Programming in C Notes

Roshan Khatri
Junior Professor
Department of Computer Science and Engineering
Nepal Engineering College

November 11, 2016

CMP 103.3 Programming in C (3-0-3)

Evaluation:

	Theory	Practical	Total
Sessional	30	20	50
Final	50	-	50
Total	80	20	100

Course Objectives:
The object of this course is to acquaint the students with the basic principles of programming and development of software systems. It encompasses the use of programming systems to achieve specified goals, identification of useful programming abstractions or paradigms, the development of formal models of programs, the formalization of programming language semantics, the specification of program, the verification of programs, etc. the thrust is to identify and clarify concepts that apply in many programming contexts:

Chapter	Content	Hrs.
1	**Introduction** History of computing and computers, programming, block diagram of computer, generation of computer, types of computer, software, Programming Languages, Traditional and structured programming concept	3
2	**Programming logic** Problems solving(understanding of problems, feasibility and requirement analysis) Design (flow Chart & Algorithm), program coding (execution, translator), testing and debugging, Implementation, evaluation and Maintenance of programs, documentation	5
3	**Variables and data types** Constants and variables, Variable declaration, Variable Types, Simple input/output function, Operators	3
4	**Control Structures** Introduction, types of control statements- sequential, branching- if, else, else-if and switch statements, case, break and continue statements; looping- for loop, while loop, do—while loop, nested loop, goto statement	6
5	**Arrays and Strings** Introduction to arrays, initialization of arrays, multidimensional arrays, String, function related to the strings	6
6	**Functions** Introduction, returning a value from a function, sending a value to a function, Arguments, parsing arrays and structure, External variables, storage classes, pre-processor directives, C libraries, macros, header files and prototyping	6

7	**Pointers**	**7**
	Definition pointers for arrays, returning multiple values form functions using pointers. Pointer arithmetic, pointer for strings, double indirection, pointer to arrays, Memory allocation-malloc and calloc	
8	**Structure and Unions**	**5**
	Definition of Structure, Nested type Structure, Arrays of Structure, Structure and Pointers, Unions, self-referential structure	
9	**Files and File Handling**	**4**
	Operating a file in different modes (Real, Write, Append), Creating a file in different modes (Read, Write, Append)	

Laboratory:

Laboratory work at an initial stage will emphasize on the verification of programming concepts learned in class and use of loops, functions, pointers, structures and unions. Final project of 10 hours will be assigned to the students which will help students to put together most of the programming concepts developed in earlier exercises.

Textbooks:

1. Programming with C, Byran Gottfried
2. C Programming, Balagurusami

References

1. A book on C by A L Kely and Ira Pohl
2. The C Programming Language by Kerighan, Brain and Dennis Ritchie
3. Depth in C, Shreevastav

Contents

List of Figures

List of Tables

1 Introduction

1.1 Computer

Basically it is a fast calculating machine which is now a days used for variety of uses ranging from house hold works to space technology. The credit of invention of this machine goes to the English Mathematician Charles Babbage.

1.2 Types of Computer

Based on nature, computers are classified into Analog computers and Digital computers. The former one deals with measuring physical quantities (concerned with continuous variables) which are of late rarely used. The digital computer operates by counting and it deals with the discrete variables.There is a combined form called Hybrid computer, which has both features.

Based on application computers are classified as special purpose computers and general computers. As the name tells special computers are designed to perform certain specific tasks where as the other category is designed to cater the needs of variety of users.

1.3 Basic Structure of a Digital Computer

The von Neumann architecture, which is also known as the von Neumann model and Princeton architecture, is a computer architecture based on that described in 1945 by the mathematician and physicist John von Neumann.This describes a design architecture for an electronic digital computer with parts consisting of a processing unit containing an arithmetic logic unit and processor registers; a control unit containing an instruction register and program counter; a memory to store both data and instructions; external mass storage; and input and output mechanisms.The key idea of the Von Neumann architecture is the stored program concept.A stored-program digital computer is one that keeps its program instructions, as well as its data, in read-write, random-access memory (RAM). Stored-program computers were an advancement over the program-controlled computers of the 1940s.

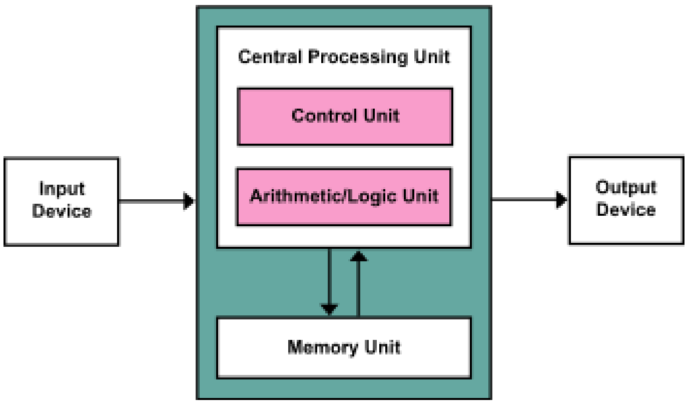


Figure 1: Block Diagram of Computer

The main components of a computer are Input unit (IU), Central Processing unit (CPU) and Output unit (OU). The information like data, programs etc are passed to the computer through input devices. The keyboard, mouse, floppy disk, CD, DVD, joystick etc are certain input devices. The output device is to get information from a computer after processing . VDU (Visual Display Unit), Printer, Floppy disk, CD etc are output devices.

The brain of a computer is CPU. It has three components- Memory unit, Control unit and Arithmetic and Logical unit (ALU)- Memory unit also called storage device is to store information. Two types memory are there in a computer. They are RAM (random access memory) and ROM (read only memory). When a program is called, it is loaded and processed in RAM. When the computer is switched off, what ever stored in RAM will be deleted.So it is a temporary memory. Where as ROM is a permanent memory, where data, program etc are stored for future use. Inside a computer there is storage device called Hard disk, where data are stored and can be accessed at any time.

The control unit is for controlling the execution and interpreting of instructions stored in the memory. ALU is the unit where the arithmetic and logical operations are performed.The information to a computer is transformed to groups of binary digits, called bit. The length of bit varies from computer to computer, from 8 to 64. A group of 8 bits is called a Byte and a byte generally represents one alphanumeric (Alphabets and Numerals) character. The physical components of a computer are called hardwares. But for the machine to work it requires certain programs (A set of instructions is called a program). They are called softwares. There are two types of softwares – System software and Application software – System software includes Operating systems, Utility programs and Language processors.

1.3.1 Operating System

The set of instructions which resides in the computer and governs the system are called operating systems, without which the machine will never function. They are the medium of communication between a computer and the user. DOS, Windows, Linux, Unix etc are Operating Systems.

1.3.2 Utility Programs

These programs are developed by the manufacturer for the users to do various tasks. Word, Excel, Photoshop, Paint etc are some of them.

1.4 Programming Language

1.4.1 Low level Language

Low level languages are machine level and assembly level language. In machine level language computer only understand digital numbers i.e. in the form of 0 and 1. So, instruction given to the computer is in the form binary digit, which is difficult to implement instruction in binary code. This type of program is not portable, difficult to maintain and also error prone. The assembly language is on other hand modified version of machine level language. Where instructions are given in English like word as ADD, SUM, MOV etc. It is easy to write and understand but it is hard for the computer to understand. So the translator used here is assembler to translate into machine level. Although language is bit easier, programmer has to know low level details related to low level language. In the assembly level language the data are stored in the computer register, which varies for different computer. Hence it is not portable.

1.4.2 High level Language

These languages are machine independent, means it is portable. The language in this category is Pascal, Cobol, Fortran etc. High level languages are not understood by the machine. So it needs to translate by the translator into machine level. A translator is software which is used to translate high level language as well as low level language in to machine level language.

1.5 Compiler, Interpreter and Assembler

Compiler and interpreter are used to convert the high level language into machine level language. The program written in high level language is known as source program and the corresponding machine level language program is called as object program. Both compiler and interpreter perform the same task but there working is different. Compiler read the program at-a-time and searches the error and lists them. If the program is error free then it is converted into object program. When program size is large then compiler is preferred. Whereas interpreter read only one line of the source code and convert it to object code. If it check error, statement by statement and hence of take more time.

An assembler program creates object code by translating combinations of mnemonics and syntax for operations and addressing modes into their numerical equivalents. This representation typically includes an operation code ("opcode") as well as other control bits and data. The assembler also calculates constant expressions and resolves symbolic names for memory locations and other entities.

1.6 Traditional and Structured Programming

Structured programming is a programming paradigm aimed at improving the clarity, quality, and development time of a computer program by making extensive use of subroutines, block structures, for and while loops—in contrast to using simple tests and jumps such as the go to statement which could lead to "spaghetti code" causing difficulty to both follow and maintain.

It is possible to do structured programming in any programming language, though it is preferable to use something like a procedural programming language. Some of the languages initially used for structured programming include: ALGOL, Pascal, PL/I and Ada – but most new procedural programming languages since that time have included features to encourage structured programming, and sometimes deliberately left out features – notably GOTO – in an effort to make unstructured programming more difficult. Structured programming (sometimes known as modular programming) is a subset of imperative programming that enforces a logical structure on the program being written to make it more efficient and easier to understand and modify.

2 Programming Logic

Problems solving

First and foremost step of solving a problem is knowing about the nature of problem in order to solve it. Almost all types of problems can be solved with computer. However, correct formulation of the problem in computer understandable terms is essential to formulate the problem and solve it using computer.

