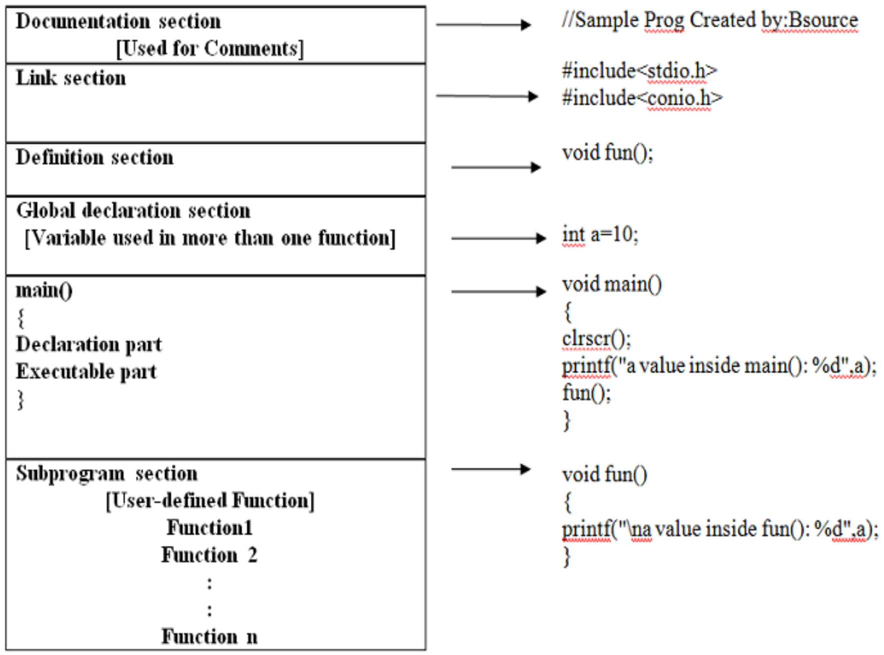
2.2 Basic Structure of C programming

The basic structure of a C program comprises the following sections:

- Documentation section
- Preprocessor directive section
- Global declaration section
- Main program section
- Sub program section

The program to demonstrate all those sections as follows :

SIC STRUCTURE OF A 'C' PROGRAM: Example:



```
//Sample Prog Created by:Bsource

#include<stdio.h>
#include<conio.h>

void fun();

int a=10;

void main()
{
clrscr();
printf("a value inside main(): %d",a);
fun();
}

void fun()
{
printf("\na value inside fun(): %d",a);
}
```

2.3 Character sets, constants and variables

The C character sets are used to form words, numbers and expressions. Different categories of character sets are – letters, digits and special characters.

Letters – upper case: A…Z

Lower case: a…z

Digits – 0…9

Special characters: ' " , . : ; { } [ ] ! # $ & ^ ( ) _ - + * <> ? / \ | !

White spaces: blank, tab, newline

Constants are fixed values that cannot be altered by the program and can be numbers, characters or strings.

Some Examples: -

char: 'a', '$', '7'

int: 10, 100, -100

unsigned: 0, 255

float: 12.23456, -1.573765e10, 1.347654E-13

double: 1433.34534545454, 1.35456456456456E-200

long: 65536, 2222222

string: “Hello World\n”

A **variable** is a named piece of memory which is used to hold a value which may be modified by the program. A variable thus has three attributes that are of interest to us: its **type,** its **value** and its **address.**

The variable’s type informs us what type and range of values it can represent and how much memory is used to store that value. The variable’s address informs us where in memory the variable is located (which will become increasingly important when we discuss pointers later on).

All C variables must be declared as follows:-

Datatype variable-list;

For Example:-

inti ;

char a, b, ch ;

2.4 Keywords

Keywords are the reserved words having fixed meaning in C. They cannot be used as identifiers. C makes use of only 32 keywords which combine with the formal syntax to the form the C programming language.

Note that all keywords are written in lower case – C uses upper and lowercase text to mean different things. If you are not sure what to use then always use lowercase text in writing your C programs. A keyword may not be used for any other purposes. For example, you cannot have a variable called **auto**. If we try to use keyword as variable name, then we’ll be trying to assign new meaning to the keyword, which is not allowed by thecomputer.

The following are reserved keywords, and may not be used as identifiers:

Auto, double,int, struct,break, else, long, switch, case, enum, register

typedef, char, extern, return, union, const, float, short, for, do, if, continue signed, void, default, goto, sizeof, while, volatile, unsigned, static

2.5 Data types

The first thing we need to know is that we can create variables to store values in. A variable is just a named area of storage that can hold a single value (numeric or character). C is very fussy about how you create variables and what you store in them. It demands that you declare the name of each variable that you are going to use and its type, before you actually try to do anything with it. Hence, data type can be defined as the storage representation and machine instruction to handle constants andvariables.

There are four basic data types associated with variables:

- Primary datatype
- Derived datatype

Primary/fundamental data type

The data type that is used without any modifier is known as primary data type. The primary data type can be categorized as follows:

1. **Integertype**
   a. signed integertype
      - int
      - shortint
      - longint
   b. unsigned integer type
      - unsigned int
      - unsigned shortint
      - unsigned longint
2. **Charactertype**
   a. Signedchar
   b. Unsignedchar

3. **Floating pointtype**

    a. float

    b. double

    c. longdouble

- int - integer: a whole number. We can think of it as a positive or negative wholenumber. But no fractional part is allowed

  To declare an **int**you use the instruction:

  int variable name;

  For example:

  int a;

- declares that you want to create an **int**variable called **a**.
- float - floating point or real value, i.e., a number with a fractionalpart.
- double - a double-precision floating point value.
- char - a single character.

  To declare a variable of type character we use the keyword **char**. - A single character stored in one byte.

