



CODING CAREER BLUEPRINT

*Complete C Programming
Language*



-ATISH JAIN

Notice of Rights

No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual, or otherwise, without the prior written permission of AH CAREER Pvt. Ltd.. except under the terms of a courseware site license agreement.

Trademark Notice

Throughout this courseware title, trademark names are used. Rather than just put a trademark symbol in each occurrence of a trademarked name, we state we are using the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.

Notice of Liability

The information in this courseware title is distributed on an 'as is' basis, without warranty. While every precaution has been taken in the preparation of this course, neither the authors nor AH CAREER Pvt. Ltd.. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the instructions contained in this book or by the computer software and hardware products described in it.

Disclaimer

We make a sincere effort to ensure the accuracy of the material described herein; however, AH CAREER Training Materials makes no warranty, expressed or implied, with respect to the quality, correctness, reliability, accuracy, or freedom from error of this document or the products it describes. Data used in examples and sample data files are intended to be fictional. Any resemblance to real persons or companies is entirely coincidental.

All information in this manual was correct at the time of writing. AH CAREER is not affiliated with nor has any control over changes made to the product described in this manual. These include, but are not limited to, changes in the application's color scheme, icon appearance and locations, addition or removal of program features, online templates, and help content. AH CAREER reserves the right to make corrections to the courseware at any time and without notification.

Terms and conditions

Sample versions: If the version of courseware that you are viewing is marked as NOT FOR TRAINING, SAMPLE, or similar, then it is made available for content and style review only and cannot be used in any part of a training course. Sample versions may be shared but cannot be re-sold to a third party. **For licensed users:** This document may only be used under the terms of the license agreement from AH CAREER Pvt. Ltd.. AH CAREER reserves the right to alter the licensing conditions at any time, without prior notice.

AH CAREER Pvt. Ltd..
Hyderabad, Telengana
India

Foreword

This material has one to one correspondence with the instructions in the class. This will be supplementary reading material to the classroom instructions.

To understand class session, we recommend the following study pattern :

1. Before attending the class, browse through the corresponding sessions material
2. Attend the lecture
3. Immediately after the lecture, go through the sessions till you understand the concept totally.
4. Get the doubts clarified from the center's faculty members.

We have tried to ensure that there are no major technical or grammatical errors in this material. But in a work of this magnitude, some small errors are bound to creep in. As student you are likely to go through every line of this material. If you notice some errors, please let us know through the feedback sheet (available at the end of this book, please hand it over to our office). Your feedback will help to us to design more effective and error free material in future.

We would like to wish you every success in your determined efforts to master computers.

Good luck and happy programming.....

Note: Send us any corrections or suggestions to **info@ahcareer.in**

AH CAREER

Index

1. Introduction to programming	5
2. C Basics.....	14
3. Control Structures	49
4. ARRAYS.....	96
5. Functions (Basic)	133
6. Macros and Storages Classes.....	146
7. Pointers & Advanced Functions.....	161
8. Structures.....	214
9. Files (Disk I/O).....	231
10. Command Line Arguments.....	256
11. OTHERS.....	263
Appendix.....	267

Introduction to Programming

Topic # 1

Atish Jain

1.1 Application areas of Computers

Now a days usage of a Computer has become a part of our everyday life, Lets see various areas where computers are useful to humans, so that you don't have any doubt in your mind about its applications

- Computerized Banking
- Reservation of Rail, Air and Bus tickets – online
- Online Bill Payments
- Email, E-Commerce / E-Shopping
- Using websites for viewing of Exam Results
- CAD (Computer Aided Designing) & CAM (Computer Aided Manufacturing)
- Medical Examination
- Finger Print Verification – in Crime Investigation
- Shopping Malls – bill generation
- Animation in Movies – Avatar (Hollywood), Eega (Tollywood)
- Mobile Phones, Online Examination System etc

1.2 What is a Program / Computer Programming?

Computers are programmable. They can be programmed by means of programming languages. In the pair of words “programming language” the term “programming” is the commanding tool (logic and technique of instructing) that is used to communicate with computers, just like English or Hindi is a communication tool between two persons. The languages like C, Pascal and Java are meant for communication between person (programmer) and the computer system. Programming means writing a sequence of steps or instructions to achieve a desired task.

Computer is a man made machine, in its simplest form it is nothing more than a collection of Silicon, Plastic, Metal, glass & wire. When you turn it ON, it does precisely what it has been instructed to do nothing more & nothing less. That's where programming comes in. A **program** is a set of instructions that tells the computer to do something. **Programming/Coding** is a process of preparing these instructions.

The Steps in Programming

S1. Algorithm / Logic Preparation: Preparing the plan to solve the particular problem. It contains the steps which are to be performed to get the work done by a Program/Software. It is independent of the programming language that we are going to use.

Algorithms are expressed in two ways:

- Pseudo code.
- Flowchart.

Pseudo code

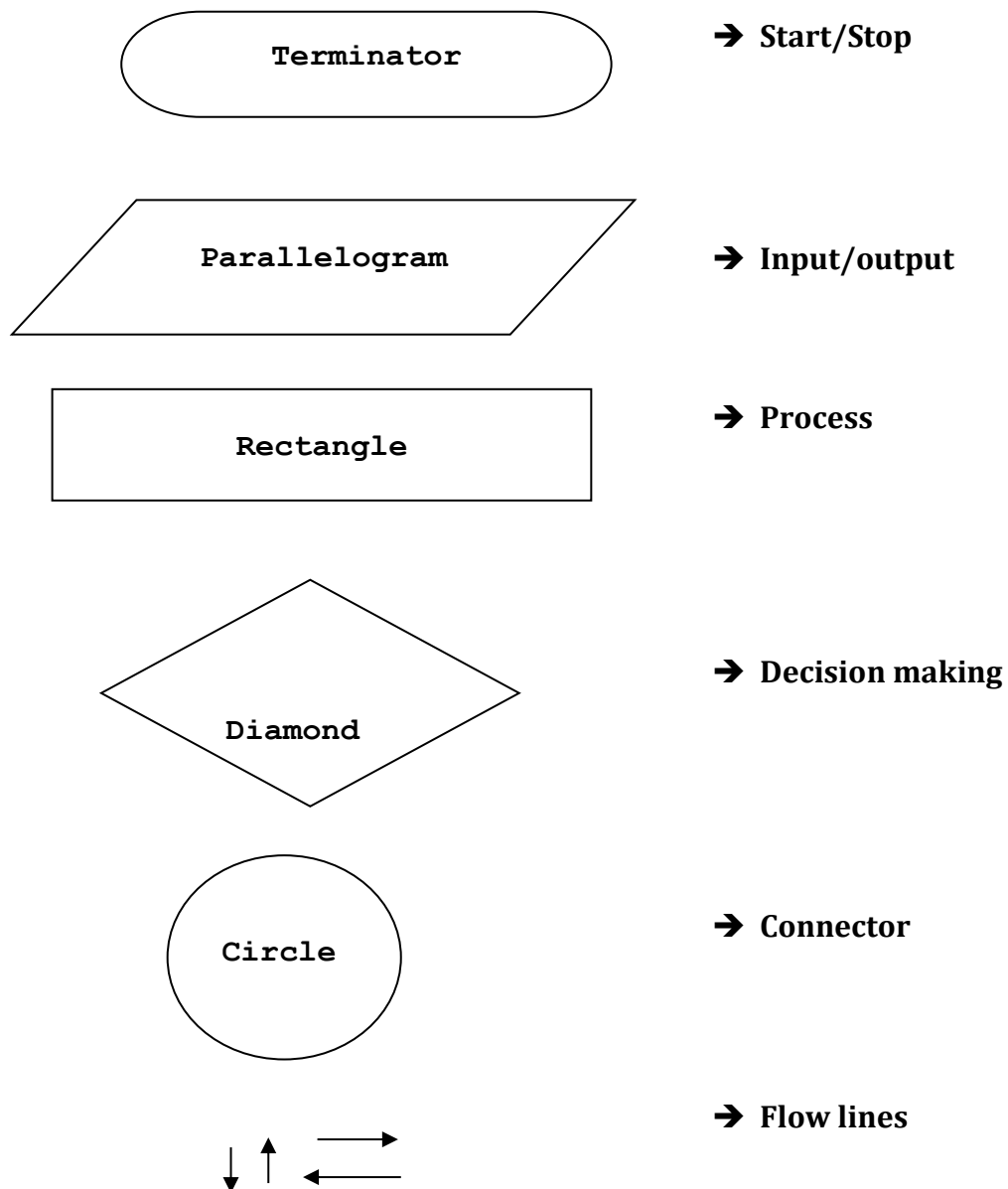
Pseudo code is a short hand way of describing a computer program. Using Pseudo code, it is easier for a non-programmer to understand the general workings of the program.

Flowchart

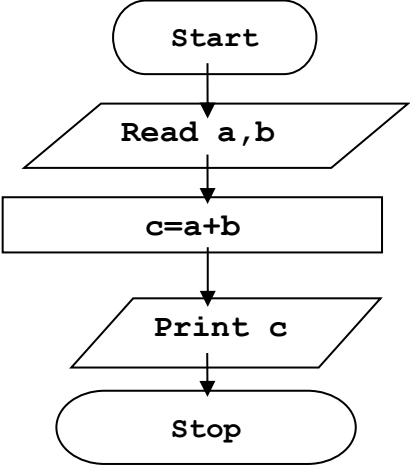
Flowchart is a graphical representation of an Algorithm.