2.1 Understanding of problems, Feasibility and Requirement Analysis

In order to start to develop a solution to a problem using computer, we need to first understand the problem about its nature, complexities and others. Problems can be of various nature and we need to understand about the problems first to be able to solve it using computer. Once the problem has been understood and we can solve it using computer the feasibility of the problem solution needs to be determined. We can solve a problem using various logic but what needs to be worked out is the most feasible solution that can be derived with the least amount of complexities. Another major portion to solving problems using computer is the analysis of the requirements. The solution to the problem can be solved however the requirements required is beyond the possible technologies of today. Then the solution may not be feasible to the problem.

2.2 Design FlowChart and Algorithm

An algorithm is a solution to a computer programming problem. In other words a step by step procedure for developing a problem is called an algorithm.Algorithms can be written in two different ways.

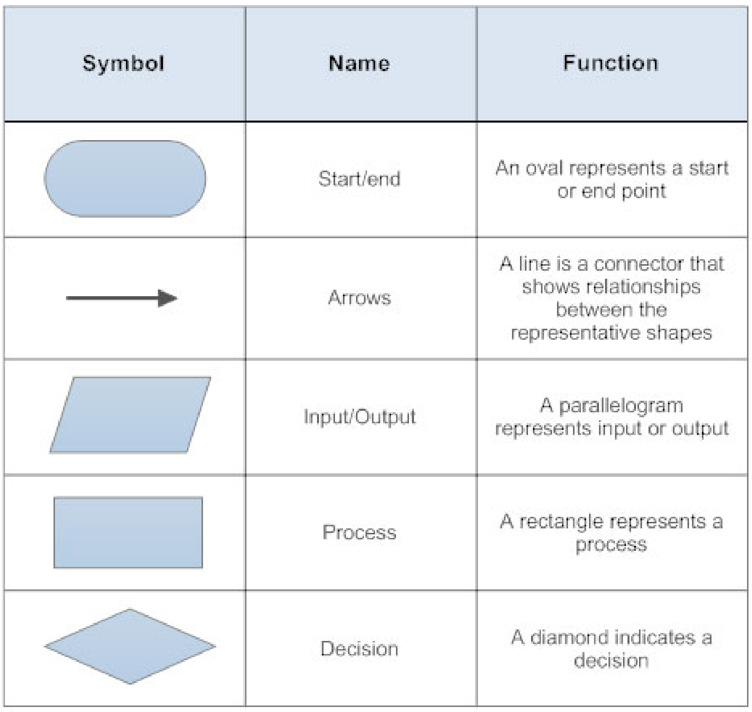
1. **Pseudocode** English like steps that describe the solution.Pseudocode is an artificial and informal language that helps programmers develop algorithms. Pseudocode is a ”text-based” detail (algorithmic) design tool.The rules of Pseudocode are reasonably straightforward. All statements showing ”dependency” are to be indented. These include while, do, for, if, switch.

```
Set total to zero
Set grade counter to one
While grade counter is less than or equal to ten
        Input the next grade
        Add the grade into the total
Set the class average to the total divided by ten
Print the class average.
```

2. **FlowChart** Pictures Detailing with specific blocks detailing out the logical flow of the solution. For a better understanding of an algorithm, it is represented pictorially.The pictorial representation of an algorithm is called a Flow Chart. The algorithm and flowchart to add two numbers can be stated as:

   (a) Step1: Input numbers as a and b
   (b) Step2: Sum = x + y
   (c) Step3: Print the sum

Various symbols can be used to represent different actions like taking input, making decisions and connecting flowcharts.



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

Figure 2: Flow chart Basic Symbols

The flow chart for the addition of two numbers whose algorithm has been stated above can be drawn as

2.3 Program Coding

Once algorithm and flowchart has been developed the task now remains is to write programs using some high level programming language.C, C++, Java, Python are the most popular programming language to develop programs. PHP and ASP remain a popular choice for developing web based applications.

2.4 Testing and Debugging

Once the program code has been written in a selected programming language of choice the next task to complete is the testing and debugging. Testing and debugging helps to find the problems associated with the program behavior under normal and abnormal circumstances. Extensive testing like white box testing and black box testing, integration testing needs to done before the program is deployed into the real world scenario.

2.5 Implementation

After a program has been written and tested it needs to be implemented to the target environment to solve the problem. Various needs of physical hardware and accessories required by the program to solve the intended problem needs to be present upon implementation. Once the program is implemented it starts to work.

2.6 Evaluation and Maintenance

The evaluation of the performance of the program needs to be done at frequent interval once the program or software is implemented. The advent of new technology, upscaling and downscaling of the business, request for the change from customers, finding of a new bug are the major reasons for the maintenance and change of the softwares. Once changes have been continuous monitoring of the software performance needs to done to discover the flaws in the software.

2.7 Documentation

The same workforce that developed the program may have left the project and gone in search of opportunities. So a program must be well documented in order for the new people to understand how the software was developed and how can it be modified. The documentation start from the very beginning of the problem formulation to the very end of the Evaluation and Maintenance.

3 Variables and Data Types

3.1 Constant and Variables

The smallest meaningful units in C programming is called C Token. Keywords, variables, various special characters fall under c tokens.

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 enumeration constants as well.Constants are treated just like regular variables except that their values cannot be modified after their definition.

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.

Type	**Size(Byte)**	**Range**
char	1	-127 to 127
int	2	-32,767 to 32,767
float	4	1*10^{-37} to 1*10^{37} six digit precision
double	8	1*10^{-37} to 1*10^{37} ten digit precision
long int	4	-2,147,483,647 to 2,147,483,647

Table 1: Data types in C

3.2 Variable declaration in C

A variable declaration provides assurance to the compiler that there exists a variable with the given type and name so that the compiler can proceed for further compilation without requiring the complete detail about the variable. A variable definition has its meaning at the time of compilation only, the compiler needs actual variable definition at the time of linking the program.

```
#include <stdio.h>
extern int a, b;
extern int c;
extern float f;
int main () {
   int a, b;
   int c;
   float f;

   a = 10;
   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;
}
```

3.3 Rules for writing variable name in C

1. A variable name can have letters (both uppercase and lowercase letters), digits and underscore only.
2. The first letter of a variable should be either a letter or an underscore. However, it is discouraged to start variable name with an underscore. It is because variable name that starts with an underscore can conflict with a system name and may cause error.
3. There is no rule on how long a variable can be. However, the first 31 characters of a variable are discriminated by the compiler. So, the first 31 letters of two variables in a program should be different.

3.4 Input Output Function

C programming language provides many of the built-in functions to read given input and write data on screen, printer or in any file.

`scanf()` and `printf()` functions

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int i;
 printf("Enter a value");
 scanf("%d",&i);
 printf( "\nYou entered: %d",i);
 getch();
}
```

3.5 Operators in C

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

- Arithmetic Operator
- Relational Operator
- Logical Operator
- Bitwise Operator
- Assignment Operator
- Misc Operator

3.5.1 Arithmetic Operator

The basic operators for performing arithmetic are the same in many computer languages:

1. + Addition
2. - Subtraction
3. * Multiplication
4. / Division
5. % Modulus (Remainder)

For exponentiations we use the library function pow. The order of precedence of these operators is % / * + - . it can be overruled by parenthesis.

Division of an integer quantity by another is referred to integer division. This operation results in truncation. i.e.When applied to integers, the division operator / discards any remainder, so 1 / 2 is 0 and 7 / 4 is 1. But when either operand is a floating-point quantity (type float or double), the division operator yields a floating-point result, with a potentially nonzero fractional part. So 1 / 2.0 is 0.5, and 7.0 / 4.0 is 1.75.

A operator acts up on a single operand to produce a new value is called a **unary operator**. The decrement and increment operators - ++ and – are unary operators. They increase and decrease the value by 1.

sizeof() is another unary operator. The output given by the sizeof() operator depends on the computer architecture. The values stated by sizeof() operator down below are based on a 16 bit compiler.

```
int x, y;
y=sizeof(x);
```

The value of y is 2. The sizeof() of an integer type data is 2 that of float is 4, that of double is 8, that of char is 1.

3.5.2 Relational Operator

< (less than), <= (less than or equal to), > (greater than), >= (greater than or equal to), == (equal to) and ! = (not equal to) are relational operators. A logical expression is expression connected with a relational operator. For example ‘b*b – 4*a*c<0 is a logical expression. Its value is either true or false.

```
int i,j,k;
i=1;
j=2;
K=i+j;
```

The expression k>5 evaluates to false and the expression k<5 evaluates to true.

3.5.3 Logical Operator

The relational operators work with arbitrary numbers and generate true/false values.You can also combine true/false values by using the Boolean operators, which take true/false values as operands and compute new true/false values. The three Boolean operators are:

1. && AND

2. || OR
3. ! NOT

```
The && (``and'') operator takes two true/false values and produces
a true (1) result if both operands are true (that is, if the left-
hand side is true and the right-hand side is true). The || (``or'')
operator takes two true/false values and produces a true (1) result
if either operand is true. The ! (``not'') operator takes a single
true/false value and negates it, turning false to true and true to
false (0 to 1 and nonzero to 0).
```

&& (and) and || (or) are logical operators which are used to connect logical expressions. Where as ! (not) is unary operator, acts on a single logical expression.