For example:

  char c;

To assign, or store, a character value in a **char** data type is easy - a character variable is just a symbol enclosed by single quotes. For example, if **c** is a **char** variable you can store the letter **A** in it using the following C statement:

  c='A'

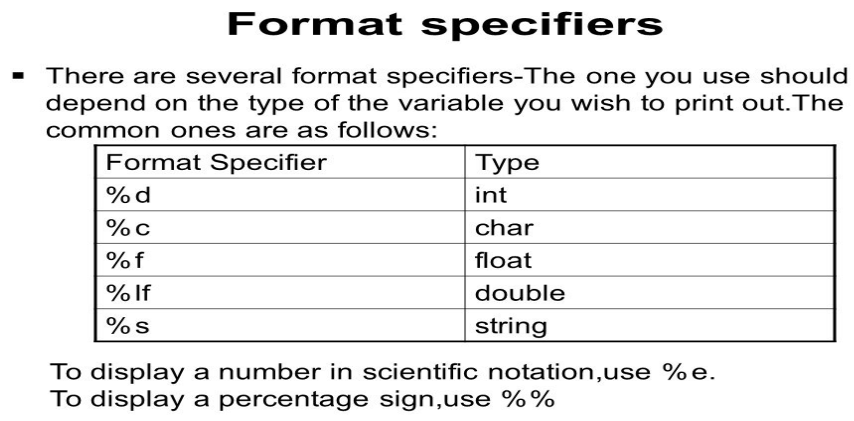
Notice that you can only store a single character in a **char** variable. But a string constant is written between double quotes. Remember that a **char** variable is **'A'** and not **"A"**.

Here is the list of data types with their corresponding required memory range of value it can take:

DATA TYPE	MEMORY (BYTES)	RANGE	FORMAT SPECIFIER
short int	2	-32,768 to 32,767	%hd
unsigned short int	2	0 to 65,535	%hu
unsigned int	4	0 to 4,294,967,295	%u
int	4	-2,147,483,648 to 2,147,483,647	%d
long int	4	-2,147,483,648 to 2,147,483,647	%ld
unsigned long int	4	0 to 4,294,967,295	%lu
long longint	8	-(2^63) to (2^63)-1	%lld
unsigned long longint	8	0 to 18,446,744,073,709,551,615	%llu
signed char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c
float	4		%f
double	8		%lf
long double	12		%Lf

Format specifiers in c

The symbols that are used to represent the type/format of the value being read and processed is known as format specifier.



Format specifiers

- There are several format specifiers-The one you use should depend on the type of the variable you wish to print out.The common ones are as follows:

Format Specifier	Type
%d	int
%c	char
%f	float
%lf	double
%s	string

To display a number in scientific notation,use %e.
To display a percentage sign,use %%

Derived data type

Different user defined data type can be created using fundamental data types which are called derived data type.

Array, structure, union and functions are derived data types.

2.6 Escape Sequences

An escape sequence in C language is a sequence of characters that doesn't represent itself when used inside string literal or character.

It is composed of two or more characters starting with backslash \. For example: \n represents new line.

2.7 Operators

C provides rich set of operator environment. Operators are symbols that tell the computer to perform certain mathematical and logical manipulations. They are used to form a part of mathematical and logical expressions. The variable/quantity on which operation is to be performed is called operand. The operators can be categorizedas:

1. Arithmeticoperators
2. Relationaloperators
3. Logical operators
4. Conditionaloperators
5. Unaryoperators
6. Assignmentoperators
7. Special operators

Arithmetic operator

These are the operators used to perform arithmetic/mathematical operations on operands.

Examples: (+, -, *, /, %,++,–).

Arithmetic operator are of two types:

1. **Unary Operators**: Operators that operates or works with a single operand

are unary operators.

For example: (++ , –)

2. **Binary Operators**: Operators that operates or works with two operands are binary operators.

   For example: (+ , – , * , /)

The following table shows all the arithmetic operators supported by the C language.

Assume variable **A** holds 10 and variable **B** holds 20 then

Operator	Description	Example
+	Adds two operands.	A + B = 30
−	Subtracts second operand from the first.	A − B = -10
*	Multiplies both operands.	A * B = 200
/	Divides numerator by de-numerator.	B / A = 2
%	Modulus Operator and remainder of after an integer division.	B % A = 0

Relational operator

Relational operators are used for comparison of the values of two operands. For example: checking if one

Operand is equal to the other operand or not, an operand is greater than the other operand or not etc.

Some of the relational operators are (==, >=, <=).

he following table shows all the relational operators supported by C. Assume variable **A**

holds 10 and variable **B** holds 20 then

Operator	**Description**	**Example**
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	(A == B) is not true.

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

Logical operator

Logical Operators are used to combine two or more conditions/constraints or to complement the evaluation of the original condition in consideration. The result of the operation of a logical operatoris a Boolean value either true or false.

Following table shows all the logical operators supported by C language. Assume variable **A**

holds 1 and variable **B** holds 0, then

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A || B) is true.

!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

Assignment operator

Assignment operators are used to assign value to a variable. The left side operand of the assignment operator is a variable and right side operand of the assignment operator is a value. The value on the right side must be of the same data-type of variable On

The left side otherwise the compiler will raise an error.

Different types of assignment operators are shown below:

- **"="**: This is the simplest assignment operator. This operator is used to assign the value on the right to the variable on the left.
  For example:

  a = 10;

  b = 20;

  ch = 'y';

- **"+="**: This operator is combination of '+' and '=' operators. This operator first adds the current value of the variable on left to the value on right and then assigns the result to the variable on the left.

Example:

(a += b) can be written as (a = a + b)

If initially value stored in a is 5. Then (a += 6) = 11.

- **"-=":** This operator is combination of '-' and '=' operators. This operator first subtracts the current value of the variable on left from the value on right and then assigns the result to the variable on the left.

Example:

(a -= b) can be written as (a = a – b)

If initially value stored in a is 8. Then (a -= 6) = 2.

- **“*=”:** This operator is combination of ‘*’ and ‘=’ operators. This operator first multiplies the current value of the variable on left to the value on right and then assigns the result to the variable on the left. Example:

  (a *= b) can be written as (a = a * b)

  If initially value stored in a is 5. Then (a *= 6) = 30.

- **“/=”:** This operator is combination of ‘/’ and ‘=’ operators. This operator first divides the current value of the variable on left by the value on right and then assigns the result to the variable on the left.

Example:

(a /= b) can be written as (a = a / b)

If initially value stored in a is 6. Then (a /= 2) = 3.

Unit: 3

Input/output Statements

3.1 Header file

Header file defines certain values, symbols and operations which is included in the file Header to obtain access to its contents. The header file has the suffix '.h'. It contains only the prototype of the function in the corresponding source file. For e.g., stdio.h contains the prototype for 'printf' while the corresponding source file contains its definition. It saves time for writing and debugging the code.