Flowchart Symbols - Some of the basic symbols used in flowcharting.

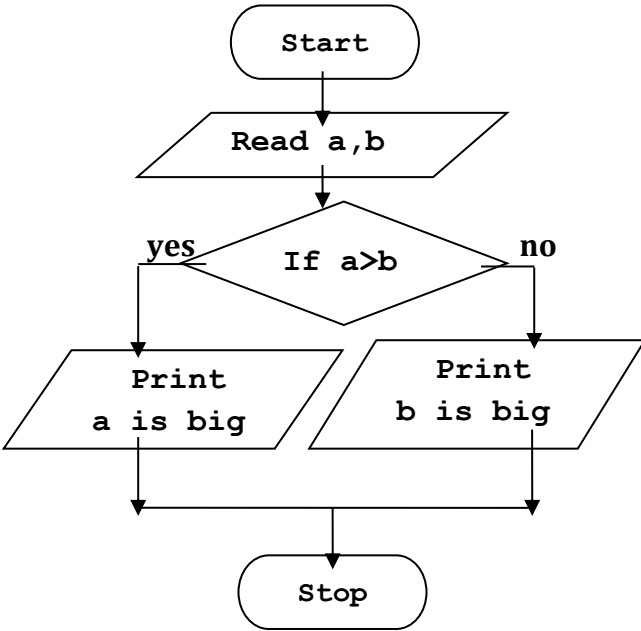
Flowchart symbols:



Pseudo code and Flowchart to add 2 numbers

<i>Pseudo code</i>	<i>Flowchart</i>
Step1: start Step2: read a,b Step3: $c=a+b$ Step4: print c Step5: stop	 <pre>graph TD; Start([Start]) --> Read[/Read a,b/]; Read --> Process[c=a+b]; Process --> Print[/Print c/]; Print --> Stop([Stop]);</pre>

Pseudo code and flowchart to find biggest of 2 numbers

<i>Pseudo code</i>	<i>Flowchart</i>
Step1: start Step2: read a,b Step3: if $a>b$ then print a is big else print b is big Step4: stop	 <pre>graph TD; Start([Start]) --> Read[/Read a,b/]; Read --> Decision{If a>b}; Decision -- yes --> PrintA[/Print a is big/]; Decision -- no --> PrintB[/Print b is big/]; PrintA --> Stop([Stop]); PrintB --> Stop;</pre>

Pseudo code and flowchart to print even nos. till 100

<i>Pseudo code</i>	<i>Flowchart</i>
Step1: start Step2: no=2 Step3: while no<=100 Print no no=no+2 end Step4: stop	<pre> graph TD Start([Start]) --> Init[/No = 2/] Init --> Print[/Print No/] Print --> Inc[No = No+2] Inc --> Cond{If No <= 100} Cond -- yes --> Print Cond -- no --> Stop([Stop]) </pre>

- Write the Pseudo code and draw the corresponding flowchart to calculate Interest for the Input principle amount, time and rate of interest.
- Write the Pseudo code and draw the corresponding flowchart to check whether the student has passed or failed from the input marks for 3 subjects.
- Write the Pseudo code and draw the corresponding flowchart to print 5th table till 12.

S2. Coding: Writing the instructions in a specific programming language.

S3. Compilation / Transformation: The code that we write is known as Source Code. It is close to English. But any computer ultimately understands only Binary Code/Machine Language. So our Source Code has to be converted into Machine Code.

Converting the Source code (the program which we write) into machine language is known as Compilation. This Job is done by special software – Compilers , Interpreters.

S4. Execution: Passing the Machine Code instructions to the computer to get the Output (result). First the Machine Code has to be loaded into main memory of the

computer i.e., RAM, then the CPU takes instruction by instruction from RAM, processes it and gives the result.

The logic preparation, is generally the most difficult part of the programming.

1.3 Different Programming languages

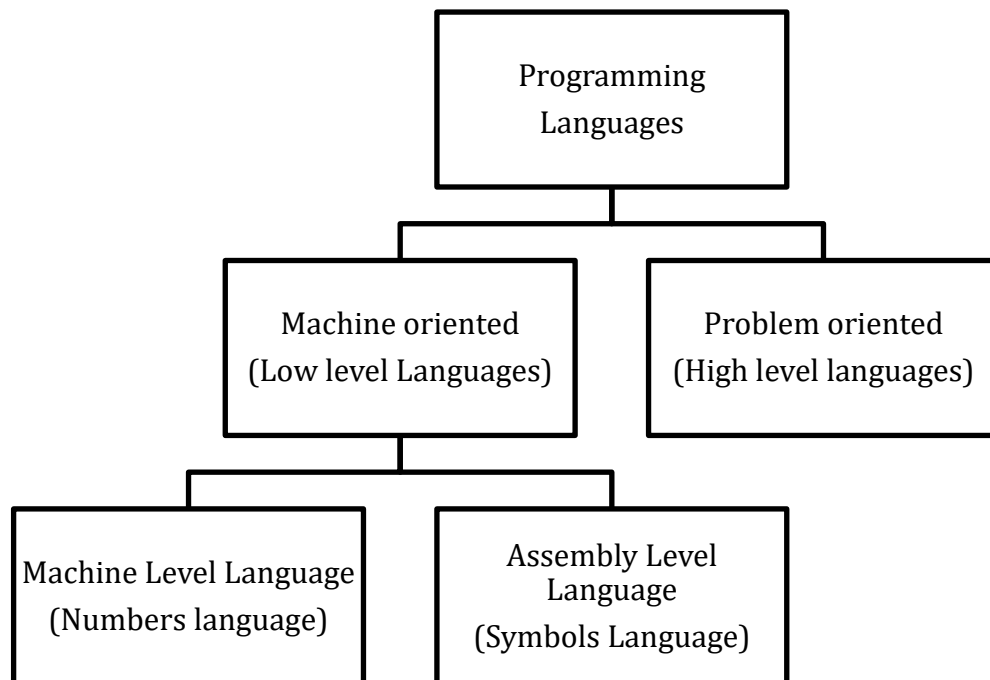
There are several programming languages – some are more popular in specific domain areas, some are very widely used in several domain areas

1.4 What does a Program consists of

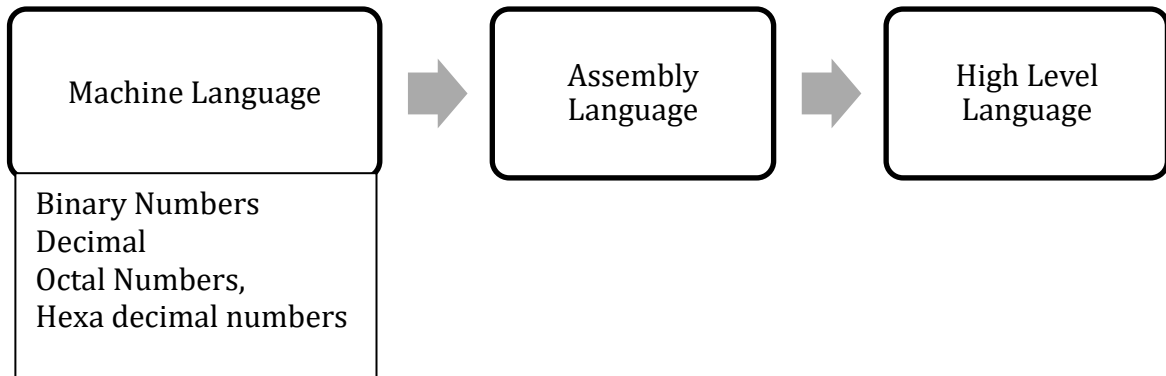
A Program is nothing but text which is comprised of alphabets, digits & special symbols.

1.5 Types of Programming languages

Basically these are two types of programming languages.



1.6 Evolution of Programming languages



Machine Language Programming

Early computers systems were literally programmed by hand. Front panel switches were used to enter instructions & data. These switches represented the **Address, Data** & the **control lines** of the computer system.

Machine languages required the writing of long strings of binary no.s to represent such operations as “add”, “sub”, “compare” etc. Later improvements allowed octal, decimal or hexadecimal representation of the binary strings.

Characteristics of Machine language programming

- Any computer can directly understand only its own machine language.
- Machine language is the natural/nature language of a particular computer.
- It is defined by the H/W design of that computer.
- Machine code languages generally consist of strings of numbers (decimal/hexadecimal/octal, which are ultimately reduced to 0's and 1's) that instructs computers to perform their most elementary operations one at a time.
- Machine code language programs are machine code dependent. i.e., a particular machine code language can be used on only one type of computer.
- Machine code language programmers must have sound knowledge of how a particular computer performs its elementary operations (arithmetic & logical). As computer became more popular, it became apparent that machine code language programming was:
 - Too slow (Program development)
 - Difficult to code
 - Error prone
 - Not portable on different types of computers.

Assembly language programming

Instead of using the Strings of numbers that computers could directly understand, programmers began using English – like abbreviations to represent the elementary operations of the computer.

Assembly language allows programmer to use symbolic addresses, which are later converted to absolute addresses by **assemblers**.

Advantages

- We can use mnemonics (abbreviations) like MOVE, CLEAR while writing source code.
- Changes could be made easier & faster.
- Easier to read & write (program development is faster)
- Error checking is provided.
- Variables are represented by symbolic names, not as memory locations.

Drawbacks

- The programmers still requires knowledge of the processor architecture & instruction set.
- The programs are machine dependent, requiring complete rewrite if the hardware is changed.
- Source program tends to be large & difficult to follow.