```
(a<5) && (a>2)
(a<=3) || (a>2)
```

In the example if a= 3 or a=6 the logical expression returns true.

3.5.4 Assignment Operator

These operators are used for assigning a value of expression to another identifier.

=, + =, - = , * =, /= and % = are assignment operators.
a = b+c results in storing the value of b+c in 'a'.
a += 5 results in increasing the value of a by 5.
a /= 3 results in storing the value a/3 in a and it is equivalent a=a/3

3.5.5 Conditional Operator

The operator ?: is the conditional operator. It is used as variable 1 = expression 1 ? expression 2 : expression 3. Here expression 1 is a logical expression and expression 2 and expression 3 are expressions having numerical values. If expression 1 is true, value of expression 2 is assigned to variable 1 and otherwise expression3 is assigned.

```
int a,b,c,d,e;
a=3; b=5; c=8;
d=(a<b) ? a : b;
e=(b>c) ? b : c;
```

The evaluation of the expression results the value of d = 3 and e = 8.

3.5.6 Bitwise Operator

C has a distinction of supporting special operator known as bitwise operator for manipulation of data at bit level. These operators are used for testing bits or shifting them right or left. Bitwise operator may not be applied to float or double. Following are the bitwise operators.

&	bitwise AND
|	bitwise OR
ˆ	bitwise exclusive OR
<<	shift left
>>	shift right

3.6 Operator Precedence in C

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

For example, x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has a higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.An arithmetic expression without parenthesis will be evaluated from *left to right* using the rules of precedence. There are two distinct priority levels of arithmetic operators in C.

- Higher Priority * /
- Lower Priority + -

The expression x=a-b/3+c*2-1 where a=9, b=12 and c=3 will be evaluated as
step1: x=9-12/3+3*2-1 Higher priority operators left to right division first
step2: x=9-4+3*2-1 Higher priority operators left to right multiply
step3: x=9-4+6-1 Lower priority operators left to right subtraction
step4: x=5+6-1 Lower priority operators left to right addition
step5: x=11-1 Lower priority operators left to right subtraction
step6: x=10 Final Result

Rules for Evaluation of Expression

- First, parenthesized sub expression from left to right are evaluated.
- If parentheses are nested, the evaluation begins with innermost sub expression.
- The precedence rule is applied in determining the order of operands in evaluating sub expressions.
- Arithmetic expressions are evaluated from left to right using the rules of precedence.
- When parentheses are used, the expressions within the parentheses assume highest priority.

3.7 Simple Input Output Function

For inputting and outputting data we use library function .the important of these functions are getch( ), putchar( ), scanf( ), printf( ), gets( ), puts( ).

3.7.1 getchar() function

It is used to read a single character (char type) from keyboard. The syntax is
char variable name = getchar( );
For reading an array of characters or a string we can use getchar( ) function.

```
#include<stdio.h>
#include<conio.h>
void main( )
{
char place[80];
int i;
for(i = 0;( place [i] = getchar( ))! ='\n', ++i);
}
```

3.7.2 putchar() function

It is used to display single character. The syntax is
putchar(char c);

```
#include <stdio.h>
#include <conio.h>
void main()
{
char alphabet;
printf("Enter an alphabet");
putchar'\n');
alphabet=getchar();
putchar(alphabet);
}
```

3.7.3 scanf() function

This function is generally used to read any data type- int, char, double, float,string.The syntax is
scanf (control string, list of arguments)
The control string consists of group of characters, each group beginning % sign and a conversion character indicating the data type of the data item. The conversion characters are c,d,e,f,o,s,u,x indicating the type resp. char decimal integer, floating point value in exponent form, floating point value with decimal point, octal integer, string, unsigned integer, hexadecimal integer. ie, "%s", "%d" etc are such group of characters.

```
#include<stdio.h>
#include<conio.h>
void main( )
{
char name[30], line;
int x;
float y;
........
........
scanf("%s%d%f", name, &x, &y);
scanf("%c",line);
}
```

3.7.4 printf() function

This is the most commonly used function for outputting a data of any type. The syntax is
printf(control string, list of arguments)
Here also control string consists of group of characters, each group having % symbol and conversion characters like c, d, o, f, x etc.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int x;
scanf("%d",&x);
x=x*x;
printf("The square of the number is %d",x);
}
```

Note that in this list of arguments the variable names are without & symbol unlike in the case of scanf( ) function. In the conversion string one can include the message to be displayed. In the above example "The square of the number is" is displayed and is followed by the value of x.

3.7.5 gets() and puts() function

The C library function gets() reads a line from stdin and stores it into the string pointed to by str. It stops when either the newline character is read or when the end-of-file is reached, whichever comes first.

```
#include<stdio.h>
#include<conio.h>
int main()
{
char str[50];
printf("Enter a string : ");
gets(str);
printf("You entered: %s", str);
return(0);
}
```

The C library function int puts() writes a string to stdout up to but not including the null character. A newline character is appended to the output.

```
#include <stdio.h>
#include <string.h>
int main()
{
char str1[15];
char str2[15];
strcpy(str1, "Nepal");
strcpy(str2, "Online");
puts(str1);
puts(str2);
return(0);
}
```

4 Control Statements in C

C provides two types of Control Statements

- Branching Structure
- Looping Structure

4.1 Branching Structure

Branching is deciding what actions to take and looping is deciding how many times to take a certain action.Decision making structures require that the programmer specifies one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false. Show below is the general form of a typical decision making structure found in most of the programming languages.

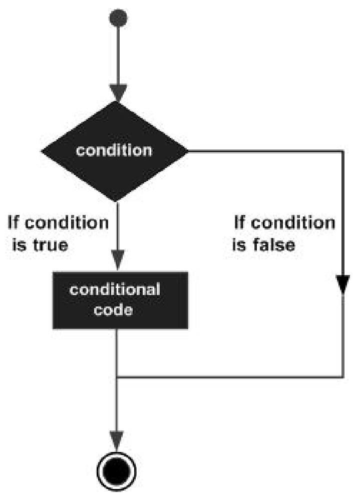


Figure 3: FlowChart Control Statement

4.1.1 if Statement

An if statement consists of a Boolean expression followed by one or more statements.If the Boolean expression evaluates to true, then the block of code inside the 'if' statement will be executed. If the 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.

```
#include <stdio.h>
int main () {
   int a = 10;
   if( a < 20 ) {
      printf("a is less than 20\n" );
   }
   printf("value of a is : %d\n", a);
   return 0;
}
```

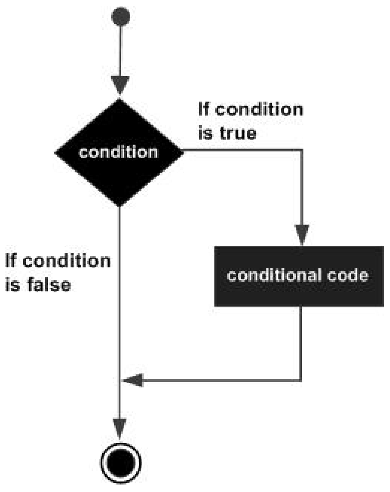


Figure 4: if statement flowchart

4.1.2 if..else Statement

An if statement can be followed by an optional else statement, which executes when the Boolean expression is false.If the Boolean expression evaluates to true, then the if block will be executed,otherwise, the else block 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.

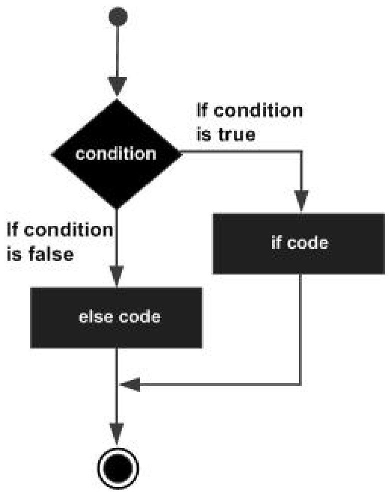


Figure 5: If else Statement

```
#include <stdio.h>
int main () {
    int a = 100;
    if( a == 10 ) {
        printf("Value of a is 10\n" );
    }
    else if( a == 20 ) {
        printf("Value of a is 20\n" );
    }
    else if( a == 30 ) {
        printf("Value of a is 30\n" );
    }
    else {
        printf("None of the values is matching\n" );
    }
    printf("Exact value of a is: %d\n", a );
    return 0;
}
```

4.1.3 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.

4.1.4 Nested if Statement

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).

4.1.5 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.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.

```
#include <stdio.h>
int main () {
    /* local variable definition */
    char grade = 'B';
    switch(grade) {
```

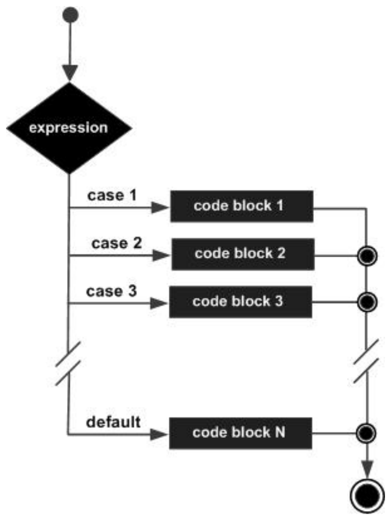


Figure 6: switch statement flowchart

```
        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;
}
```

4.1.6 The ? : Operator

Exp1? Exp2 : Exp3 is the general structure of the ? : operator.

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.