Example

stdio.h It will have all the information about i/o related functions definition. If our c program needs to say something to the user or get some input from the user, we should use printf and scanf function accordingly.

whenever we need to use input and output related (ex. printf/scanf) functions in our program, we should include stdio.h header file.

Likewise, math.h - will have all mathematical functions' definition. Usingmath.h header file we can calculate, the square root of a number, power(x, y) i.e. x^y etc.

3.2 Formatted input/output

- C provides standard functions scanf() and printf(), for performing formatted input and output .These functions accept, as parameters, a format specification string and a list of variables.
- printf is used to print the contents that are inside the parenthesis of the printf statement
- Scanf is used to read and store the input given by a user to a variable. For scanf variable is necessary to store the information given by the user

The simple program to add two numbers is as follows:

```
#include<stdio.h>

Voidmain ()

{
```

```
int a,b,c;
printf(“enter values for a and b”);
scanf(“%d %d”,&a,&b);
C=a+b;
Printf(“the sum is %d”,c);
}
```

Here we can see how the printf and scanf function are used in a program

3.3 Character Input/output

SINGLE CHARACTER INPUT-- THE getchar FUNCTION

Single characters can be entered into the computer using the C library function getchar. The function does not require any arguments, though a pair of empty parentheses must follow the word getchar.

```
Syntax: character variable= getchar();
Usage char ch;
Ch=getchar();
```

SINGLE CHARACTER OUTPUT-- THE putchar FUNCTION

Single character output can be displayed using the C library function putchar. The character being transmitted will normally be represented as a character- type variable. It must be expressed as an argument to the function, following the word putchar.

```
Syntax: putchar(character variable);
Usage char ch;
………..
putchar(c);
```

Following program accepts a single character from the keyboard and displays it.

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

```
#include<conio.h>
void main()
{
char ch;
clrscr();
printf("\n Enter any character of your choice: -");
ch = getchar();
printf ("\n the character u entered was ");
putchar(ch);
getch();
}
```

1.4 Program using input/output

```
#include<stdio.h>
void main()
{
   // defining a variable
inti;
   /* displaying message on the screen asking the user to input a value    */
printf("Please enter a value...");
   /* reading the value entered by the user*/
scanf("%d", &i);
   /* displaying the number as output    */
printf( "\nYou entered: %d", i);
}
```

Unit: 4

Control Statements

4.1 Selective structures

Selective structures is used in C program to select appropriate output or input from a group of input/output. The various selective structures are

- If
- If else
- If else ladder
- Switch
- Goto
- Label

4.1.1 if, if else and if else ladder

The **if** statement is a conditional branch statement. If the condition is true, then the statement after condition is executed otherwise execution will skip to next statement.

```
Syntax:   if (condition)
                 statement body;
          else
                 statement body;
```

<u>or just:</u>

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

In the first more general form of the statement one of two code blocks are to be executed. If the condition evaluates to TRUE the first statement body is executed otherwise for all other situations the second statement body is executed.

In the second form of the statement the statement body is executed if the condition evaluates to TRUE. No action is taken otherwise.

For example:

```
 #include<stdio.h>
void main()
{
int a;
printf("Enter the value of a:");
scanf("%d", &a);
if(a>=0)
printf("The number is positive");
}
```

The if else Statement

The if…else statement consists of an if statement followed by statement or block of statement, followed by else keyword which is again followed by another statement or block of statement. In an if…else statement, the condition is evaluated first. If the condition is true, the statement in the immediate block is executed. If the condition is false, the statement in the else block is executed. This is used to decide whether to do something at a special point, or to decide between two courses of action.

for e.g.

```
if (result >= 75)
printf("Passed: Grade A\n");
else if (result >= 60)
printf("Passed: Grade B\n");
else if (result >= 45)
printf("Passed: Grade C\n");
```

```
else
printf("Failed\n");
```

if else ladder

Nested ifs are very common in programming. Nested if is a structure which has another if…else body within its body of structure. When you nest ifs, the main thing to remember is that an else statement always refers to the nearest if statement that is within the same block.

```
Syntax:    if (condition_1)
                statement_1;
           else if (condition_2)
                statement_2;
           else if (condition_3)
                statement_3;
           ...
           else if (condition)
                statement;
           else
                statement default;
```

for e.g.

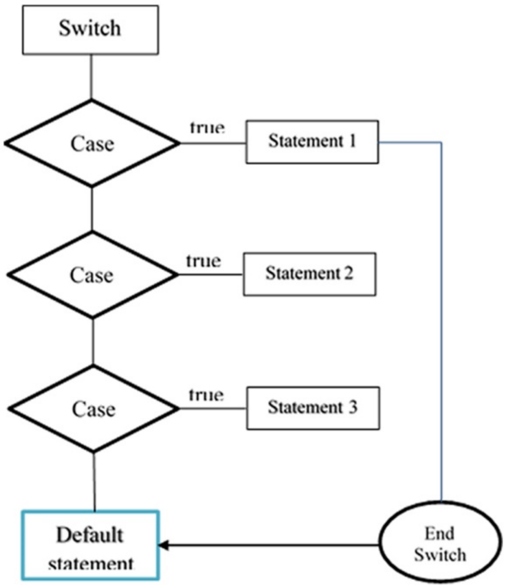
```
if (a>b)
{
If(b>c)
{
If(c>d)
{
Printf (“a is greatest”);
Else
Printf (“b is greatest”);
}
```

4.1.2 Switch and goto statement

This is another form of the multi way decision. It checks the value of an expression to the list of constant values. If the condition is matched, the statement/statements associated with it will be executed. If the expression does not match any of the case statement, and if there is a default statement, execution switches to default statement otherwise the switch statement ends. It is well structured, but can only be used in certain caseswhere;

- Only one variable is tested, all branches must depend on the value of that variable. The variable must be an integral type. (int, long, short orchar).
- You cannot use ranges as an expression i.e. the expression must give as absolutevalue.
- Each possible value of the variable can control a single branch. A final, catch all, default branch may optionally be used to trap all unspecifiedcases.