High-Level language programming

To speed the programming (software development) process, HLLs were developed in which a particular problem is solved in a very fewer number of instructions than that of an equivalent LLL instructions.

It tells the programmer **concentrate on the logic** of the problem to the solved rather than the intricacies of the machine code architecture.

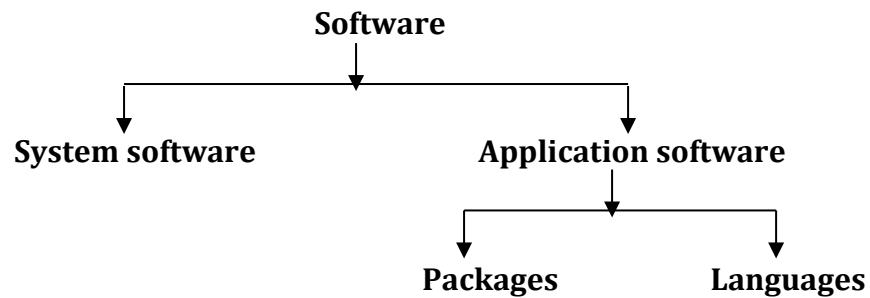
The source code is not machine code dependent. Hence programs are **portable**. (i.e., even if computer architecture changes source code need not be rewritten)

Translator programs called **Compilers/Interpreters** convert HLL programs into machine code languages.

Examples: FORTRAN, COBOL, BASIC, Pascal, C, C++, Java etc.

Software

- Set of programs which perform a particular task.
- There are two types of Softwares:



System software

- The software which is used to start the computer system or the software which makes the hardware parts of the computer system functioning is known as system software.
- E.g. DOS, Windows, UNIX, Linux etc...
- System software is also known as *Operating System or OS*.

Application software

- The software which performs a particular task is known as application software. Application software is used to perform user tasks.

Package

- Set of pre-written programs to solve a particular task.
- E.g. Word, Excel, tally etc...

Language

- Set of keywords, which are used to write a program.
- E.g. BASIC, COBOL, C, FORTRAN, PASCAL etc...

<i>Language</i>	<i>Package</i>
1. Set of keywords 2. Develops system software 3. No programming Boundaries 4. Not user friendly E.g. Basic, Cobol etc...	Set of pre-written programs Develops application software Programming boundaries User friendly E.g. Excel, Tally etc...

Exercise

State True or False

- A program is a set of instructions.
- Assembly language is also called as mnemonic language.
- Compiler is Hardware.
- Computer understands only 0s and 1s.
- Assembly language programs are portable.

C Basics

Topic # 2

Atish Jain

2. C Basics

2.1 What is 'C'?

C is a **general purpose programming language**. (slightly biased towards system programming) developed by **Dennis Ritchie** in **1972** at **AT&T** (American Telephone & Telegraph) **Bell labs., New Jersey, U.S.A.** It was developed from **B** (Bell), **BCPL** (Basic Combined Programming Language) and **ALGOL** (Algorithmic language).

C was originally designed for and implemented on the **UNIX OS** on the **DEC PDP – II** computer by **Ritchie**. The **Unix OS** (near about 95%), the **C Compiler** and essentially all **UNIX** application programs are written in **C**.

C language was standardized in 1988 by **ANSI**.

We know that a book is a collection of chapters where each chapter deals with a topic; similarly a '**C**' language program is nothing but a **collection of functions**, where a function is a block of code, which performs a particular task. Hence **C** language is called as **procedural programming language**.

2.2 Why should one learn 'C'?

All Software Engineering Aspirants should start with "**C**". Everyone has it in their Academics – **BTech, MCA, BSc Computers** etc

- No Programming background is required.
- Using **C** we can learn the common concepts of programming very easily.
 - **eg:** variables, memory allocation, control structures, arrays, functions
 - pointers, structures, disk I/O etc
- **C** is a very small language -with only 32 keywords (at the time of its creation)
- **C** is very simple.
- **C** is a general-purpose Programming language.
- As of today, if you want to be a Programmer, you should be proficient in atleast
- **Java/C#.Net**.

The Path: C -> C++ -> Java / C#.Net

2.3 Why 'C' language is so popular/powerful?

C is a Middle Level Language; it has the features of both low-level languages and high-level languages. i.e., software development is relatively **faster** than assembly language (low level) and program execution is relatively **faster** than other high – level programming languages.

Communication with the Hardware is easy which was earlier possible only with assembly languages.

It is suited for a variety of programming domains.

- System programming
- General application programming
- Embedded systems programming
- Game Programming
- Network programming.

Most of the Software in the world were developed using C/C++. The world's software giants, Microsoft, Borland develop most of their software using C & C++.

In fact anything and every thing that can be done on computer, can be done using 'C', provided we are good at it and the particular programming domain.

Why was "C" invented

C was invented to create a portable Operating System and the first portable Operating System was UNIX

Features of C language:

1. Portability:

C programs are portable, they can be transferred to any other system for execution.

2. Modularization/structured language:

A lengthy program can be divided into modules.

3. Free form language:

There are no specific rules for the positions on the screen at which the instructions should be written.

4. Middle level language:

It is the very important feature of C language. All the programming languages are divided into two categories.

Low level languages:

Direct interaction with the hardware, which gives better Machine efficiency.

Ex: Machine language and Assembly language.

High level language:

Programs are written using simple English language which gives Better programming efficiency

Ex: Basic, COBOL, Pascal, etc.

C has the features of both high level and low level language.

That's why C is known as middle level language.

5. General purpose language:

C can be used for system programming as well as application Programming.

6. Case sensitive

It means Capital letters are different from small letters.

7. Function oriented

Entire program is written as a collection of functions. There are nearly 400+ functions in c language.

8. Weakly typed language

There is no strict datatype checking. We can convert integer to float and float to integer.

9. Liberal and Error prone

C doesn't check all errors.

10. Few Keywords

There are 32 keywords in c language.

History of C language:

Year	Language	Developed by	Remarks
1960	ALGOL	International committee	Too general, too abstract
1963	CPL	Cambridge university	Hard to learn, difficult to
1967	BCPL	Martin Richards at Cambridge	Could deal with only specific
1970	B	Ken Thompson at AT&T Bell labs	Could deal with only specific
1972	C	Dennis Ritchie at AT&T bell labs	Taken the good features of

2.4 “Welcome” program

How to write a program?

To write a program we need an editor software. To write a “C” program we have a special editor called “**TurboC editor**” also called as “**TurboC IDE**” (Integrated Development Environment), which is an integration of several tools.

- Editor
- Preprocessor
- Debugger
- Compiler
- Linker
- Standard Object Library
- Header Files
- Help Document.

Program 2.1

```
i.) void main()
    {
        printf("Welcome to C");
    }
```

Output

Welcome to C

Turbo C IDE

After typing the program follow these steps:



Step 1 : Save your program using the **File** menu of your **Turbo C IDE**.

- Step 2 : Compile -> Alt + F9
- Step 3 : Link -> F9
- Step 4 : Execute -> Ctrl + F9
- Step 5 : To view the result -> Alt + F5

Explanation

a) The same program could have been written in two more ways.

```
ii.) main()
{
    printf("Welcome to C");
}
```

```
iii.) int main()
{
    printf("Welcome to C");
    return 0;
}
```

Method (i) (**Program 2.1**) and (iii) are perfectly correct but method (ii) is not, some will understand the difference between them at the end of Chapter 5 (Functions).

b) Earlier we learned that a C Program is nothing but a collection of functions. There can

be any number of functions in a program, **main()** is the compulsory function in a C program because it is both the **entry point** and **exit point** of a program

c) Parentheses are used to indicate a function.

d) A function contains a set of instructions enclosed with a pair of braces ({ and })

e) **printf()** is a library function which is used to print a message (with in double quotes) on

the screen.

f) **void** : it indicates that the main() function doesn't return any value.

g) Every instruction must be terminated with a semicolon.

How to Open Turbo C IDE on a 64-bit Windows OS

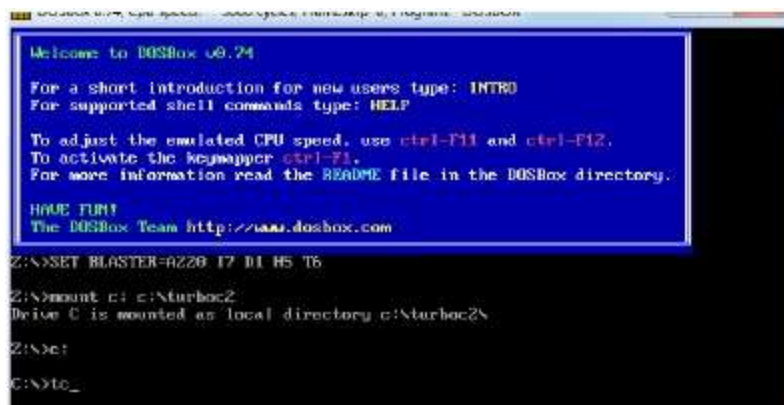
Note: Turbo C IDE is a 16-bit Application. A 16-bit application can be run a 32-bit Architecture, but not on a 64-bit Architecture, so its backward compatibility is only upto 32-bit architecture.

However we can use DOS Emulator software using which we can run a 16-bit application on a 64- bit Architecture. DOSBox is one such software

Step1. Download and install DOSBox - DOSBox0.74-win32-installer

Step2. From the Desktop shortcut icon, open it

Step3.