4.2 Looping Structure

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

- while loop
- do while loop
- for loop

4.2.1 while loop

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

```
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.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. flow chart and program remaining

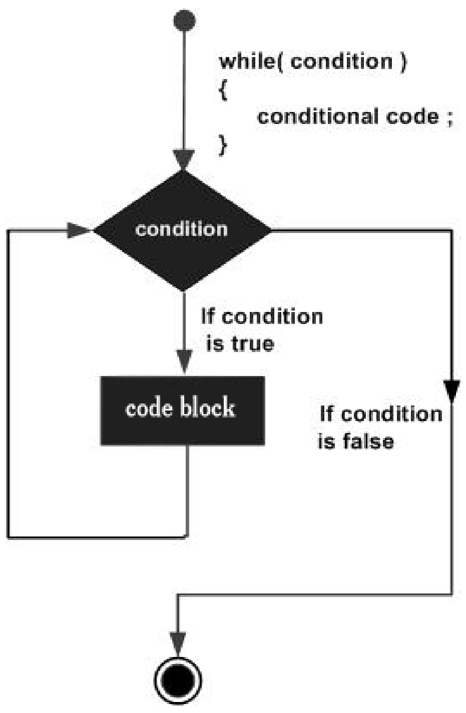


Figure 7: while loop flowchart

4.2.2 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.

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

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 the flow of control jumps to the next statement just after the 'for' loop.
3. 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.
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.

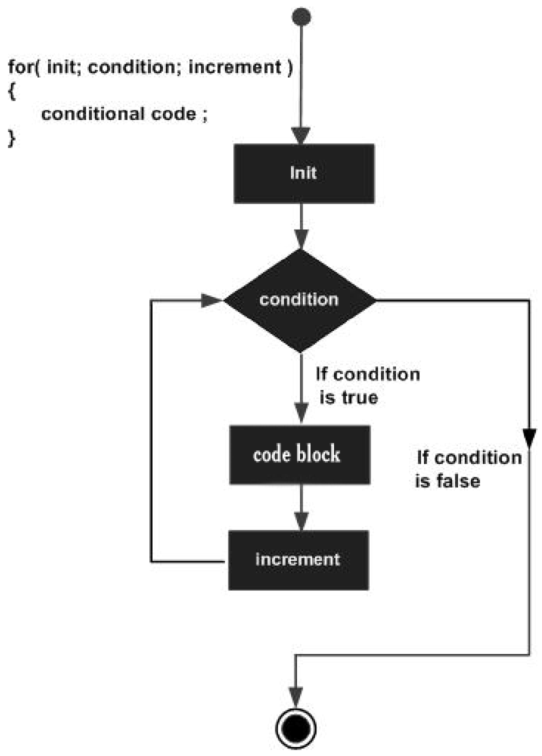


Figure 8: for loop flowchart

4.2.3 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.

```
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.

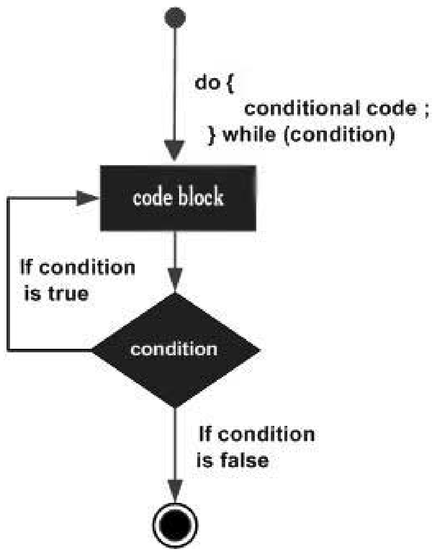


Figure 9: do..while loop flowchart

4.2.4 nested loop

C programming allows to use one loop inside another loop.

- Nested for loop

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

- Nested while loop

```
while(condition) {

   while(condition) {
      statement(s);
   }

   statement(s);
}
```

- Nested do while loop

```c
do {

   statement(s);

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

}while( condition );
```

- Find Prime numbers between 2 to 100 using prime number

```c
#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;
}
```

4.3 Break Continue and Goto Statement

4.3.1 Break Statement

The break statement terminates the loop (for, while and do...while loop) immediately when it is encountered. The break statement is used with decision making statement such as if...else. The syntax of the statement is
break;
Example of Goto Statement

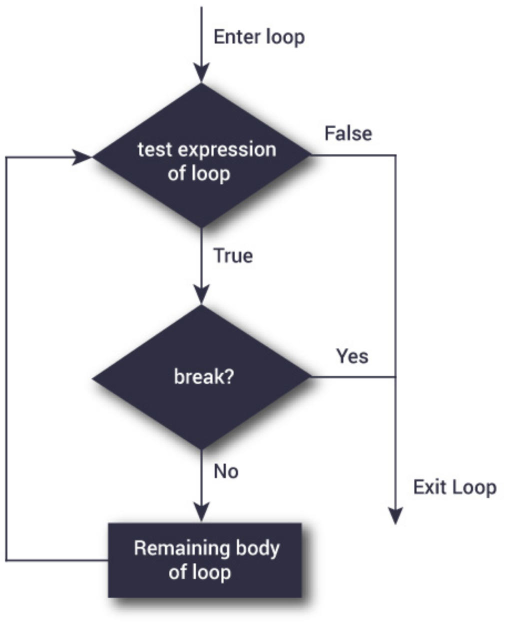
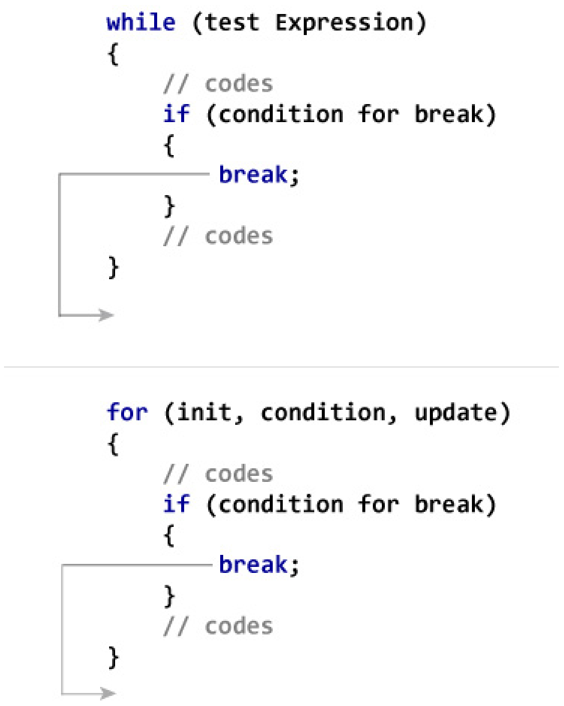


Figure 10: Flowchart Break Statement

```c
// Program to calculate the sum of maximum of 10 numbers
// Calculates sum until user enters positive number
# include <stdio.h>
int main()
{
int i;
double number, sum = 0.0;
for(i=1; i <= 10; ++i)
{
printf("Enter a n%d: ",i);
scanf("%lf",&number);
// If user enters negative number, loop is terminated
if(number < 0.0)
{
break;
}
sum += number; // sum = sum + number;
}
printf("Sum = %.2lf",sum);
return 0;
}
```

Working of Break Statement



```
while (test Expression)
{
    // codes
    if (condition for break)
    {
        break;
    }
    // codes
}

for (init, condition, update)
{
    // codes
    if (condition for break)
    {
        break;
    }
    // codes
}
```

Figure 11: Working of Break Statement

4.3.2 Continue Statement

The continue statement skips some statements inside the loop. The continue statement is used with decision making statement such as if...else. The syntax is
continue;

Example of Continue Statement

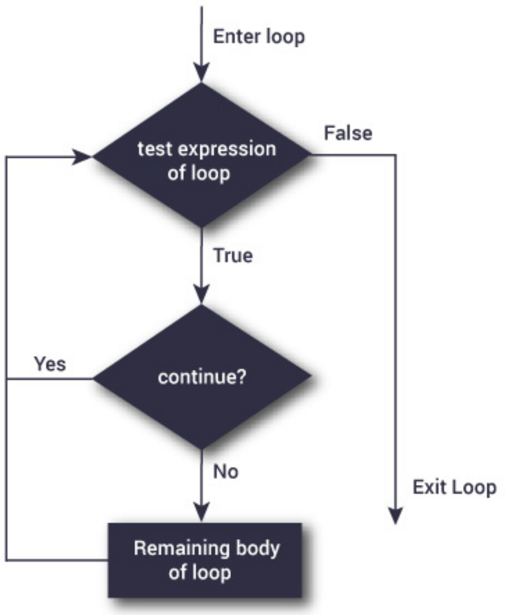
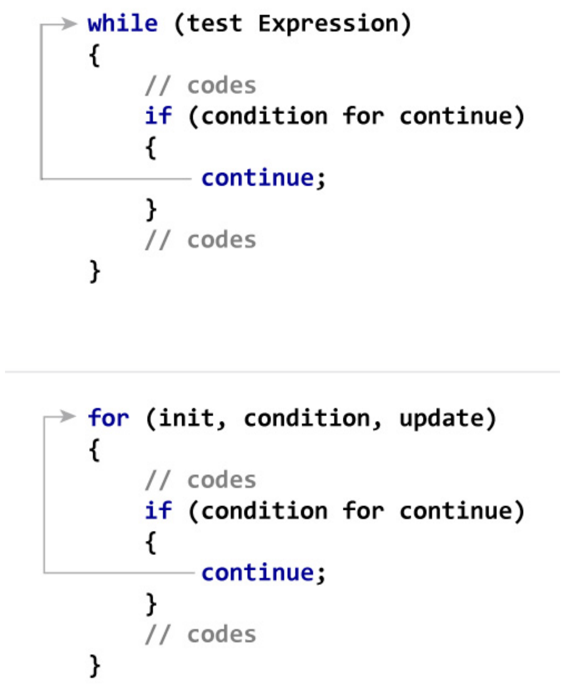