The flowchart to represent the flow of switch is shown in the figure:



Syntax of switch...case

```
switch (n)

{

case constant1:

        // code to be executed if n is equal to constant1;

        break;

case constant2:

        // code to be executed if n is equal to constant2;

break;

        .

        .

        .

default:

        // code to be executed if n doesn't match any constant

}
```

e.g. of switch statement

```
#include <stdio.h>

int main()

{

int x = 2;

switch (x)

{

case 1: printf ("Choice is 1");

break;

case 2: printf ("Choice is 2");
break;
case 3: printf ("Choice is 3");
break;
default: printf ("Choice other than 1, 2 and 3");
break;
   }
}
```

Goto statement

The goto statement is one of C's branching, or unconditional jump, statements. Whenever the program reaches a goto statement, the execution jumps to the location specified by the goto statement. We call the goto statement is an unconditional statement because whenever the program encounters a goto statement, the execution of the program branches accordingly, which does not depend on any condition like the if statement.

Example to illustrate the use of goto statement in C

```
#include <stdio.h>
void main ()
{
  goto a;
  b:
    printf("gram");
    goto c;
  a:
printf ("C pro");
    goto b;
  c:
printf("ming");
}
```

Output will be c programming

4.2. Repetitive structure

Repetitive structure is also known as loops or iteration statement. In c programming there are three types of loops which are used for iteration purposes. The three types of loop are

- While loop
- Do while loop
- For loop

5.2.1 While loop

A **while** loop in C programming 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 −

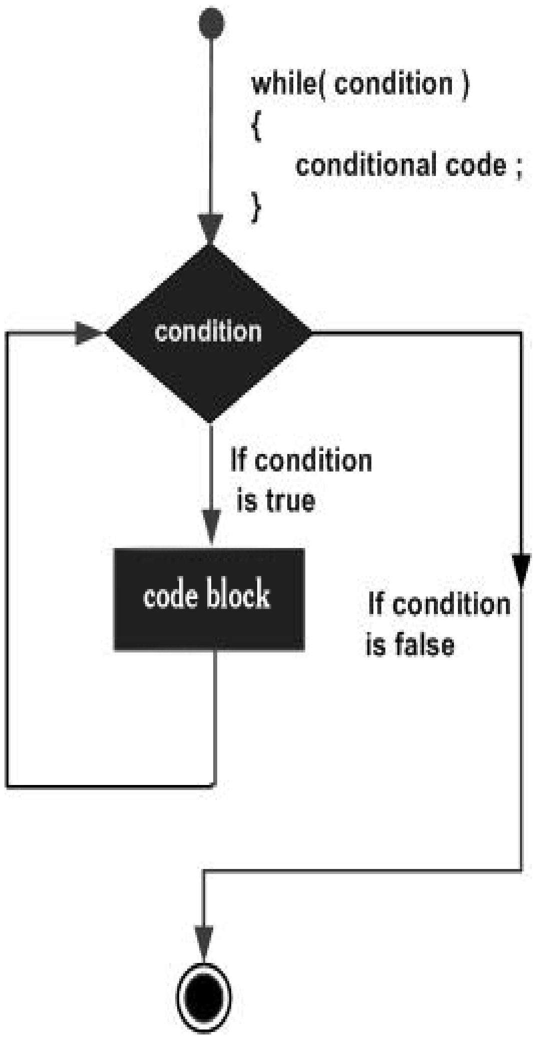
```
while(condition) {
  statement(s);
}
```

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, the program control passes to the line immediately following the loop.

Flow Diagram

Here, the key point to note is that a while loop might not execute at all. 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>
Void main ()
{
  int a = 10;
  /* while loop execution */
while (a<15) {
printf ("value of a: %d\n", a);
```

```
      a++;
   }
}
```

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
```

4.2.2 Do….While Loop

Unlike for and while loops, which test the loop condition at the top of the loop, the do...while loop in C programming checks its condition at the bottom of the loop.

A do...while loop is similar to a while loop, except the fact that it is guaranteed to execute at least one time.

Syntax

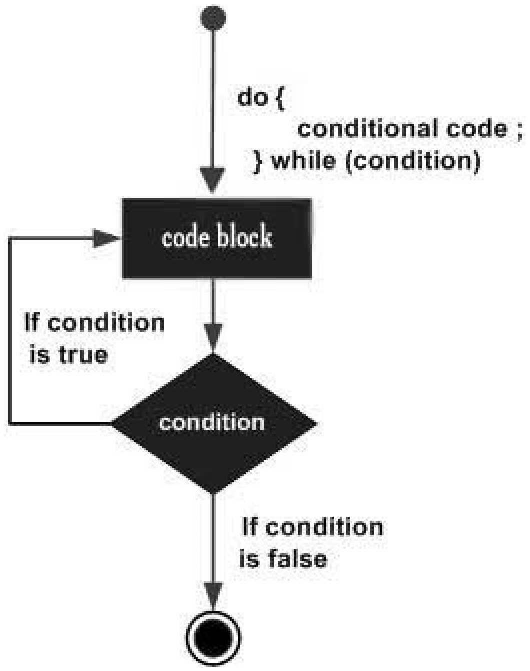
The syntax of a do...while loop in C programming language is −

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

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop executes once before the condition is tested.

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop executes again. This process repeats until the given condition becomes false.

Flow Diagram



Example

```
#include <stdio.h>

int main () {

  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
```

4.2.3 For Loop

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 (initialization; condition; increment) {
   statement(s);
}
```

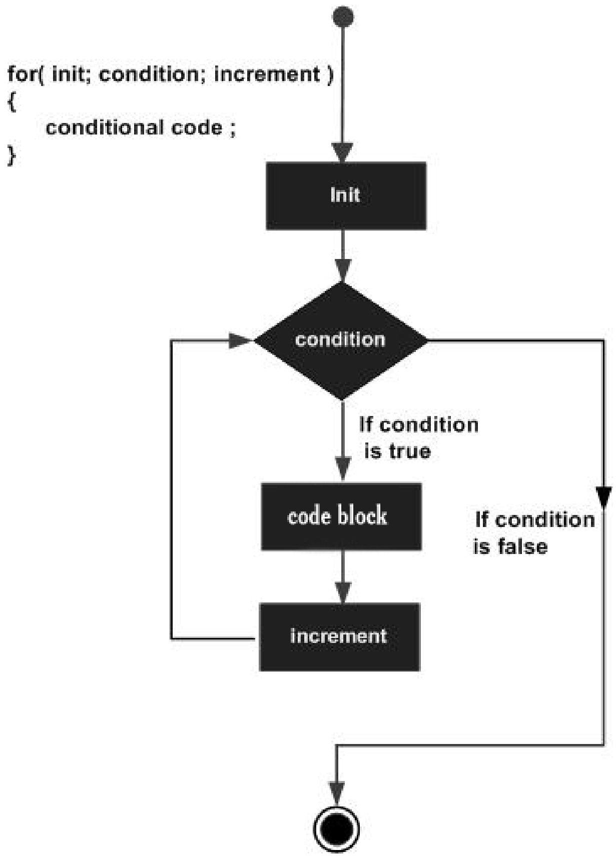
Here is the flow of control in a 'for' loop −

- 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.
- 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 the flow of control jumps to the next statement just after the 'for' loop.
- After the body of the 'for' loop executes, the flow of control jumps back up to the increment statement. This statement allows you to update any loop control

variables. This statement can be left blank, as long as a semicolon appears after the condition.

- 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 ()

{

  int a;

  /* for loop execution */
for (a = 10; a < 20; a = a + 1 ){
printf ("value of a: %d\n", a);
  }
}
```