Step4. Modify the directories path:

From:



To



Because, now C: itself means C:\Turboc2

Note: For compilation, linking, execution and viewing the output, better use the menu options instead of ShortCut keys as some of them may collide with the shortcut keys of DOSBOX

What is DOSBOX?

DOSBox is an emulator that recreates a MS-DOS compatible environment. This environment is complete enough to run many classic MS-DOS applications completely unmodified. In order to utilize all of DOSBox's features you need to first understand some basic concepts about the MS-DOS environment.

TurboC++ 4.0 for windows 8(64 bit):

As we all know the Popular Turbo C 3 [C and C++ compiler by Borland] does not run well on 64 bit OS's like Windows 7, Windows 8 etc., and on 32 bit OS's they can't work full screen, so we have developed the emulated version of the same TurboC compiler within an environment called DosBoX which works on all OS's Full screen

How to use an Online IDE/Compiler

Note: You may be using a DOS based compiler like Turbo C IDE. You may want to test your program on a Windows based 32-bit compiler or a Ubuntu Linux based compiler, or you might have gone to some vacations to your relatives house, you got bored of playing, you want to revise your "C" Programming notes

In all such above situations , online Compilers are very handy provided you have an internet connection.

Eg: http://www.tutorialspoint.com/compile_c_online.php
It's a Linux based Compiler

C- Free (Evaluation version)

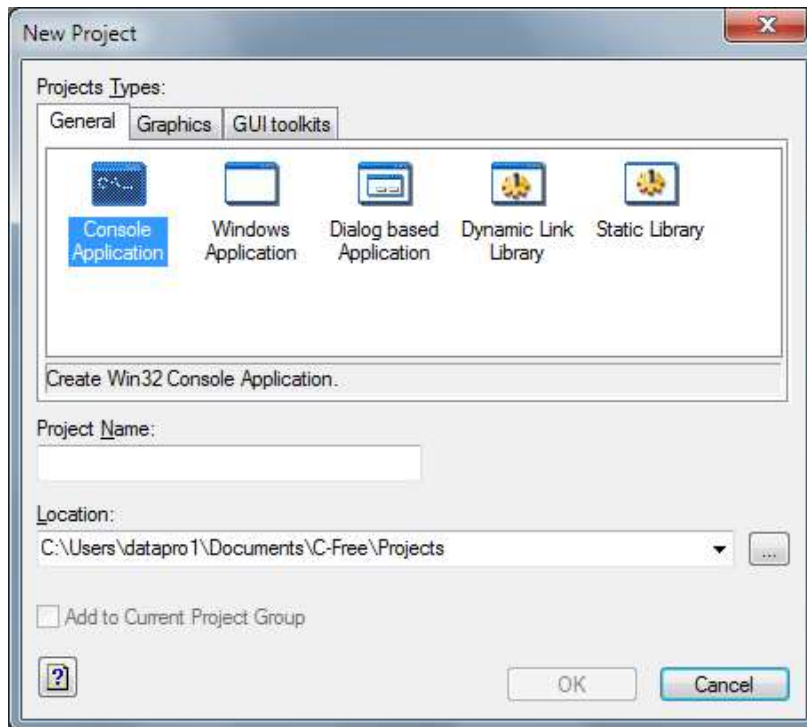


We have used C-Free Evaluation version editor to run all programs in this book. It is one of the best C IDE available online.

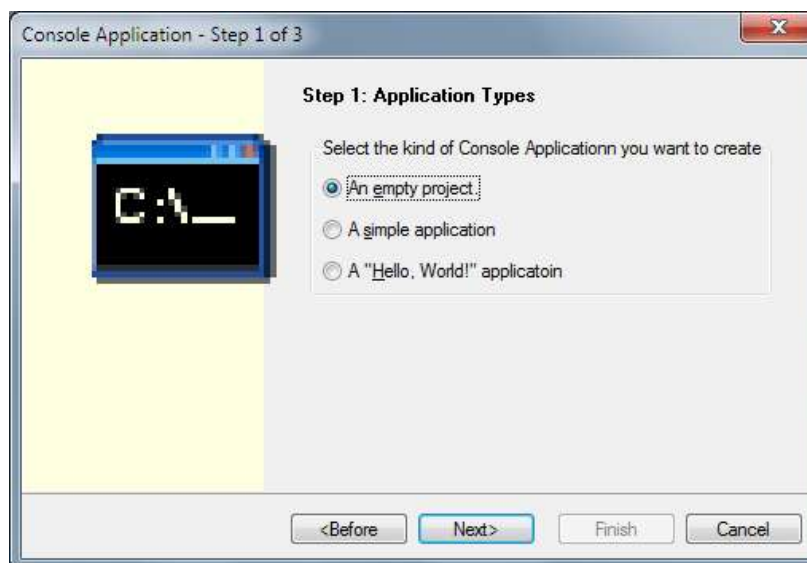
Use following steps to write any C program in C-Free editor

C-Free is a professional C/C++ integrated development environment (IDE) that supports multi-compilers. With this software, user can edit, build, run and debug programs freely. With C/C++ source parser included, although C-Free is a lightweight C/C++ development tool, it has powerful features to let you make use of it in your project.

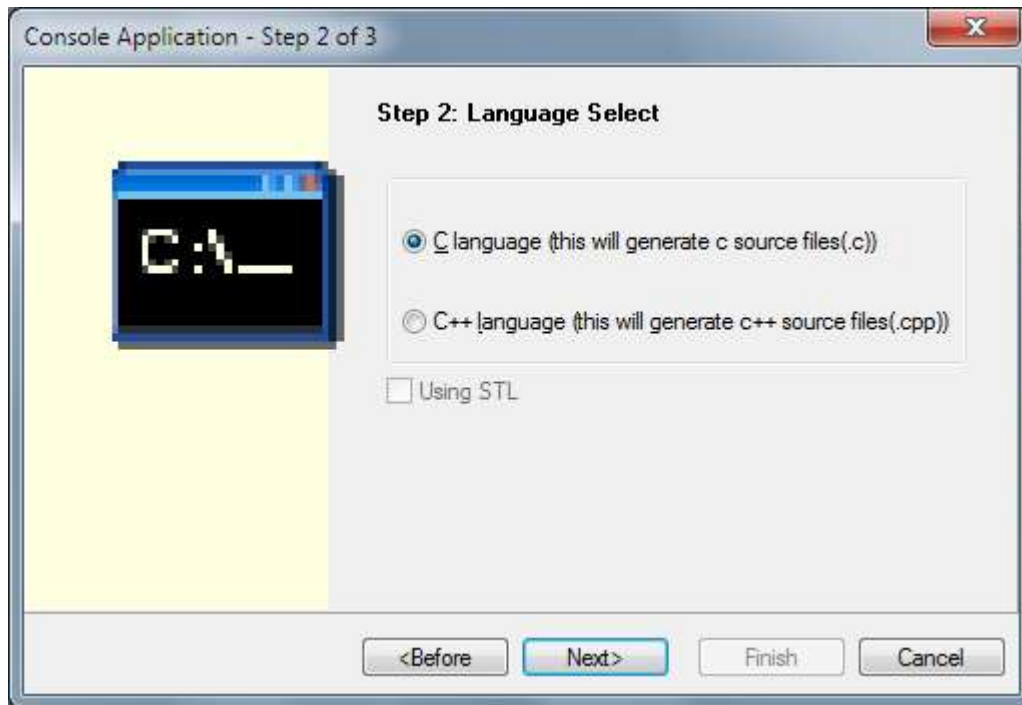
Select Project type as 'Console Application'



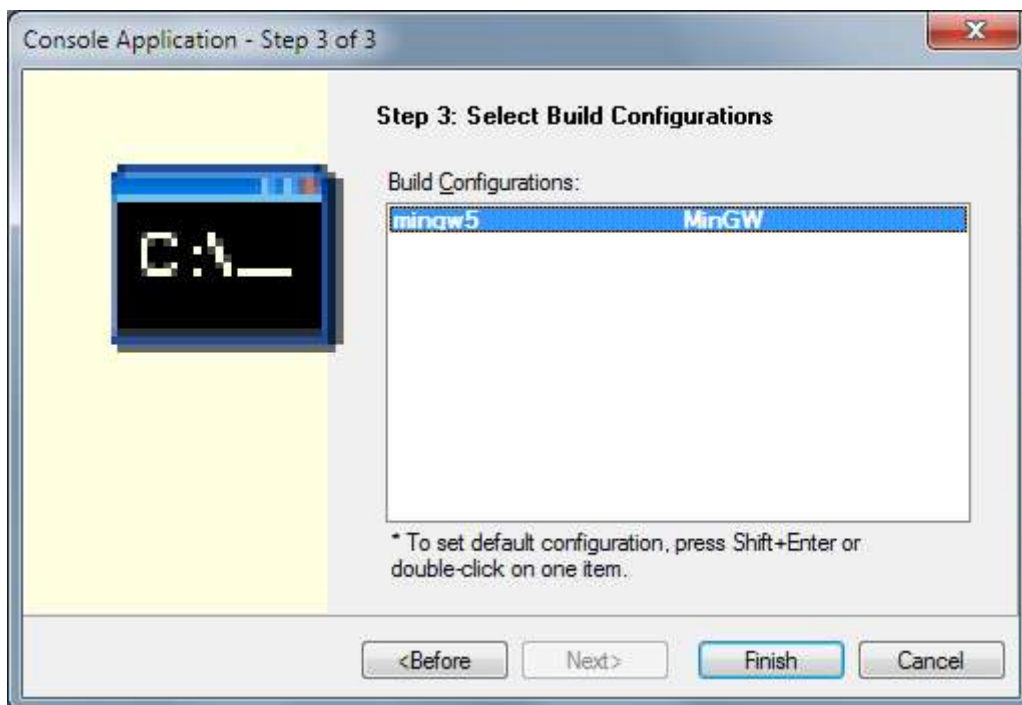
Choose Application Type as – A simple application