Figure 12: Flowchart Continue Statement

```
// Program to calculate sum of maximum of 10 numbers
// Negative numbers are skipped from calculation
# include <stdio.h>
int main()
{
int i;
double number, sum = 0.0;
for(i=1; i <= 10; ++i)
{
printf("Enter a n%d: ", i);
scanf("%lf",&number);
// If user enters negative number, loop is terminated
if(number < 0.0)
{
continue;
}
sum += number; // sum = sum + number;
}
printf("Sum = %.2lf",sum);
return 0;
}
```

Working of Continue Statement



```
while (test Expression)
{
    // codes
    if (condition for continue)
    {
        continue;
    }
    // codes
}
```

```
for (init, condition, update)
{
    // codes
    if (condition for continue)
    {
        continue;
    }
    // codes
}
```

Figure 13: Working of Continue Statement

4.3.3 Goto Statement

The goto statement is used to alter the normal sequence of a C program. The syntax of goto statement is

```
goto label;
.......
.......
.......
label:
statements;
```

Example of Goto Statement

```
// Program to calculate the sum and average of maximum of 5 numbers
# include <stdio.h>
int main()
{
const int maxInput = 5;
int i;
double number, average, sum=0.0;
for(i=1; i<=maxInput; ++i)
{
printf("%d. Enter a number: ", i);
scanf("%lf",&number);
// If user enters negative number, flow of program moves to label jump
if(number < 0.0)
goto jump;
sum += number; // sum = sum+number;
}
jump:
average=sum/(i-1);
printf("Sum = %.2f\n", sum);
printf("Average = %.2f", average);
return 0;
}
```

Reasons to avoid using goto statement

1. The use of goto statement may lead to code that is buggy and hard to follow.
2. Goto make jumps that are out of scope so it is difficult to follow.

5 Arrays and Strings

Arrays a kind of data structure that 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 often 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.

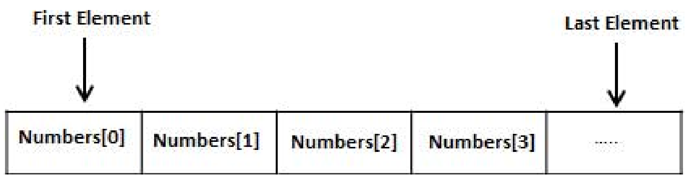


Figure 14: Arrays Representation

5.1 One Dimensional Array

5.1.1 Declaring and Initialising

```
data_type array_name[array_size];
double balance[10];
```

balance is a variable array which is sufficient to hold up to 10 double numbers.size of array defines the number of elements in an array. Each element of an array can be accessed and used as per the need of the program.

```
int age[5]={2,4,34,3,4};
int age[]={4,6,8,9,10};
```

In the second case the compiler determines the size of an array by calculating the number of elements in an array.

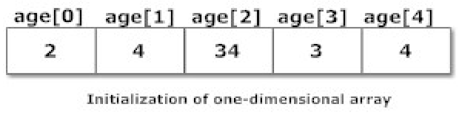


Figure 15: Arrays Initialisation

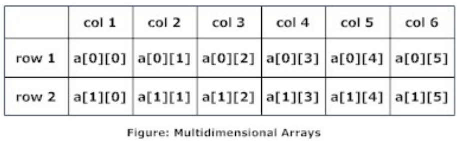
Program to find sum of marks of n students

```
#include <stdio.h>
int main(){
    int marks[10],i,n,sum=0;
    printf("Enter number of students: ");
    scanf("%d",&n);
    for(i=0;i<n;++i){
        printf("Enter marks of student%d: ",i+1);
        scanf("%d",&marks[i]);
        sum+=marks[i];
    }
    printf("Sum= %d",sum);
return 0;
}
```

5.2 MultiDimensional Array

C programming language allows programmer to create arrays of arrays known as multidimensional arrays.

```
float a[2][6];
```



	col 1	col 2	col 3	col 4	col 5	col 6
row 1	a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]	a[0][5]
row 2	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]	a[1][5]

Figure: Multidimensional Arrays

Figure 16: Arrays Initialisation

5.2.1 Initialisation of Multidimensional Arrays

```
int c[2][3]={{1,3,0},{-1,5,9}};
int c[2][3]={1,3,0,-1,5,9};
```

In the second case the compiler creates two rows from the given array.
Program to add two matrices

```
#include <stdio.h>
int main(){
    float a[2][2], b[2][2], c[2][2];
    int i,j;
    printf("Enter␣the␣elements␣of␣1st␣matrix\n");

    for(i=0;i<2;++i)
         for(j=0;j<2;++j){
         printf("Enter␣a%d%d:␣",i+1,j+1);
         scanf("%f",&a[i][j]);
         }

    printf("Enter␣the␣elements␣of␣2nd␣matrix\n");
    for(i=0;i<2;++i)
         for(j=0;j<2;++j){
         printf("Enter␣b%d%d:␣",i+1,j+1);
         scanf("%f",&b[i][j]);
         }

    for(i=0;i<2;++i)
         for(j=0;j<2;++j){
         c[i][j]=a[i][j]+b[i][j];
         }
    printf("\nSum␣Of␣Matrix:");
    for(i=0;i<2;++i)
         for(j=0;j<2;++j){
         printf("%.1f\t",c[i][j]);
              if(j==1)
                  printf("\n");
        }
return 0;
}
```

Program to Multiply Two matrices

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

```c
int main()
{
  int m, n, p, q, c, d, k, sum = 0;
  int first[10][10], second[10][10], multiply[10][10];

  printf("Enter rows and columns of first matrix\n");
  scanf("%d%d", &m, &n);
  printf("Enter the elements of first matrix\n");

  for (c = 0; c < m; c++)
    for (d = 0; d < n; d++)
      scanf("%d", &first[c][d]);

  printf("Enter rows and columns of second matrix\n");
  scanf("%d%d", &p, &q);

  if (n != p)
    printf("can't be multiplied with each other.\n");
  else
  {
    printf("Enter the elements of second matrix\n");

    for (c = 0; c < p; c++)
      for (d = 0; d < q; d++)
        scanf("%d", &second[c][d]);

    for (c = 0; c < m; c++) {
      for (d = 0; d < q; d++) {
        for (k = 0; k < p; k++) {
          sum = sum + first[c][k]*second[k][d];
        }

        multiply[c][d] = sum;
        sum = 0;
      }
    }

    printf("Product of entered matrices:-\n");

    for (c = 0; c < m; c++) {
      for (d = 0; d < q; d++)
        printf("%d\t", multiply[c][d]);

      printf("\n");
    }
  }

  return 0;
}
```

5.3 String

Strings are actually one-dimensional array of characters terminated by a null character. 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."

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

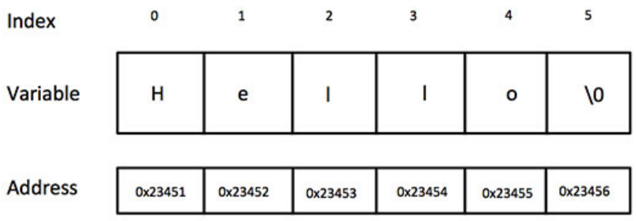


Figure 17: String Representation

5.3.1 Accessing Strings

```
//no need to use address & in scanf for strings
scanf("%s",greeting);
gets(greeting);

printf("%s",greeting);
puts(greeting);
```

5.3.2 String related fuctions

- strcpy(s1,s2) copies string s2 into the string s1.
- strcat(s1,s2) concatenates string s2 onto the end of string s1
- strlen(s1) returns the length of string s1
- strcmp(s1,s2) returns 0 if s1 and s2 are the same, less than 0 if s1<s2 and greater than 0 if s1>s2
- strrev(s1) reverses the string s1 and places it in s1

6 Function

6.1 Function Definition

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

6.2 Types of C function

There are two types of C functions

- Library Function
- User Defined Function

6.2.1 Library Function

Library functions are the in-built function in C programming system.

```
main()
printf()
scanf()
strcpy()
strcat()
strcmp()
```

are the examples of Library functions available in C.

6.3 User Defined Function

C allows programmer to define their own function according to their requirement. These types of functions are known as user-defined functions.
Function Prototype(Declaration)
Every function in C programming should be declared before they are used. These type of declaration are also called function prototype. Function prototype gives compiler information about function name, type of arguments to be passed and return type.

```c
int add(int a,int b);
```

6.4 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 consists of a function header and a function body.Here are all 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.

6.5 Calling a function

Function with return and without return can be called to using the different syntax.

```
add(a,b); //calling function without return
c=add(a,b); //calling function with return
```

6.6 Call by Value

If data is passed by value, the data is copied from the variable used in for example main() to a variable used by the function. So if the data passed (that is stored in the function variable) is modified inside the function, the value is only changed in the variable used inside the function.

```
#include <stdio.h>
void call_by_value(int x) {
        printf("Inside function x = %d before adding 10.\n", x);
        x += 10;
        printf("Inside function x = %d after adding 10.\n", x);
}
int main() {
        int a=10;

        printf("a = %d before function.\n", a);
        call_by_value(a);
        printf("a = %d after function.\n", a);
        return 0;
}
```

In the main() we create a integer that has the value of 10. We print some information at every stage, beginning by printing our variable a. Then function call_by_value is called and we input the variable a. This variable (a) is then copied to the function variable x. In the function we add 10 to x (and also call some print statements). Then when the next statement is called in main() the value of variable a is printed. We can see that the value of variable a isn't changed by the call of the function call_by_value().