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
```

4.3 Nested loop

C programming allows to use one loop inside another loop. The following section shows a few examples to illustrate the concept.

Syntax

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

```
for (int; condition; increment) {
for (int; 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 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>
void main ()
{
inti, j;
for(i = 2; i<10; 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);
  }
 }
```

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
```

4.4 Break and continue statements

Break statement

We have already met break in the discussion of the switch statement. It is used to exit from a loop or a switch, control passing to the first statement beyond the loop or a switch.

With loops, break can be used to force an early exit from the loop, or to implement a loop with a test to exit in the middle of the loop body. A break within a loop should always be protected within an if statement which provides the test to control the exit condition.

The following program calculates the sum of entered number only if the value is positive otherwise it exits from the loop. Here the break statement causes to terminate the loop when we enter the negativevalue.

E.g. to demonstrate break statement

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

```
#include<conio.h>
void main()
{
inti, sum = 0, num; clrscr();

for(i= 0; i<=5; i++ )
{
printf("Enter number %d :", i); scanf("%d", &num);
if(num<0)
break;
sum+= num;
}
printf("The sum = %d", sum);
getch();
}
```

Continue statement

This is similar to break but is encountered less frequently. It only works within loops where its effect is to force an immediate jump to the loop control statement.

- In a while loop, jump to the teststatement.
- In a do while loop, jump to the teststatement.
- In a for loop, jump to the test, and perform theiteration.

Like a break, continue should be protected by an if statement. You are unlikely to use it very often.

The following program calculates the sum of entered number only if the value is positive. Unlike the break statement, the continue statement does not terminate the loop when we enter the negative value, but it skips the following statement and continues the loop.

```
#include<stdio.h>
#include<conio.h>
void main()
{
inti, sum = 0, num;
clrscr();
for(i= 0; i<=5; i++)
{
printf("Enter number %d:", i);
scanf("%d", &num);
 if(num<0)
continue;
sum=sum+ num;
}
printf("The sum = %d", sum); getch();
}
```

Unit: 5

Function

5.1 Definition

A function is a group of statements which together perform a specific task together. Every C program has at least one function, which is main(), and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.

A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

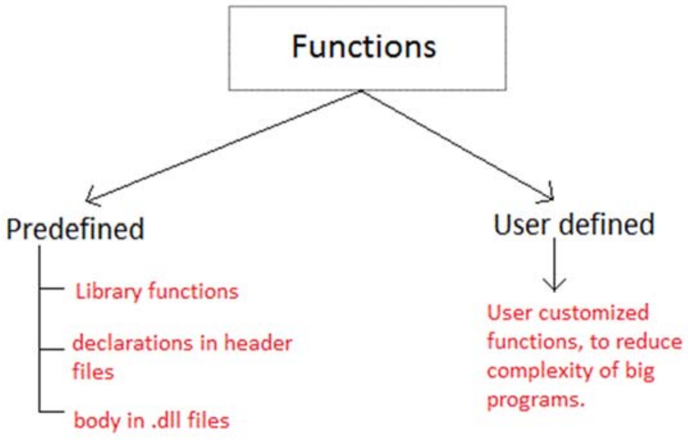
The C standard library provides numerous built-in functions that your program can call. For example, strcat() to concatenate two strings, memcpy() to copy one memory location to another location, and many more functions.

A function can also be referred as a method or a sub-routine or a procedure, etc.

5.2 Function Types

C functions can be classified into two categories,

1. Library functions
2. User-defined functions


Functions
Predefined
Library functions
declarations in header files
body in .dll files
User defined
User customized functions, to reduce complexity of big programs.

Library functions are those functions which are already defined in C library, example `printf()`, `scanf()`, `strcat()` etc. You just need to include appropriate header files to use these functions. These are already declared and defined in C libraries.

A **User-defined functions** on the other hand, are those functions which are defined by the user at the time of writing program. These functions are made for code reusability and for saving time and space.

Benefits of Using Functions

1. It provides modularity to your program's structure.
2. It makes your code reusable. You just have to call the function by its name to use it, wherever required.
3. In case of large programs with thousands of code lines, debugging and editing becomes easier if you use functions.
4. It makes the program more readable and easier to understand.

5.3 Programming examples of user defined functions:

1. Program to add two numbers using function

```
#include<stdio.h>
int sum(int,int);
void main()
{
int num1, num2, res;
printf("\nEnter the two numbers : ");
scanf("%d %d", &num1, &num2);
//Call Function Sum with Two Parameters
res = sum(num1, num2);
printf("nAddition of two number is : %d",num3);
}
```

```
int sum(int num1, int num2)
{
int num3;
num3 = num1 + num2;
return (num3);
}
```

2. Program to find area of rectangle using function

```
#include<stdio.h>
#include<conio.h>
intrect_area(int,int)
void main()
{
intl,b,a;
printf("enter the values for length and breadth");
scanf("%d %d",&l.&b);
a=rect_area(l,b);
printf("the area of rectangle is %d",a);
getch();
}
```

5.4 Function call by value and call by reference

Call by value and Call by reference in C

There are two methods to pass the data into the function in C language, i.e., call by value and call by reference.

Let's understand call by value and call by reference in c language one by one.