For Language Selection choose 'C Language'

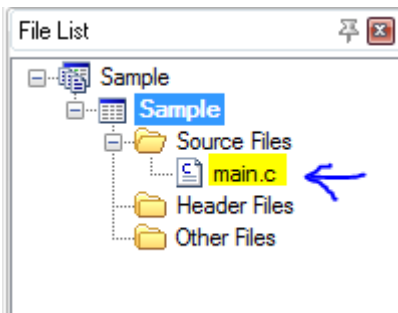


Build Configuration is mingw5



Click on Finish

Open main.c file

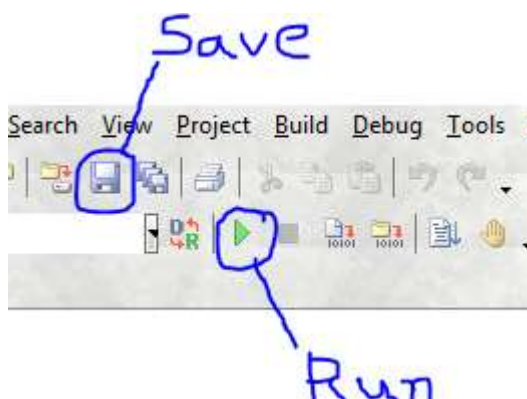


Write program code in editor

```
#include <stdio.h>

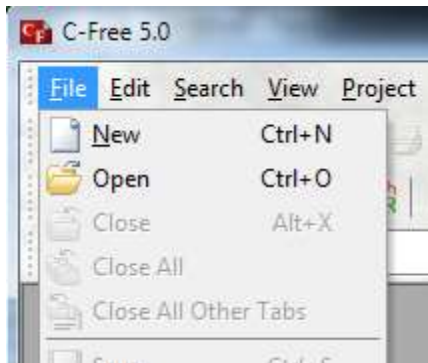
int main(int argc, char *argv[])
{
    return 0;
}
```

Save and Run the above program



Steps to create Individual Programs in C-Free

File -> New



Type program in editor

Save this file as .C file



Run the above program (F5)

Language Elements

- Identifier
 - Names of variables/functions
- Standard datatypes
 - To specify the type of data.
- Constants
 - Constant is an entity that doesn't change.
 - Ex: character, Integer, float, String

Basic data types are(turboC):(Vary from compiler to compiler)**

Data Type	Bytes	Format specifier	Range
int	2 or 4	%d	-32768 to +32767
char	1	%c	-128 to +127
float	4	%f	-
double	8	%lf	-

Note:

The difference between float and double is the precision.

float – 6 digits.

double – 15 digits.

2.5 Basic I/O operations

How to store data in the memory?

Any program is meant for processing some kind of input data & get desired results. The microprocessor device processes data. The microprocessor can process data only if the data is available in RAM (primary memory). To store any data in RAM we have to reserve required amount of memory.

Reading a number from the Keyboard, storing it in the memory & then printing it on the screen

Program 2.2

```
void main()
{
    int n;
    printf("Enter a number:");
    scanf("%d", &n);
    printf("%d", n);
}
```

output

Enter a number: 786

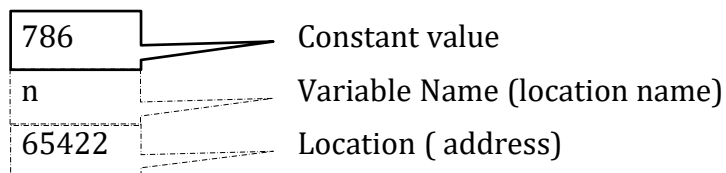
786

Explanation

a) `int n;`

this statement tells the compiler to reserve sufficient amount of memory to store an integer (number) value(2 bytes for an integer) and give a name 'n' to that memory location.

- **int** is called as datatype.
- '**n**' is the variable name
- **variable** is the name given to a memory location where a constant value is stored.



b) **scanf()** is a library function which is used to accept a data item from the keyboard & store it in the specified memory location.

c) **%d** called as format specifier. It tells the compiler that we are going to read a **decimal** number.

d) **&** is called as **address of** operator

e) **&n** tells the compiler to store the accepted value in the location whose name is n.

f) note that **&** is required only in the **scanf()** function, but not in the **printf()** function. Why? the reason you will understand at the end of pointers (chapter 7)

Reading a symbol (character) & print it**Program 2.3**

```
void main()
{
    char ch;

    printf("Enter a character:");
    scanf("%c", &ch);

    printf("%c", ch);
}
```

Output

Enter a character: y
y

Explanation

- a) **char** is a datatype
- b) Only one byte is allocated for a char type of value.
- c) Remember that characters are internally stored as integers.

Some Important ASCII values

a-z : 97 – 122
A-Z : 65 – 90
0-9 : 48 – 57

Read a fraction value and print it

Program 2.4

```
void main()
{
    float f;

    printf("Enter a fraction value:");
    scanf("%f", &f);

    printf("%f", f);
}
```

Output

Enter a fractional value: 12.79
12.79

Explanation

- a) **float** is a datatype
- b) Four bytes are allocated for a **float** type of value

Read a name and Print it

Program 2.5

```
void main()
{
    char na[30];

    printf("Enter a name:");
    scanf("%s", na);

    printf("%s", na);
}
```

Output

```
Enter a name: Raju
Raju
```

Explanation

- a) we know that name is a group of characters, hence **char** datatype is used. As it is a group, we have to mention its maximum size within square brackets. Now '**na**' is treated as an array (collection), we shall discuss arrays in detail in chapter 4.
- b) in **%s**, '**s**' stands for '**string**', which means **group of characters**.
- c) note that even in **scanf()** function we haven't used **&** (address of operator) Why? The reason you can understand at the end of the chapter 7 (pointers).

2.6 Rules for variable names

- It can contain : a -z, A - Z, 0 - 9 and _ (underscore)
- Can not begin with a digit
- Can not have any blanks within a variable name.
- Ideal maximum number of characters is eight. However it is compiler dependent. Turbo C compiler allows a maximum of 32 characters.
- Variable names must be unique within a block.
- Variable names cannot be keywords (pre defined words).
- It is must to define variables at the top of the block before the first executable statement.

2.7 Sum of two integers

Program 2.6

```
void main()
{
    int a, b, c;

    printf("Enter two numbers:");
    scanf("%d%d", &a, &b);

    c = a + b;

    printf("Sum is %d", c);
}
```

Output

```
Enter two numbers : 45
72
Sum is 117
```

Explanation

- a) '+' and '=' are operators in C, whereas 'a', 'b', 'c' are operands. Operand is an entity, which can be a variable/constant/ expression. Note that the left side of '=' operator can be only a variable.
- b) The value of a+b is assigned to the variable C

2.8 Keywords

Keywords are nothing but reserved or predefined words. This is the actual part of any language. (Kernel of the language). 'C' language has 32 keywords . These are shown below.

int	unassigned	do	auto
char	sizeof	break	static
float	const	continue	register
long	volatile	goto	extern
short	if	switch	struct
double	else	case	typedef
void	for	default	union
signed	while	return	enum

More about translators:

Translators are the programs which convert source code into machine code. The Different translators are.

- **Assemblers – for Assembly language**
- **Compilers – for HLL**
- **Interpreter – for 4GL's**

Compiler:

- The Whole program is compiled at once.
- A small change in the program requires whole program compilation.
- So, the Program can be executed only if it is translated.
- Ex: BASIC, COBOL, C, C++, Pascal, etc...

Interpreter:

- Code is translated line by line.
Interpreter program and source code program both are loaded in the memory at a time. No copy of translation code exists, if the program has to be executed again, it has to be interpreted again.
- Ex: VB, Small Talk, PERL, etc....

Conclusion:

Compiled code: Program development is slow, Execution is fast.

Interpreted Code: Program development and testing is fast, execution is slow.

2.9 Errors

There are four types of errors in “C” language

- Compile time errors
- Linking errors
- Logical errors
- Runtime errors

Compile time errors : These errors can occur during the compilation (infact debugging) which occurs just before compilation of a program. these errors occur if we violate ‘C’ language rules & regulations. only after correcting these errors the program can be compiled.

Examples

- a) Missing semicolon at the end of an instruction.
- b) Forgetting to define a variable.

Detecting & correcting these type of errors is very easy.

Linking errors

Usually program development in C, C++, Java etc languages, employs add-on strategy i.e., instead of developing everything from the beginning, we use some readymade components, so that our program development would be faster. These readymade components (in 'C' language known as library functions) are usually available in machine – code format (Object – code).

Linking is a process of combining the machine code versions of our source code with the machine code versions of the library functions.

If there is any problem during this process, then the linker software reports a linking error.

examples

- Mostly if there is any spelling mistake in the library function
- Detecting & correcting these type of errors is very easy as well.

Logical errors

These errors occur if there is any mistake in the logic of the program written.

Logic : is nothing but approach / planning towards solving a problem.

These errors are not displayed on the screen

What we get is incorrect results. We can identify that there is a logical error only if we know in advance what should be the result of a program.

Detecting & correcting these type of errors is very difficult.

Runtime errors

The errors, which occur during the execution of a program, are called as run time errors.