6.7 Call by Reference

If data is passed by reference, a pointer to the data is copied instead of the actual variable as is done in a call by value. Because a pointer is copied, if the value at that pointers address is changed in the function, the value is also changed in main().

```
#include <stdio.h>
void call_by_reference(int *y) {
        printf("Inside function y = %d before adding 10.\n", *y);
        (*y) += 10;
        printf("Inside function y = %d after adding 10.\n", *y);
}
```

```
int main() {
        int b=10;
        printf("b = %d before function.\n", b);
        call_by_reference(&b);
        printf("b = %d after function.\n", b);
        return 0;
}
```

We start with an integer b that has the value 10. The function call_by_reference() is called and the address of the variable b is passed to this function. Inside the function there is some before and after print statement done and there is 10 added to the value at the memory pointed by y. Therefore at the end of the function the value is 20. Then in main() we again print the variable b and as you can see the value is changed (as expected) to 20.

6.8 Passing array as an argument to a function

Likewise int and floats an array can be passed as an argument to the function .

```
float largest(float a[],int size);
main(){
 float value[4]={2.5,-1.6,3.4,6.8};
 printf("%f\n",largest(value,4));
}
float largest(float a[],int size){
        int i;
        float max;
        max=a[0];
        for(i=1;i<size;i++){
                if(max<a[i])
                max=a[i];
                return(max);
        }
}
```

6.9 Storage Class in C

6.9.1 auto

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 in the same storage class. 'auto' can only be used within functions, i.e., local variables.

6.9.2 register

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.

6.9.3 static

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 the objects of its class.

```c
#include <stdio.h>
void func(void);
static int count = 5; /* global variable */
main() {
   while(count--) {
      func();
   }
   return 0;
}
/* function definition */
void func( void ) {
   static int i = 5; /* local static variable */
   i++;
   printf("i is %d and count is %d\n", i, count);
}
```

6.9.4 extern

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 however, it points 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 also be used in other files, then extern will be used in another file to provide the reference of defined variable or function. Just for understanding, extern is used to declare a global variable or function in another file.

6.10 Recursion

A function that calls itself is known as recursive function and this technique is known as recursion in C programming.Recursion is more elegant and requires few variables which make program clean. Recursion can be used to replace complex nesting code by dividing the problem into same problem of its sub-type.In other hand, it is hard to think the logic of a recursive function. It is also difficult to debug the code containing recursion.

6.10.1 Factorial Using Recursion

```c
#include<stdio.h>
int factorial(int n);
int main()
{
    int n;
    printf("Enter an positive integer: ");
    scanf("%d",&n);
    printf("Factorial of %d = %ld", n, factorial(n));
    return 0;
}
int factorial(int n)
{
    if(n!=1)
     return n*factorial(n-1);
}
```

6.10.2 Fibonacci Series Using Recursion

```
#include<stdio.h>
int Fibonacci(int);
main()
{
   int n, i = 0, c;
   scanf("%d",&n);
   printf("Fibonacci series\n");
   for ( c = 1 ; c <= n ; c++ )
   {
      printf("%d\n", Fibonacci(i));
      i++;
   }
   return 0;
}
int Fibonacci(int n)
{
   if ( n == 0 )
      return 0;
   else if ( n == 1 )
      return 1;
   else
      return ( Fibonacci(n-1) + Fibonacci(n-2) );
}
```

6.11 Preprocessor Directives

Before a C program is compiled in a compiler, source code is processed by a program called preprocessor. This process is called preprocessing.Commands used in preprocessor are called preprocessor directives and they begin with "#" symbol.

```
#include<stdio.h>
#define height 100
#define number 3.14
#define letter 'A'
#define letter_sequence "ABC"
#define backslash_char '\?'
void main()
{
        printf("value of height: %d\n",height);
        printf("value of number : %f\n",number);
        printf("value of letter : %c\n",letter);
        printf("value of letter_sequence : %s\n",letter_sequence);
        printf("value of backslah_char : %c\n",backslash_char);
        getch();

}
```

Preprocessor	**Syntax**	**Description**
Macro	#define	macro defines constant value and can be any of the basic data types
Header Inclusion Files	#include<file_name>	source code of the file file_name is included in the program at the specified place
Conditional Compilation	#ifedf, #endif,#if,#else	set of commands are included or excluded in source program before compilation with respect to the condition
Other directives	#undef,#pragma	#undef is used to undefine a defined macro variable.#pragma is used to call a function before and after function in C program.

Table 2: Preprocessor Directives

6.12 Macro Substitution

One of the major use of the preprocessor directives is the creation of macros.Macro Substitution is a process where an identifier in a program is replaced by a predefined string composed of one or more tokens.The general form of macro definition is

```
#define identifier_string
#define count 100
#define area 5*12.46
#define cube(x) (x*x*x)
```

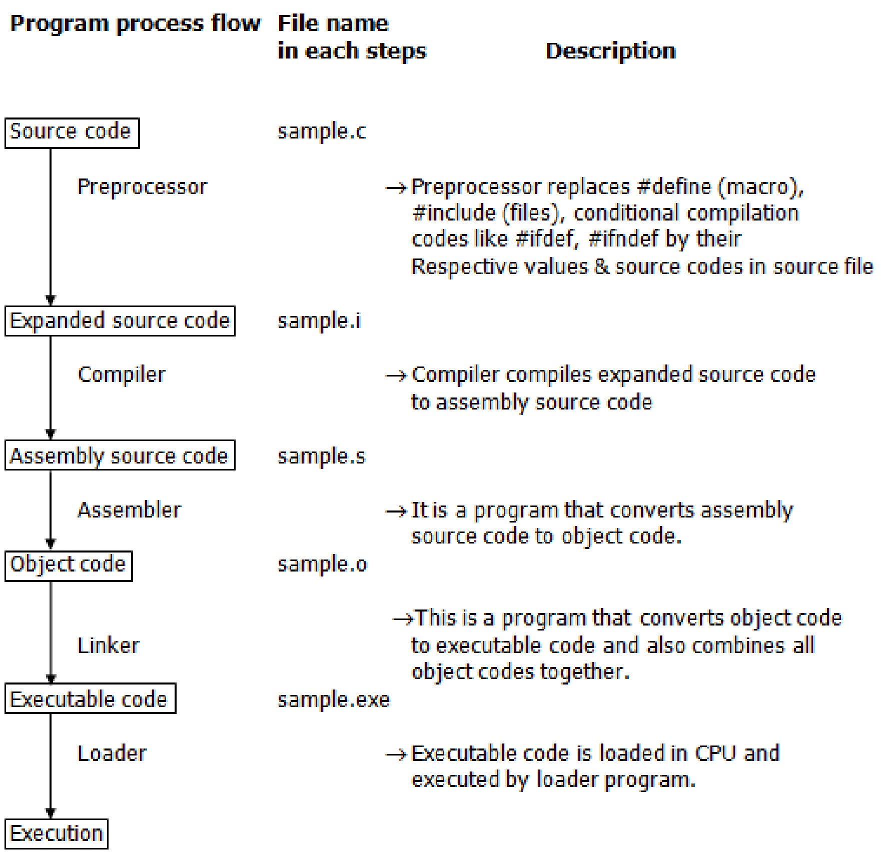

Program process flow
File name in each steps
Description
Source code
sample.c
Preprocessor
→Preprocessor replaces #define (macro), #include (files), conditional compilation codes like #ifdef, #ifndef by their Respective values & source codes in source file
Expanded source code
sample.i
Compiler
→Compiler compiles expanded source code to assembly source code
Assembly source code
sample.s
Assembler
→It is a program that converts assembly source code to object code.
Object code
sample.o
Linker
→This is a program that converts object code to executable code and also combines all object codes together.
Executable code
sample.exe
Loader
→Executable code is loaded in CPU and executed by loader program.
Execution

Figure 18: C program Compilation

7 Pointer

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 using it to store any variable address. The general form of a pointer variable declaration is

```
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 * used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations

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

7.1 Using Pointers

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

    int   var = 20;    /* actual variable declaration */
    int   *ip;         /* pointer variable declaration */
    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;
}
```

7.2 Null Pointer

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an 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.

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

   int  *ptr = NULL;
   printf("The value of ptr is : %x\n", ptr  );
   return 0;
}
```

7.3 Pointer Arithmetic

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

```
ptr++
```

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 2 bytes next to the current location in 16 bit system. This operation will move the pointer to the next memory location without impacting the actual value at the memory location. If ptr points to a character whose address is 1000, then the above operation will point to the location 1001 because the next character will be available at 1001.

7.3.1 Incrementing 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.

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

   int  var[3] = {10, 100, 200};
   int  i, *ptr;

   /* let us have array address in pointer */
   ptr = var;

   for ( i = 0; i < 3; 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;
}
```

7.3.2 Decrementing a Pointer

Likewise the increment of pointer the pointer variable can also be decremented which is shown as:

```
include <stdio.h>
int main () {

    int  var[3] = {10, 100, 200};
    int  i, *ptr;
    /* let us have array address in pointer */
    ptr = &var[2];
    for ( i = 2; 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;
}
```

7.4 Definition Pointer to Arrays

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 as the address of the first element of balance.

```
double *p;
double balance[10];
p = balance; //p=&balance[0];
```

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 the first element in 'p', you can access the array elements using *p, *(p+1), *(p+2) and so on.

```
#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; //p=&balance[0]

    /* 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;
}
```

```
main(){
int *p, sum,i;
int x[5]={5,9,6,7,3};
i=0;
p=x;   //p=&x[0];

          while(i<5){
                    printf("x[%d] %d %u",i,*p,p);
                    sum=sum+*p;
                    i++; p++;
          }
          printf("\n sum=%d\n",sum);
}
```

Here the base address of array x is assigned to the pointer p. sum adds the value of the int pointed by the pointer p using (*p) indirection operator and the increment i++ increases the value of the variable i by i and the increment p++ increase the value of the address pointed by p which is an integer and infact increased by 2 bytes.Ultimately the sum is calculated and printed.