Call by value in C

- In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.
- In call by value method, we cannot modify the value of the actual parameter by the formal parameter.
- In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.
- The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

Let's try to understand the concept of call by value in c language by the example given below:

```
#include<stdio.h>
void change(int num) {
   printf("Before adding value inside function num=%d \n",num);
   num=num+100;
   printf("After adding value inside function num=%d \n", num);
}
int main() {
   int x=100;
   printf("Before function call x=%d \n", x);
   change(x);//passing value in function
   printf("After function call x=%d \n", x);
return 0;
}
```

Output

```
Before function call x=100

Before adding value inside function num=100

After adding value inside function num=200

After function call x=100
```

Call by reference in C

- In call by reference, the address of the variable is passed into the function call as the actual parameter.
- The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.
- In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

Consider the following example for the call by reference.

```
#include<stdio.h>

void change(int *num) {

   printf("Before adding value inside function num=%d \n",*num);

   (*num) += 100;

   printf("After adding value inside function num=%d \n", *num);

}

int main() {

   int x=100;

   printf("Before function call x=%d \n", x);

   change(&x);//passing reference in function

   printf("After function call x=%d \n", x);

return 0;

}
```

Output

```
Before function call x=100

Before adding value inside function num=100

After adding value inside function num=200

After function call x=200
```

Difference between call by value and call by reference.

Call by Value	Call by Reference
The actual arguments can be variable or constant.	The actual arguments can only be variable.
The values of actual argument are sent to formal argument which are normal variables.	The reference of actual argument is sent to formal argument which are pointer variables.
Any changes made by formal arguments will not reflect to actual arguments.	Any changes made by formal arguments will reflect to actual arguments.

5.5 Return type and Non return type functions

A function in C can be called either with arguments or without arguments. These functions may or may not return values to the calling functions. All C functions can be called either with arguments or without arguments in a C program. Also, they may or may not return any values. Hence the function prototype of a function in C is as below:

1. **Function with no argument and no return value:** When a function has no arguments, it does not receive any data from the calling function. Similarly, when it does not return a value, the calling function does not receive any data from the called function.

Syntax:

Function declaration: void function();

Function call: function();

```
Function definition:
void function()
{           statements;
                 }
#include <stdio.h>
void value(void);
void main()
{
   value();
}
void value(void)
{
   int year = 1, period = 5, amount = 5000, inrate = 0.12;
   float sum;
   sum = amount;
   while (year <= period) {
      sum = sum * (1 + inrate);
      year = year + 1;
   }
   printf(" The total amount is %f:", sum);
}
```

Output: The total amount is 5000.000000

2. **Function with arguments but no return value**: When a function has arguments, it receives any data from the calling function but it returns no values.

Syntax:

```
        Function declaration: void function ( int );
        Function call : function( x );
        Function definition:
                    void function( int x )
                    {
                      statements;
                    }
        #include <stdio.h>
        void function(int, int[], char[]);
        int main()
        {
           int a = 20;
           intar[5] = { 10, 20, 30, 40, 50 };
           char str[30] = "Nepalinbeauty";
           function(a, &ar[0], &str[0]);
           return 0;
        }
        void function(int a, int* ar, char* str)
        {
           inti;
           printf("value of a is %d\n\n", a);
           for (i = 0; i< 5; i++) {
              printf("value of ar[%d] is %d\n", i, ar[i]);
           }
        printf("\nvalue of str is %s\n", str);
    }
```

Output:

```
value of a is 20
value of ar[0] is 10
value of ar[1] is 20
value of ar[2] is 30
value of ar[3] is 40
value of ar[4] is 50
The given string is: Nepalinbeauty
```

3. **Function with no arguments but returns a value:** There could be occasions where we may need to design functions that may not take any arguments but returns a value to the calling function. An example for this is getchar function it has no parameters but it returns an integer an integer type data that represents a character.

Syntax:

```
Function declaration: int function();

Function call : function();

Function definition :

        int function()
        {
            statements;
            return x;
        }
#include <math.h>
#include <stdio.h>
```

```
int sum();
int main()
{
   int num;
   num = sum();
   printf("\nSum of two given values = %d", num);
   return 0;
}
int sum()
{
   int a = 50, b = 80, sum;
   sum = sqrt(a) + sqrt(b);
   return sum;
}
```

Output:

```
Sum of two given values = 16
```

4. Function with arguments and return value

Syntax:

```
Function declaration : int function ( int );
Function call : function( x );
Function definition:
          int function( int x )
          {
            statements;
            return x;
```

```
        }
#include <stdio.h>
#include <string.h>
int function(int, int[]);
int main()
{
  inti, a = 20;
  intarr[5] = { 10, 20, 30, 40, 50 };
  a = function(a, &arr[0]);
  printf("value of a is %d\n", a);
  for (i = 0; i< 5; i++) {
    printf("value of arr[%d] is %d\n", i, arr[i]);
  }
  return 0;
}
int function(int a, int* arr)
{
  inti;
  a = a + 20;
  arr[0] = arr[0] + 50;
  arr[1] = arr[1] + 50;
  arr[2] = arr[2] + 50;
  arr[3] = arr[3] + 50;
  arr[4] = arr[4] + 50;
  return a;
```

```
}
```

Output:

```
value of a is 40

value of arr[0] is 60

value of arr[1] is 70

value of arr[2] is 80

value of arr[3] is 90

value of arr[4] is 100
```

5.6 Function prototyping

Function prototype tells compiler about number of parameters function takes, data-types of parameters and return type of function. By using this information, compiler cross checks function parameters and their data-type with function definition and function call. If we ignore function prototype, program may compile with warning, and may work properly. But sometimes, it will give strange output and it is very hard to find such programming mistakes.

Syntax:

```
return_typefunction_name(parameters)
{
//function body
}
```

Unit: 6
Arrays and String

6.1 Definition

An array is a collection of variables of the same type that are referenced by a common name. Specific elements or variables in the array are accessed by means of an index into the array.

In C all arrays consist of contiguous memory locations. The lowest address corresponds to the first element in the array while the largest address corresponds to the last element in the array.

6.2 Array Types

C supports both single and multi-dimensional arrays.