Examples

- i) Divide a number with zero.
- ii) Modifying illegal memory locations

Detecting & correcting these type of errors is very difficult as well.

A software success depends on how far the run time errors have been tackled. Hence 60 – 80 % of the program development time is spent on detecting & correcting runtime errors.

2.10 Typecasting

Let us consider the following program

Program 2.7

```
void main()
{
    int a, b;
    float c;

    printf("Enter two numbers:");
    scanf("%d%d", &a, &b);

    c = a / b;

    printf("Quotient = %f", c);
}
```

Output

```
Enter two numbers: 5
2
Quotient is 2.0
```

Explanation

The output is not as expected isn't it? We expected 2.5 but the result is 2.0 Why?

The rule is like this:

When an arithmetic operation takes place between two operands :

Operand 1	Operand 2	Result
Integer	integer	integer
integer	float	float
float	integer	float
float	float	float

So to a fraction result, one of the operand must be fraction value.

One solution is define a,b,c as **float**, but the problem if data type changes their properties too will change. the following program illustrates how to get a function result even with integer operands

Program 2.8

```
void main()
{
    int a, b;
    float c;

    printf("Enter two numbers:");
    scanf("%d%d", &a, &b);

    c = (float) a / b;    /*Type casting*/

    printf("Quotient = %f", c);
}
```

Output

```
Enter two numbers: 5
2
Quotient is 2.5
```

Explanation

c= (float) a/b;

When this statement gets executed, the value of 'a' is temporarily promoted to **float** value & have the output.

2.11 Operators

In the following table of operator precedence, the Turbo C operators are divided into 16 categories.

The #1 category has the highest precedence; category #2 (Unary operators) takes second precedence, and so on to the Comma operator, which has lowest precedence. The operators within each category have equal precedence.

The Unary (category #2), Conditional (category #14), and Assignment (category #15) operators associate right-to-left; all other operators associate left-to-right.

Priority No.	Category	Operator	What it is (or does)
1	Highest	() [] . ->	Function call Array subscript C direct component selector C indirect component selector
2	Unary	! ~ + - ++ -- & * sizeof	Logical negation (NOT) Bitwise (1's) complement Unary plus Unary minus Preincrement or postincrement Predecrement or postdecrement Address of Value at or Indirection Returns size of operand, in bytes
3	Multiplicative	* / %	Multiply Divide Remainder(modulus)
4	Additive	+ -	Binary plus Binary minus
5	Bitwise Shift	<< >>	Shift left Shift right
6	Relational	< <= > >=	Less than Less than or equal to Greater than Greater than or equal to
7	Equality	== !=	Equal to Not equal to
8	Bitwise	&	Bitwise AND
9	Bitwise	^	Bitwise XOR
10	Bitwise		Bitwise OR
11	Logical	&&	Logical AND
12	Logical		Logical OR
13	Conditional	?:	(a ? x : y means "if a then x, else y")

14	Assignment	= *= /= %= += -= &= ^= = <<= >>=	Simple assignment <u>Compound assignment</u> Assign product Assign quotient Assign remainder (modulus) Assign sum Assign difference Assign bitwise AND Assign bitwise XOR Assign bitwise OR Assign left shift Assign right shift
15	Comma	,	Evaluate

2.12 Modulus operator (%)

This is a binary operator (has two operands), which give the reminder. Remember that the operator should be of integral type only.

Program 2.9

```
void main()
{
    int a, b, c;

    printf("Enter two numbers:");
    scanf("%d%d", &a, &b);
    c = a % b;
    printf("Remainder is %d", c);
}
```

Output

Enter two numbers: 11

5

Remainder is 1

Note that the numerator value is smaller than the denominator value then the remainder is numerator itself

eg : a is 10, b is 12 then , c would be 10

2.13 clrscr(), getch()

clrscr() is a library function which clears the screen contents & positions the cursor at 0th row and 0th column of the screen

if we don't use **clrscr()** function, we may get confused identifying the result of our program as it will be mixed with the previous result.

getch() . It happens to be one more library function. It stands for **get a character** from the keyboard buffer area.

Use

Till now we have been viewing our output by pressing Alt + F5. Instead use **getch()** function at the end of the program which helps in displaying the output until the user presses any key.

Example

Suppose that we want to check the reminders of different sets of values, so every time, after executing the program we have to press Alt + F5 to view the result. **getch()** usage will avoid this frequent usage of ALT +F5

There are many other applications of **getch()** function which we will discuss later. The following program illustrates the usage of both these functions

Program 2.10

```
void main()
{
    int a, b, c;
    printf("Enter two numbers:");
    scanf("%d%d", &a, &b);
    c = a % b;
    printf("Remainder is %d", c);
    getch();
}
```

Explanation

Remember that variables must be defined at the top of a block. i.e., before the first executable statement in a block.

2.14 Escape sequence characters

These are special characters, which performs some formatting operations on the output to be displayed on the screen. The following program illustrates the usage of some of the escape sequences

some common escape sequences

Escape Sequence	Description
<code>\n</code>	new line . positions the cursor at the beginning of the next line
<code>\t</code>	Horizontal tab moves the cursor to the next tab stop.
<code>\a</code>	Alert . Sound the system bell
<code>\0</code>	null character (will be discussed in strings)
<code>\\</code>	back slash inserts a backslash character
<code>\"</code>	Quotation mark , inserts a quotation mark

The following program illustrates the usage of some of the escape sequences

Program 2.11

```
void main()
{
    int a, b;
    char name[20];

    printf("Enter two numbers:");
    scanf("%d%d", &a, &b);
    printf("Enter a name:");
    scanf("%s", name);

    printf("%d\t%d\n%s", a, b, name);
}
```

Output

```
Enter two numbers: 23
96
Enter a name: Arnold
23      96
Arnold
```

2.15 Initialization

Program 2.12

```
void main()
{
    int p = 93;
    printf("%d", p);
}
```

Output

93

Explanation

- Had we not initialized 'p' the output would have been some garbage (unknown) value
- If we know in advance, the initial value of a variable, then it's better to initialize it rather than putting a value using **scanf() function**

2.16 Unary operators

These are one of the most commonly used operators in 'C'. Unary operators means that the operator has only one operand.

Let us discuss the unary operator ++ and - operators in a program if a = 10 (the value of 'a' is 10)

To increase its value by 1, there are several methods.

- 1.) a = a+1
- 2.) a+=1
- 3.) a++ (post incrementation)
- 4.) ++a (pre incrementation)

a = 10

To decrease its value by 1, there are several methods

- 1.) a = a-1
- 2.) a-=1
- 3.) a-- (post decrementation)
- 4.) --a (pre decrementation)

Conclusion

We can find that `a++/ ++a` or `a--/--a` are more compact methods than the other methods.

You will soon understand about this post and pre incrementaion.

The following program illustrates the usage of these operators.

Program 2.13

```
void main()
{
    int a = 10, b = 17, c = 83, d = 24;

    a++;
    ++b;
    c--;
    --d;

    printf("%d, %d, %d, %d", a, b, c, d);
}
```

Output

11, 18, 82, 23

Explanation

Here we can notice that there is no difference between post and pre incrementation, but there is a slight difference between these two. the following two programs demonstrates this ;

Program 2.14

```
void main()
{
    int a = 5, b;

    b = a++;

    printf("%d, %d", a, b);
}
```

Output

6, 5

Explanation

`b = a++;` is same as `b = a; a++;`

i.e., first the value of 'a' is assigned to 'b' and then value of 'a' gets incremented

Program 2.15

```
void main()
{
    int a = 5, b;

    b = ++a;

    printf("%d, %d", a, b);
}
```

Output

6,6

Explanation

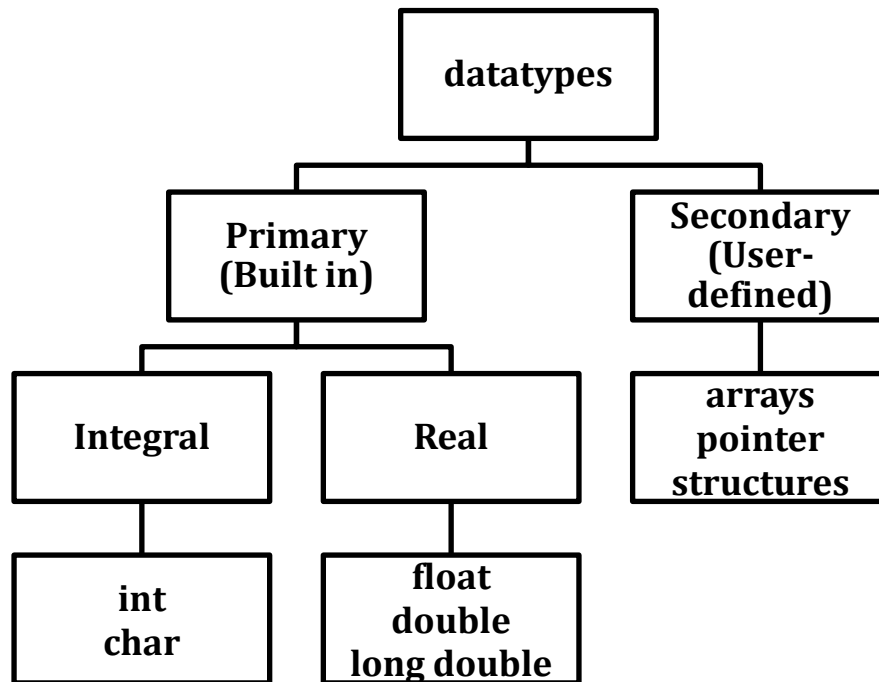
`b = ++a;` is same as `a = a + 1; b = a;`

i.e., first the value of 'a' gets incremented by 1, and then the incremented value gets assigned to 'b' So the difference between the post and pre incrementation / decrementation operators can be felt only when they are involved in an expression.