7.5 Returning Multiple Values from a function

Return statement can return a single value. however multiple values can be returned from functions using arguments that we pass to a function.The arguments that are used to send out information are called output parameters.The mechanism of sending back information through arguments is achieved using what are known as the `address operator(&)` and `indirection operator (*)`.

```
void mathoperation(int x,int y,int *s,int *d);
main(){
          int x=20,y=20,s,d;
          mathoperation(x,y,&s,&d);
          printf("sum=%d\n diff=%d",s,d);
          }
          void mathoperation(int a,int b,int *sum, int *diff){
                    *sum=a+b;
                    *diff=a-b;
          }
}
```

The variables *sum and *diff are known as pointers and sum and diff as pointer variables.Since they are declared as the int, they can point to locations of int type data.

7.6 Pointer to String

As string can be considered as a character array. C supports an alternative way to create strings using pointer variables of type char.

```
char *str ="good";
```

This creates a string literal and then stores its address in the pointer variables str. The pointer now pointer to the first character of the string good.we can print the content of the str using printf or puts function

```
printf("%s",str);
puts(str);
```

Remember although str is a pointer to the string, it is also the name of the string. Therefore we do not need an indirection operator *.

```
main(){
 char *name;
 int length;
 char *cptr=name;
 name="DELHI"
 printf("%s\n",name);
 while(*cptr!='\0'){
        printf("%c is stored at address at %u\n",*cptr,cptr);
        cptr++;
 }
 length=cptr-name;
 printf("\n Length of String is =%d\n",name);
}
```

7.7 Double pointer

Stores the address of a pointer variable. Generally declared as

```
**ptr;
```

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

int num = 45 , *ptr , **ptr2ptr ;
ptr     = &num;
ptr2ptr = &ptr;
printf("%d",**ptr2ptr);
return(0);
}
```

7.8 Dynamic Memory Allocation

The exact size of array is unknown untill the compile time,i.e., time when a compier compiles code written in a programming language into a executable form. The size of array you have declared initially can be sometimes insufficient and sometimes more than required. Dynamic memory allocation allows a program to obtain more memory space, while running or to release space when no space is required.

7.8.1 Malloc

The name malloc stands for "memory allocation". The function malloc() reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form

```
ptr=(cast-type*)malloc(byte-size)
```

Function	Use of Function
malloc	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free	dellocate the previously allocated space
realloc	Change the size of previously allocated space

Table 3: Function Dynamic Memory Allocation

Here, ptr is pointer of cast-type. The malloc() function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

```
ptr=(int*)malloc(100*sizeof(int));
```

This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

7.8.2 Calloc

The name calloc stands for "contiguous allocation". The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

```
ptr=(cast-type*)calloc(n,element-size);
```

This statement will allocate contiguous space in memory for an array of n elements.

```
ptr=(float*)calloc(25,sizeof(float));
```

This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes.

7.8.3 free

Dynamically allocated memory with either calloc() or malloc() does not get return on its own. The programmer must use free() explicitly to release space.

```
free(ptr);
```

This statement cause the space in memory pointer by ptr to be deallocated.
Find sum of n elements entered using malloc

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)malloc(n*sizeof(int));  //malloc allocation
    if(ptr==NULL)
    {
        printf("Error! memory not allocated.");
        exit(0);
    }
    printf("Enter elements of array: ");
    for(i=0;i<n;++i)
    {
        scanf("%d", ptr+i);
        sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
    return 0;
}
```

Find sum of n elements entered using malloc

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)calloc(n,sizeof(int));
    if(ptr==NULL)
    {
        printf("Error! memory not allocated.");
        exit(0);
    }
    printf("Enter elements of array: ");
    for(i=0;i<n;++i)
    {
        scanf("%d", ptr+i);
        sum+=*(ptr+i);
    }
    printf("Sum=%d",sum);
    free(ptr);
    return 0;
}
```

8 Structure and Unions

A structure is a collection of logically related data items grouped together under a single name, called structure tag.The data items enclosed within a structure are known as members. The members can be of same or different data types.

```
struct structure_name
{
        data_type member1;
        data_type member2;
        data_type member3;
}
```

Examples of Structure can be as follows

```
struct employee{
        int emp_id;
        char name[25];
        int age;
        float salary;

}e1, e2;
```

The members of a structure do not occupy memory until they are associated with a structure variable. Above examples shows the structure variable e1 and e2 of structure employee.

8.1 Accessing Members of Structure

The members of a structure can be accessed with the help of .(dot) operator. The syntax for accessing the members of the structure variable is as follows

```
struct_variable.member
struct employee e1;
e1.emp_id;
e1.name;
e1.salary;
```

8.2 Initialising Structure

The values to the members of the structure must appear in the order as in the definition of the structure within braces and separated by the commas. C does not allow the initialisation of individual structure members as within the definition.The general from is

```
struct_namevariable={value_1, value_2......, value_n};
struct employee
{
        int emp_id;
        char name[25];
        int age;
        float salary;

};
struct employee e1={555,"Sudip",25,80000.0};
```

The members of the variable e1 emp_id,name,age,salary are initialised to 555, "Sudip",25,80000.0

8.3 Array of Structure

Likewise the array of data types arrays of structure can be initialised.

```
#include<stdio.h>
#include<conio.h>
struct employee{
        int emp_id;
        char name[25];
        int age;
        float salary;
}
int main()
{
        struct employee e[50];
        int i,n;
        for(i=0;i<50;i++)
        {
        printf("Enter id, name, age and salary");
        scanf("%d%s%d%f",&e[i].emp_id,e[i].name,&e[i].age,&e[i].salary);
        }
        for(i=0;i<50;i++){
                printf("Data on Employee %d",i+1);
                printf("ID:%d",e[i].emp_id);
                printf("Name:%s",e[i].name);
                printf("Age:%d",e[i].age);
                printf("Salary:%f",e[i].salary);
        }
        getch();
}
```

8.4 Nested Structure

Sometimes the member of structure needs to have multiple values, For the case the members should be another variable of another structure type.C allows the member structure to be the structure. The mechanism is called nesting of the structure. Nesting enables to build powerful data structures.Following is an illustration of nested structure

```
struct date{
        int day,month,year;
};
struct employee{
        int emp_id;
        char name[25];
        struct date dob;
}
```

```
struct employee{
        int emp_id;
        char name[25];
        struct date{
                int day;
                int month;
                int year;
        }dob;
}
```

Accessing the members of the nested structure

```
#include<stdio.h>
#include<conio.h>
struct employee{
        int emp_id;
        char name[25];
        int age;
        float salary;
        struct date
        {
                int day;
                int month;
                int year;
        }dob;
}
int main()
{
        struct employee e[50];
        int i,n;
        for(i=0;i<50;i++)
        {
        printf("Enter id, name, age and salary");
        scanf("%d%s%d%f",&e[i].emp_id,e[i].name,&e[i].age,&e[i].salary);
        printf("Enter the date of birth");
        scanf("%d%d%d",&e[i].dob.day,&e[i].dob.month,&e[i].dob.year);
        }
        for(i=0;i<50;i++){
                printf("Data on Employee %d",i+1);
                printf("ID:%d",e[i].emp_id);
                printf("Name:%s",e[i].name);
                printf("Age:%d",e[i].age);
                printf("Salary:%f",e[i].salary);
                printf("Date of Birth is %d-%d-%d",e[i].dob.year,
                e[i].dob.month,e[i].dob.day);
        }
        getch();
}
```

8.5 Pointer to a structure

We can have a pointer that holds the address of the structure.We can declare a pointer variable of a structure by writing

```
struct structname *pstructvar;
```

where struct is the required keyword structname represents the name of the userdefined structure and pstructvar represents the name of the pointer variable.The address of the structure variable to the pointer variable can be assigned as

```
pstructvar=&varibale;
```

Likewise the pointer to the structure variable can be accessed and processed as

```
(*struct_name).field_name=variable;
```

```
struct_name->field_name=variable;
```

```
#include<stdio.h>
struct employee
{
        int emp_id;
        char name[25];
        int age;
        float salary;
}
int main(){
        struct employee e1={1,"Roshan",25,80000.0};
        struct employee *pstructvar;
        pstructvar=&e1;
        printf("Data on Employee");
        printf("Emp_ID=%d",pstructvar->emp_id);
        printf("Name=%s",pstructvar->char);
        printf("Age=%d",pstructvar->age);
        printf("Salary=%f",pstructvar->salary);
        getch();
        return 1;
}
```

8.6 Self Referential Structure

A structure can have members which point to a structure variable of the same type. These types of structure are called self referential structure. It is widely used in dynamic data structure like linklist, trees etc.Following is the syntax of the self referential structure

```
struct struct_name
{
        data_type member1;
        .......... .......
        struct struct_name*next;
}
```

The next refers to the name of a pointer variable to point structure of its own type. The structure of type struct_name will contain a member that points to another structure of type struct_name.

8.7 Union

Union are almost like structures with subtle differences.Declaration of the union is the same as structure. Instead of the keyword struct the keyword union is used.