6.3 Single Dimensional Arrays

Syntax: type var_name[size] ;

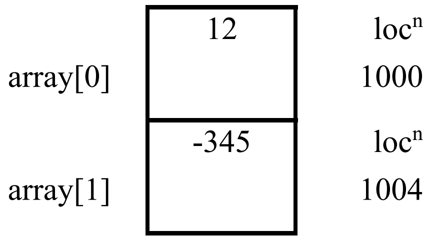
where type is the type of each element in the array, var_name is any valid C identifier, and size is the number of elements in the array which has to be a constant value.

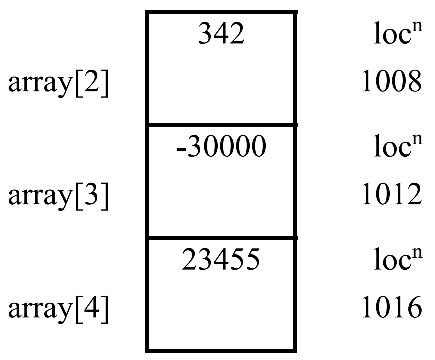
Note: In C all arrays use zero as the index to the first element in the array.

For Example: -

```
int array[ 5 ] ;
```

Which we might illustrate as follows for a 32-bit system where each int requires 4 bytes.





Note: The valid indices for array above are 0 .. 4, i.e. 0... Number of elements - 1

For Example: - To load an array with values 0 .. 99

```
int x[100] ;

inti ;

    for ( i = 0; i< 100; i++ )

        x[i] = i ;
```

Arrays should be viewed as just collections of variables so we can treat the individual elements in the same way as any other variables. For example we can obtain the address of each one as follows to read values into the array

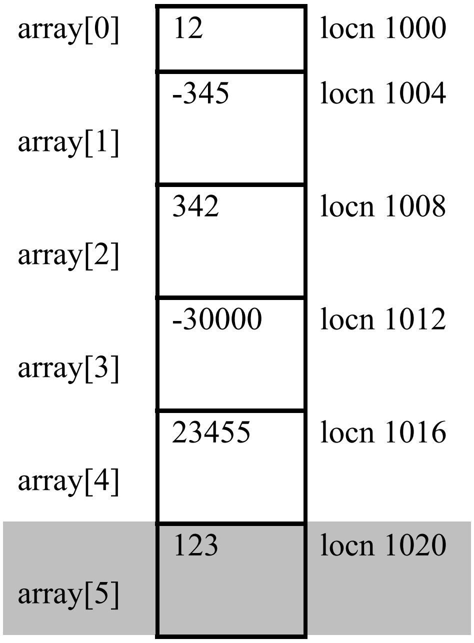
```
for ( i = 0; i< 100; i++ ) {

    printf( "Enter element %d", i + 1 ) ;

        scanf( "%d\n", &x[i] ) ;

        }
```

Note: Note the use of the printf statement here. As arrays are normally viewed as starting with index 1 the user will feel happier using this so it is good policy to use it in “public”.

To determine to size of an array at run time the sizeof operator is used. This returns the size in bytes of its argument. The name of the array is given as the operand

```
size_of_array = sizeof( array_name )  ;
```

Note: C carries out no boundary checking on array access so the programmer has to ensure he/she is within the bounds of the array when accessing it. If the program tries to access an array element outside of the bounds of the array C will try and accommodate the operation. For example if a program tries to access element array[5] above which does not exist the system will give access to the location where element array[5] should be i.e. 5 x 4 bytes from the beginning of the array.



This piece of memory does not belong to the array and is likely to be in use by some other variable in the program. If we are just reading a value from this location the situation isn't so drastic our logic just goes haywire. However if we are writing to this memory location we will be changing values belonging to another section of the program which can be catastrophic.

Initializing Arrays

Arrays can be initialized at time of declaration in the following manner.

typearray[size] = { value list };

For Example:-

inti[5] = {1, 2, 3, 4, 5 } ;

i[0] = 1, i[1] = 2, etc.

The size specification in the declaration may be omitted which causes the compiler to count the number of elements in the value list and allocate appropriate storage.

For Example:- inti[ ] = { 1, 2, 3, 4, 5 } ;

6.4 Multidimensional Arrays

Multidimensional arrays of any dimension are possible in C but in practice only two- or three-dimensional arrays are workable. The most common multidimensional array is a two-dimensional array for example the computer display, board games, a mathematical matrix etc.

Syntax: type name [rows] [columns];

For Example: - 2D array of dimension 2 X 3.

int d[2] [3] ;

d[0][0]	d[0][1]	d[0][2]
d[1][0]	d[1][1]	d[1][2]

A two-dimensional array is actually an array of arrays, in the above case an array of two integer arrays (the rows) each with three elements, and is stored row-wise in memory.

For Example: - Program to fill in a 2D array with numbers 1 to 6 and to print it out row-wise.

```
#include <stdio.h>
    void main( )
    {
    inti, j, num[2][3] ;
        for ( i = 0; i< 2; i++ )
            for ( j = 0; j < 3; j ++ )
            num[i][j] =  i * 3 + j + 1 ;
        for ( i = 0; i< 2; i++ )
    {
        for ( j = 0; j < 3; j ++ )
        printf("%d ",num[i][j]  ) ;
        printf("\n" );
        }
    }
```

Advantages and disadvantages of array:

Advantages:

1. It is used to represent multiple data items of same type by using only single name.
2. It can be used to implement other data structures like linked lists, stacks, queues, trees, graphs etc.
3. 2D arrays are used to represent matrices.

Disadvantages:

1. We must know in advance that how many elements are to be stored in array.
2. Array is static structure. It means that array is of fixed size. The memory which is allocated to array cannot be increased or reduced.
3. Since array is of fixed size, if we allocate more memory than requirement then the memory space will be wasted. And if we allocate less memory than requirement, then it will create problem.
4. The elements of array are stored in consecutive memory locations. So, insertions and deletions are very difficult and time consuming.

Unit: 7

Structures

7.1 Definition

Structure is a user-defined datatype in C language which allows us to combine data of different types together. Structure helps to construct a complex data type which is more meaningful. It is somewhat similar to an Array, but an array holds data of similar type only. But structure on the other hand, can store data of any type, which is practical more useful.

For example: If I have to write a program to store Student information, which will have Student's name, age, branch, permanent address, father's name etc, which included string values, integer values etc, how can I use arrays for this problem, I will require something which can hold data of different types together.

In structure, data is stored in form of **records**.