2.17 Datatypes

Till now we have been dealing with **int**, **char** & **float**. Let's discuss in detail A data type of a variable tells us three things:

- The type of value that can stored in a variable
- The range of values
- The type of operations that can be carried out.
- Data types are of two types



The following table illustrates various types of data types, respective format specifiers etc.

Datatype	Number of Bytes allowed	Format Specifier	Range of values
signed int	2	%d (or) %i (or) %o (or) %x	-32768 to +32767
unsigned int	2	%u	0 to +65535
signed char	1	%c (single) %s (string)	-128 to +127
unsigned char	1	%u	0 to 255
long int	4	%ld	-214,74,83,648 to +214,74,83,647
unsigned long int	4	%lu	0 to +429,49,67,295
float	4	%f or %e	$3.4 * 10^{-38}$ to $3.4 * 10^{+38}$
double	8	%lf or %le	$1.7 * 10^{-308}$ to $1.7 * 10^{+308}$
long double	10	%Lf or %Le	$3.4 * 10^{-4932}$ to $1.1 * 10^{+4932}$

Note:-

- There is one more datatype: short – which is relevant on 32-bit compilers only where “int” occupies 4 bytes, short occupies 2 bytes.
- Even characters are internally stored as integers (ASCII values)
- The important difference between float, double and long double is the precision they support.

float	6 digits
double	15 digits
long double	17 digits

2.18 Swapping

Swapping / interchanging of two values is one of the most common programming logics.

Program 2.16

```
void main()
{
    int a = 10, b = 20, t;

    printf("Before : %d, %d\n", a, b);

    t = a;
    a = b;
    b = t;

    printf("After : %d, %d", a, b);
}
```

output
20,10

2.19 Handling characters

To convert a lower case alphabet into its equivalent uppercase alphabet

Program 2.17

```
void main()
{
    char ch1, ch2;

    printf("Enter an alphabet in lowercase:");
    scanf("%c", &ch1);

    ch2 = ch1 - 32;

    printf("Equivalent uppercase alphabet is %c", ch2);
}
```

Output

```
Enter a alphabet in lowercase: b
Equivalent uppercase alphabet is B
```

Explanation

- we know that the difference between equivalent uppercase & lowercase characters is 32. also uppercase character is 32 less than its equivalent lowercase character. Hence the formula is $ch2 = ch1 - 32$
- There is one more method $ch2 = toupper(ch1)$; **toupper()** is a library function which returns an equivalent uppercase character.

2.20 Coding the formulae

In C formulae can be easily coded. the following program converts temperature value from Centigrade to Fahrenheit.

Program 2.18

```
void main()
{
    float cen, fah;

    printf("Enter temperature in centigrade scale:");
    scanf("%f", &cen);

    fah = 9/5.0 * cen + 32;

    printf("Equivalent Fahrenheit scale value is %f", fah);
}
```

Output

Enter temperature in centigrade scale: 27.5
Equivalent Fahrenheit scale value is 81.5

2.21 sizeof keyword

It is used to get the size of a variable or data type i.e., number of bytes allocated in memory. The value returned by sizeof is depends on compiler for int datatype.

TurboC compiler by default takes short as default modifier for int. So, it gives 2 Bytes. Whereas C-Free compiler takes long as default modifier for int datatype. So it gives 4 Bytes.

You can see this difference by executing below program in both compilers.

Program 2.19

```
void main()
{
    int x;
    char y;
    float z;

    printf("x=%d\n", sizeof( x ) );
    printf("int =%d\n", sizeof( int ) );

    printf("y=%d\n", sizeof( y ) );
    printf("char=%d\n", sizeof( char ) );

    printf("z=%d\n", sizeof( z ) );
    printf("float =%d\n", sizeof( float ) );
}
```

output

```
x=2
int =2
y=1
char=1
z=4
float =4
```

2.22 *const* keyword

If we want that the value of a variable is fixed & should not be changed even accidentally, then such a variable is defined as **const**

Program 2.20

```
void main()
{
    const int a = 512;

    printf("%d", a);
}
```

Output : 512

Explanation

Any where in the program if we use 'a' in the left hand side of '=' (assignment operator), the compiler reports an error message.

2.23 The four files

If we use **Turbo C IDE** by the time we execute a program, four files would have been created.

On saving → **.C** (the source code file)

On updating → **.bak** (the back up file)

On compiling → **.obj** (**object file** , the compiled version of our source code)

On linking → **.exe** (**executable file**. this is the file which finally gets executed and this is the file that will be installed in the target computer)

Exercise

State True or False

- a) 'C' language is a middle level language
- b) A **variable** is the name given to a memory location
- c) ASCII value of z is 122
- d) **main()** is the compulsory function of a 'C' program.
- e) Full stop is used to terminate an instruction in 'C' language.

Control Structures

Topic # 3

Atish Jain

3. Control Structures

3.1 What are Control Structures?

Till now we have been dealing with sequential program execution, i.e., every statement in the program would be executed sequentially. It means that :

- A statement can be executed only after the execution of its previous statement.
- Every statement in the program will be executed.
- Each statement will be executed only once.

However, many a times we need to control the sequence of execution of the instructions, for that purpose we have use control structures.

Control structures in C are classified into three categories:

- Decision control structures
- Loop control structures
- Case control structures

Lets discuss one by one.

3.2 Decision Control Structures

These are used to express decisions based on conditions.

We all need to change our actions based on the changing circumstances. Lets take some real-life examples:

- a) If we win the match we would celebrate
- b) If the highway is free, I would go in 90s (kmph)
- c) If the food is tasty then I would eat more.
- d) If it rains I would go by bus.

A program too must be able to perform different sets of actions depending on the circumstances.

The mechanism, which meets this need, is the decision control structures.

Decision control structures may be further classified into three types:

- Uni - directional Decision control structure
- Bi - directional Decision control structure
- Multi- directional Decision control structure

Comparison operators:

>	Greater than
<	Less than
==	Equal to
>=	Greater than equal to
<=	Less than equal to
!=	Not equal to

Note:

= is used for Initialization whereas == is used for Comparison.

Logical operators:

Logical operators are used to combine the conditions.

&&	And
	Or
!	Not

if and else are the two keywords which are used to construct selection type of programming.

Uni – directional Decision control structure

Whenever we need to perform a particular action when a condition gets satisfied, then we use uni – directional decision control structure. It is implemented through the keyword '*if*'

Syntax (general form)

```
if ( condition )  
{  
    statements  
    ...  
    ...  
    ...  
}
```

If the condition is **TRUE**, the block associated with '*if*' gets executed.

Note If only one statement is associated with an '*if*' expression then enclosing that statement within a pair of braces is not compulsory.

Example 1

Program 3.1

```
void main()
{
    char ch;

    printf("Got the ticket?[y/n]:");
    scanf("%c",&ch);

    if(ch == 'y')
        printf("Enjoy!");
}
```

Output

```
Got the ticket?[y/n]:y
Enjoy!
```

Explanation

If the user presses the character 'y' then the '*if*' expression (condition) becomes **TRUE** (satisfied). So the statement associated with it gets executed. If the user presses other than 'y' then the output would be nothing.

Remember that there should not be any semicolon at the end of the '*if*' expression.

Example 2

Program 3.2

```
void main()
{
    int n;

    printf("Enter a number:");
    scanf("%d", &n);

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

    if(n == 786)
        printf("Bismillah");
}
```

Bi – directional Decision control structure

Whenever we need to perform a particular action when a condition gets satisfied and perform a different action if the condition doesn't get satisfied, then we use bi-directional decision control structure. It is implemented through the keywords '*if*' and '*else*'

Syntax

```
if ( condition )
{
    statements
    ...
    ...
    ...
}
else
{
    statements
    ...
    ...
    ...
}
```

If the condition is TRUE, the statements associated with 'if' block gets executed, otherwise the statements associated with the '**else**' block get executed.

Example 1**Program 3.3**

```
void main()
{
    int m;

    printf("Enter marks:");
    scanf("%d", &m);

    if(m >= 35)
        printf("Passed");
    else
        printf("Failed");
}
```

Output

Enter marks : 23
Failed

Example 2

Program 3.4

```
void main()
{
    int n;

    printf("Enter a number:");
    scanf("%d", &n);

    if(n % 2 == 0)
        printf("Even");
    else
        printf("Odd");
}
```

Multi – directional Decision control structure

These are used for multi – way decision making.

Syntax

```
if ( condition )
{
    statements
    ...
    ...
    ...
}
```

```
else if ( condition )
{
    statements
    ...
    ...
    ...
}
else if ( condition )
{
    statements
    ...
    ...
    ...
}
else if ( condition )
{
    statements
    ...
    ...
    ...
}
...
...
...
```

<pre>else { Statements }</pre>	<hr/>	optional
--	-------	-----------------

The conditions are evaluated in order, if any expression is **TRUE**, the statements associated with it are executed, and this terminates the whole chain. The last ***else*** part gets executed when none of the other conditions are satisfied. Sometimes there is no action for the default, in that case the trailing ***else*** block can be omitted.