```
union union_name{
        data_type member1;
        data_type member2;
        ......... .......
        ......... .......
        data type member3;
}
```

8.8 Difference between Structure and Union

Structure	Union
Keyword `struct` defines structure	keyword `union` defines Union

```
struct mystructure{
        int id;
        char name[20];
        float age;
}s1;
```
	```
	struct myunion{
	        int id;
	        char name[20];
	        float age;
	}u1;
	```

Within a structure all member gets allocated and members have address that increase as the declarators are read left to right.The total size of the structure is the sum of all the members.	For a union compiler allocates the memory for the largest of all member.
Within a structure all members gets memory allocated; therefore any member can be retrieved at any time.	While retrieving data from a union the type that is being retrieved must be the type most recently stored.
One or more members of structure can be initialised at once.	A union may only be initialised with a value of the type of its first member.

Table 4: Difference between structure and union

9 File Handling in C

A file represents a sequence of bytes on the disk where a group of related data is stored. File is created for permanent storage of data. It is a ready made structure.In C language, we use a structure pointer of file type to declare a file.

```
FILE *fp;
```

9.1 Opening a File in C

The fopen() function is used to create a new file or to open an existing file.

```
*fp = FILE *fopen(const char *filename, const char *mode);
fp=fopen("text.txt","w");
//opens a file text.txt in writing mode
```

Here filename is the name of the file to be opened and mode specifies the purpose of opening the file. *fp is the FILE pointer (FILE *fp), which will hold the reference to the opened(or created) file.

9.2 Closing a File

The fclose() function is used to close an already opened file.

```
int fclose( FILE *fp );
```

Here fclose() function closes the file and returns zero on success, or EOF if there is an error in closing the file. This EOF is a constant defined in the header file stdio.h.

mode	**description**
r	opens a text file in reading mode
w	opens or create a text file in writing mode
a	open a file in append mode
r+	opens a file in both reading and writing mode
w +	opens a file in both reading and writing mode
a+	opens a file in both reading and writing mode

Table 5: File opening modes

9.3 Writing Character by Character in File

In the above table we have discussed about various file I/O functions to perform reading and writing on file. getc() and putc() are simplest functions used to read and write individual characters to a file

```c
#include<stdio.h>
#include<conio.h>
main()
{
 FILE *fp;
 char ch;
 fp = fopen("one.txt", "w");
 printf("Enter data");
 while( (ch = getchar()) != EOF) {
     putc(ch,fp);
 }
 fclose(fp);
 fp = fopen("one.txt", "r");
 while( (ch = getc()) != EOF)
     printf("%c",ch);
 fclose(fp);
}
```

9.4 Reading and Writing From a file using `fprintf()` and `fscanf()`

```c
#include<stdio.h>
#include<conio.h>
struct emp
{
    char name[10];
    int age;
};

void main()
{
    struct emp e;
    FILE *p,*q;
    p = fopen("one.txt", "a");
    q = fopen("one.txt", "r");
    printf("Enter Name and Age");
    scanf("%s %d", e.name, &e.age);
    fprintf(p,"%s %d", e.name, e.age);
    fclose(p);
    do
    {
        fscanf(q,"%s %d", e.name, e.age);
        printf("%s %d", e.name, e.age);
    }
    while( !feof(q) );
    getch();
}
```

In this program, we have create two FILE pointers and both are refering to the same file but in different modes. fprintf() function directly writes into the file, while fscanf() reads from the file, which can then be printed on console usinf standard printf() function.

9.5 fread() and fwrite()

```
fwrite( ptr, int size, int n, FILE *fp );
```

The fwrite() function takes four arguments.
ptr : ptr is the reference of an array or a structure stored in memory.
size : size is the total number of bytes to be written.
n : n is number of times a record will be written.
FILE* : FILE* is a file where the records will be written in binary mode.

```
#include<stdio.h>
struct Student
{
    int roll;
    char name[25];
    float marks;
};
void main()
{
      FILE *fp;
      char ch;
      struct Student Stu;
      fp = fopen("Student.dat","w");
      do
      {    printf("\nEnter Roll : ");
           scanf("%d",&Stu.roll);
           printf("Enter Name : ");
           scanf("%s",Stu.name);
           printf("Enter Marks : ");
           scanf("%f",&Stu.marks);
           fwrite(&Stu,sizeof(Stu),1,fp);
           printf("\nDo you want to add another data (y/n) : ");
           ch = getche();
       }while(ch=='y' || ch=='Y');
         printf("\nData written successfully...");
         fclose(fp);
}
```

```
fread( ptr, int size, int n, FILE *fp );
```

The fread() function takes four arguments.
ptr : ptr is the reference of an array or a structure where data will be stored after reading.
size : size is the total number of bytes to be read from file.
n : n is number of times a record will be read.

FILE* : FILE* is a file where the records will be read.

```c
#include<stdio.h>
struct Student
{
    int roll;
    char name[25];
    float marks;
};
void main(){
     FILE *fp;
     char ch;
     struct Student Stu;
     fp = fopen("Student.dat","r");
     if(fp == NULL)
     {
         printf("\nCan't open file or file doesn't exist.");
         exit(0);
     }
          printf("\n\tRoll\tName\tMarks\n");
          while(fread(&Stu,sizeof(Stu),1,fp)>0)
          printf("\n\t%d\t%s\t%f",Stu.roll,Stu.name,Stu.marks);
          fclose(fp);
       }
```

Write a program to write details of student to a file and read the same file and display the records.

```c
#include<stdio.h>
struct student{
 char name[20];
 int age;
 char dept[20];
};
int main(){
        FILE *fp;
        fp=fopen("student.txt","w+");
        int i;
        struct student s[2];
        for(i=0;i<2;i++){
                printf("Enter name");
                scanf("%s",s[i].name);
                printf("Enter age");
                scanf("%d",&s[i].age);
                printf("Enter dept");
                scanf("%s",s[i].dept);
                fwrite(&s[i],sizeof(s),1,fp);
}
        rewind(fp);
        for(i=0;i<2;i++){
                fread(&s[i],sizeof(s),1,fp);
                printf("\n Name: %s\t Age: %d\t Department: %s",
                s[i].name,s[i].age,s[i].dept);
        }
        fclose(fp);
        return 0;
}
```

Write a program to input data of 100 employee to a file and display records of those employees living in "Kathmandu"

```
#include<stdio.h>
#include<string.h>
struct employee{
        char name[20];
        char address[20];
        long telephone;
        float salary;
        struct dob{
                int mm,dd,yy;
        }db;

};
int main(){
struct employee e[100];
int i;
FILE *fp;
fp=fopen("employee.txt","w");
for (int i = 0; i < 100; ++i)
{
        printf("\nEnter name");
        scanf("%s",e[i].name);
        printf("\nEnter address");
        scanf("%s",e[i].address);
        printf("\nEnter telephone");
        scanf("%ld",&e[i].telephone);
        printf("\nEnter salary");
        scanf("%f",&e[i].salary);
        printf("\nEnter date of birth in dd-mm-yyyy");
        scanf("%d-%d-%d",&e[i].db.dd,&e[i].db.mm,&e[i].db.yy);
        fwrite(&e[i],sizeof(e),1,fp);
}
for (int i = 0; i < 100; ++i)
{
        if(strcmp("Kathmandu",e[i].address)==0)
        {
                printf("\n%s",e[i].name);
                printf("\n%s",e[i].address);
                printf("\n%ld",e[i].telephone);
                printf("\n%f",e[i].salary);
                printf("\n%d-%d-%d",e[i].db.dd,e[i].db.mm,e[i].db.yy);
        }
}
return 0;
}
```

Write a program to read the name, author and price of 500 books in a library from the file library.dat. Now print the book name and price of those books whose price is above 300.

```
#include <stdio.h>
#include <string.h>
struct book{
        char name[100];
        char author[100];
        float price;
};
```

```c
int main()
{
        FILE *fp;
        struct book b;
        fp=fopen("library.dat","r");

        while(fread(&b,sizeof(b),1,fp)>0){
                if(b.price >300.0)
                {
                        printf("%s\n",b.name);
                        printf("%f\n",b.price);
                }
        }

        return 0;
}
```

Write a program to create a "student.txt" file to store the above records for 100 students. Also display the records of students who are not from Pokhara.

```c
#include <stdio.h>
#include <string.h>
struct student
{
        int roll;
        char name[20];
        char address[20];
        char faculty[20];
        struct dob
        {
                int mm,dd,yy;
        }db;
};
int main()
{
        FILE *fp;
        int i;
        struct student s[100];
        fp=fopen("student.txt","w");
        for (int i = 0; i < 100; ++i)
        {
                printf("Enter Roll No.\n");
                scanf("%d",&s[i].roll);
                printf("\nEnter name");
                scanf("%s",s[i].name);
                printf("\nEnter address");
                scanf("%s",s[i].address);
                printf("\nEnter Faculty");
                scanf("%s",s[i].faculty);
                printf("\n Enter date of birth in format dd-mm-yyyy");
                scanf("%d-%d-%d",&s[i].db.dd,&s[i].db.mm,&s[i].db.yy);
                fwrite(&s[i],sizeof(s),1,fp);
        }
        for (int i = 0; i < 100; ++i)
        {
                if(strcmp("Pokhara",s[i].address)!=0)
                {
                        printf("%d\n",s[i].roll  );
                        printf("%s\n",s[i].name  );
```

```
                    printf("%s\n",s[i].address );
                    printf("%s\n",s[i].faculty );
                    printf("%d-%d-%d\n",s[i].db.dd,s[i].db.mm,s[i].db.yy );
            }
        }
        fclose(fp);
        return 0;
}
```