7.2 Structure variable declaration

struct keyword is used to define a structure. struct defines a new data type which is a collection of primary and derived datatypes.

Syntax:

```
struct [structure_tag]
{
//member variable 1
//member variable 2
//member variable 3
...
}[structure_variables];
```

As you can see in the syntax above, we start with the struct keyword, then it's optional to provide your structure a name, we suggest you to give it a name, then inside the curly braces, we have to mention all the member variables, which are nothing but normal C language variables of different types like int, float, array etc.

After the closing curly brace, we can specify one or more structure variables, again this is optional.

Note: The closing curly brace in the structure type declaration must be followed by a semicolon (;).

Example of Structure

```
struct Student
{
char name[25];
   int age;
char branch[10];
// F for female and M for male
   char gender;
};
```

Here struct Student declares a structure to hold the details of a student which consists of 4 data fields, namely name, age, branch and gender. These fields are called structure elements or members.

Each member can have different datatype, like in this case, name is an array of char type and age is of int type etc. Student is the name of the structure and is called as the structure tag.

Declaring Structure Variables

It is possible to declare variables of a **structure**, either along with structure definition or after the structure is defined. **Structure** variable declaration is similar to the declaration of any normal variable of any other datatype. Structure variables can be declared in following two ways:

1. **Declaring Structure variables separately**

```
struct Student
```

```
{
char name[25];
        int age;
char branch[10];
//F for female and M for male
        char gender;
};
```
struct Student S1, S2;//declaring variables of struct Student

2. Declaring Structure variables with structure definition

```
struct Student
{
char name[25];
        int age;
char branch[10];
//F for female and M for male
char gender;
}S1, S2;
```

Here S1 and S2 are variables of structure Student. However this approach is not much recommended.

7.3 Accessing members of a structure

Structure members can be accessed and assigned values in a number of ways. Structure members have no meaning individually without the structure. In order to assign a value to any structure member, the member name must be linked with the **structure** variable using a dot. Operator also called **period** or **member access** operator.

For example:

```
#include<stdio.h>
#include<string.h>
struct Student
{
char name[25];
   int age;
char branch[10];
//F for female and M for male
   char gender;
};
int main()
{
   struct Student s1;
/*s1 is a variable of Student type and  age is a member of Student*/
   s1.age =18;
/*using string function to add name */
strcpy(s1.name,"Viraaj");
/* displaying the stored values */
printf("Name of Student 1: %s\n", s1.name);
printf("Age of Student 1: %d\n", s1.age);
return0;
}
```

```
Name of Student 1: Viraaj
Age of Student 1: 18
```

We can also use scanf() to give values to structure members through terminal.

```
scanf(" %s ", s1.name);
scanf(" %d ",&s1.age);
```

Difference between array and structure.

Arrays	**Structures**
1. An array is a collection of related data elements of the same type.	1. Structure can have elements of different types
2. An array is a derived data type	2. A structure is a programmer-defined data type
3. Any array behaves like a built-in data type. All we have to do is to declare an array variable and use it.	3. But in the case of structure, first, we have to design and declare a data structure before the variable of that type are declared and used.
4. Array allocates static memory and uses index/subscript for accessing elements of the array.	4. Structures allocate dynamic memory and uses (.) operator for accessing the member of a structure.
5. An array is a pointer to the first element of it	5. Structure is not a pointer
6. Element access takes relatively less time.	6. Property access takes relatively large time.

Unit: 8
Pointers in c

8.1 Pointer Definition

A Pointer in C language is a variable which holds the address of another variable of same data type. Pointers are used to access memory and manipulate the address. Pointers are one of the most distinct and exciting features of C language. It provides power and flexibility to the language.

Benefits of using pointers

1. Pointers are more efficient in handling Arrays and Structures.
2. Pointers allow references to function and thereby helps in passing of function as arguments to other functions.
3. It reduces length of the program and its execution time as well.
4. It allows C language to support Dynamic Memory management.

8.2 Declaration of pointer variable

General syntax of pointer declaration is,

datatype *pointer_name; or datatype* pointer_name;

Data type of a pointer must be same as the data type of the variable to which the pointer variable is pointing. Void type pointer works with all data types, but is not often used.

Here are a few examples:

```
int *ip     // pointer to integer variable
float *fp;// pointer to float variable
double *dp;// pointer to double variable
char *cp;// pointer to char variable
```

Initialization of Pointer variable

Pointer Initialization is the process of assigning address of a variable to a **pointer** variable. Pointer variable can only contain address of a variable of the same data type. In C language **address operator** & is used to determine the address of a variable. The & (immediately preceding a variable name) returns the address of the variable associated with it.

```
#include<stdio.h>
void main()
{
   int a =10;
int *ptr;//pointer declaration
ptr =&a;//pointer initialization
}
```

Pointer variable always point to variables of same datatype. Let's have an example to showcase this:

```
#include<stdio.h>
void main()
{
   float a;
int *ptr;
ptr =&a;// ERROR, type mismatch
}
```

If you are not sure about which variable's address to assign to a pointer variable while declaration, it is recommended to assign a NULL value to your pointer variable. A pointer which is assigned a NULLvalue is called a **NULL pointer**.

```
#include <stdio.h>
int main()
{
int *ptr = NULL;
return0;
}
```

8.3 Reference operator (&)

The reference operator noted by ampersand ("&"), is also a unary operator in c languages that uses for assign address of the variables. It returns the pointer address of the variable. This is called "referencing" operator.

Referencing means taking the address of an existing variable (using &) to set a pointer variable. In order to be valid, a pointer has to be set to the address of a variable of the same type as the pointer, without the asterisk:

```
int  c1;
int* p1;
c1 =5;
p1 =&c1;
//p1 references c1
```

8.4 Dereference operator (*)

The dereference operator or indirection operator, noted by asterisk ("*"), is also a unary operator in c languages that uses for pointer variables. It operates on a pointer variable, and returns l-value equivalent to the value at the pointer address. This is called "dereferencing" the pointer.

Dereferencing a pointer means using the * operator (asterisk character) to access the value stored at a pointer: NOTE: The value stored at the address of the pointer must be a value of the same type as the type of variable the pointer "points" to, but there is **no guarantee** this is the case unless the pointer was set correctly. The type of variable the pointer points to is the type less the outermost asterisk.

```
int n1;
n1 =*p1;
```