Example 1

Program 3.5

```
void main()
{
    int m;

    printf("Enter marks:");
    scanf("%d", &m);

    if( m >= 60 )
        printf("First class");
    else if( m >= 50 )
        printf("Second class");
    else if( m >= 35 )
        printf("Third class");
    else
        printf("Failed");
}
```

Output

Enter marks : 46
Third class

Example 2

Program 3.6

```
void main()
{
    int n;

    printf("Enter a number:");
    scanf("%d", &n);

    if( n > 0)
        printf("Positive");
    else if( n < 0)
        printf("Negative");
    else
        printf("Zero");
}
```


3.3 Loop Control Structures

Loop control structures can also be called as **iterative** or **repetitive** control structures.

Whenever we need to repeat a set of instructions, certain number of times, then we use loop control structures.

In **C** we have three types of loops :

- ***for***
- ***while***
- ***do - while***

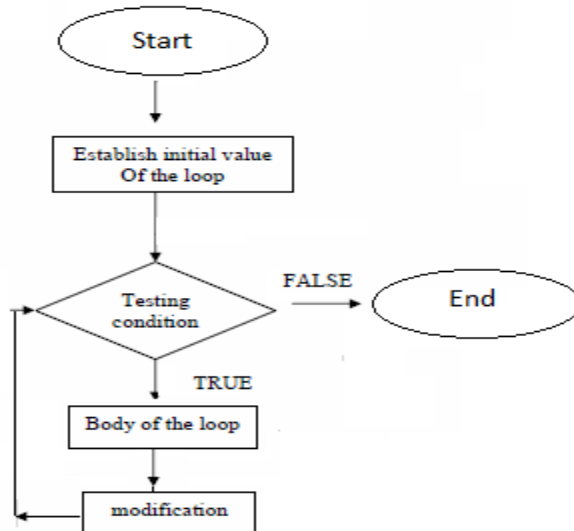
All of them are keywords.

For any loop we usually specify three things:

Initialization counter : It tells us the initial value of the loop.

Condition testing counter: It tells us how long a loop will continue.

Modification counter : It tells us the modification that takes place during each repetition of the loop.

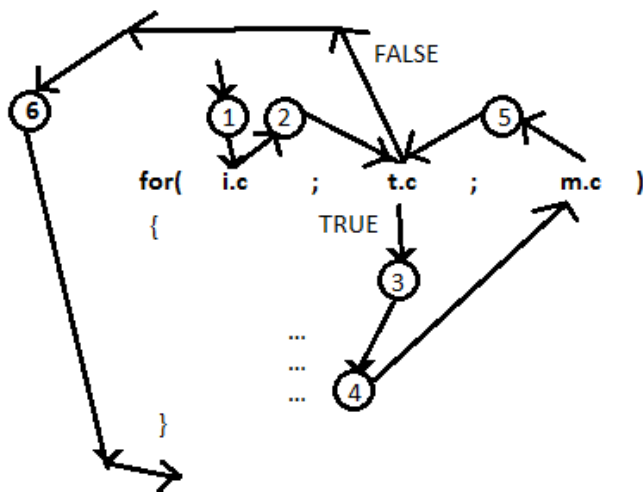


Lets start with the ***for*** loop:

Syntax

```
for (initialization counter ; condition testing counter ; modification counter)
{
    statements to be repeated
    ...
    ...
    ...
}
```

Control flow within a for loop



Lets understand its working with the help of a flow chart (pictorial representation of a control flow).

The **for** loop runs as long as the testing condition is **TRUE**.

Example 1

Program 3.7

```
void main()
{
    int i;

    for(i = 1 ; i <= 100 ; i++)
        printf("Hello\t");
}
```

Output

“Hello” gets printed 100 times

Explanation

a) If there is only statement associated with a loop then no need of enclosing it within the braces.

b) The loop runs till ‘i’ value is ≤ 100 . So when ‘i’ becomes 101 the loop terminates.

c) Note that there should not be any semicolon at the end of the *for* loop expression.

Example 2

To print all the even numbers between 1 to 100

Program 3.8

```
void main()
{
    int i;

    for(i = 1 ; i <= 100 ; i++)
    {
        if( i % 2 == 0)
            printf("%d\t", i);
    }
}
```

A program, which read a number and generates its multiplication table

Program 3.9

```
void main()
{
    int n, i;

    printf("Enter a number:");
    scanf("%d", &n);

    for(i = 1 ; i <= 20 ; i++)
        printf("%3d * %2d = %5d\n", n, i, n*i);
}
```

called as width specifier (numbers are aligned towards right)

Output

Enter a number : 9

```
9* 1= 9
9* 2= 18
9* 3= 27
9* 4= 36
9* 5= 45
9* 6= 54
9* 7= 63
9* 8= 72
9* 9= 81
9 * 10 = 90
9 * 11 = 99
9 * 12 = 108
9 * 13 = 117
9 * 14 = 126
9 * 15 = 135
9 * 16 = 144
9 * 17 = 153
9 * 18 = 162
9 * 19 = 171
9 * 20 = 180
```

Nested loops

A loop inside another loop is called as nested loop.

Program 3.10

```
void main()
{
    int i, j;

    for(i = 1 ; i <= 5 ; i++)
    {
        for(j = 1 ; j <= 3 ; j++)
        {
            printf("%d, %d\n", i, j);
        }
    }
}
```

Output

```
1, 1
1, 2
1, 3
2, 1
2, 2
2, 3
3, 1
3, 2
3, 3
4, 1
4, 2
4, 3
5, 1
5, 2
5, 3
```

Explanation

- a) When the control reaches the inner loop, only after the complete execution of the inner loop, the control goes to the modification counter of the outer loop.
- b) Within the inner loop, the outer loop's counter variable's value remains constant, whereas the inner loop's counter variable's value keeps changing.

A program which read a number and generates that many multiplication tables

Program 3.11

```
void main()
{
    int n, i, j;

    printf("Enter a number:");
    scanf("%d", &n);

    printf("\t");
    for(i = 1 ; i <= n ; i++)
        printf("%5d", i);

    printf("\n\n");
}
```

```
    for(i = 1 ; i <= 20 ; i++)
    {
        printf("%d\t", i);
        for(j = 1 ; j <= n ; j++)
        {
            printf("%5d", i * j);
        }
        printf("\n");
    }
}
```

Output

Enter a number : 5

	1	2	3	4	5
1	1	2	3	4	5
2	2	4	6	8	10
3	3	6	9	12	15
4	4	8	12	16	20
5	5	10	15	20	25
6	6	12	18	24	30
7	7	14	21	28	35
8	8	16	24	32	40
9	9	18	27	36	45
10	10	20	30	40	50
11	11	22	33	44	55
12	12	24	36	48	60
13	13	26	39	52	65
14	14	28	42	56	70
15	15	30	45	60	75
16	16	32	48	64	80
17	17	34	51	68	85
18	18	36	54	72	90
19	19	38	57	76	95
20	20	40	60	80	100

***while* loop**

Syntax

i.c

```
while( t.c )
{
    statements to be executed
    ...
    ...
    ...
    m.c
}
```

Example 1

To print 1 to 10 numbers

Program 3.12

```
void main()
{
    int i;

    i = 1;

    while( i <= 10 )
    {
        printf("%d\n", i);

        i++;
    }
}
```

Example 2

To find the reverse of the given number

Program 3.13

```
void main()
{
    int n, r, rev = 0;

    printf("Enter a number:");
    scanf("%d", &n);

    while( n != 0)
    {
        r = n%10;

        rev = rev * 10 + r;

        n = n/10;
    }

    printf("Reverse : %d", rev);
}
```

Output

```
Enter a number: 786
Reverse : 687
```

Explanation

The logic is extract each and every digit, starting from the last digit, multiply it by 10 and find the running sum.

You should know that when we divide a number by 10 the remainder is the last digit of that number. You should also know that when we divide a number by 10 the quotient is except the last digit.

To understand the program, we have to analyze it.

Analysis

n	r	rev
		0
786	6	6
78	8	68
7	7	687
0		

Few more logic programs

Program 3.14

```
void main()
{
    int n, r, sum = 0;

    printf("Enter a number:");
    scanf("%d", &n);

    while( n != 0)
    {
        r = n%10;

        sum = sum + r;           /*called as running sum*/

        n = n/10;
    }

    printf("Sum of digits is %d", sum);
}
```

Output

```
Enter a number : 786
Sum of digits is 21
```

Note : While performing a running sum we have to initialize the variable with zero(0).

A program which reads a number and finds whether the given number is Armstrong number or not

Armstrong number: If the sum of cubes of each and every digit is equal to the given number, then such a number is called as an Armstrong number.

Eg : 153, 371

Program 3.15

```
void main()
{
    int n, r, sum = 0, t;

    printf("Enter a number:");
    scanf("%d", &n);

    t = n;
    while( n != 0)
    {
        r = n%10;
        sum = sum + r * r * r;
        n = n/10;
    }

    if( t == sum )
        printf("armstrong number");
    else
        printf("not an armstrong number");
}
```

Explanation

The significance of the statement,

t = n;

by the time the control terminates the **while** loop, '**n**' would be zero. Hence to store the original value of '**n**' we used the above statement.

To find whether the given number is prime or not

Prime number: a number, which is divisible by one and itself, is called as a prime number. i.e., a prime number has only two factors, one and itself.

Program 3.16 (a)
(Method-1)

```
void main()
{
    int i, n, count = 0;

    printf("Enter a number:");
    scanf("%d", &n);

    for(i = 1 ; i <= n ; i++)
        if( n % i == 0)
            count++;

    if(count == 2)
        printf("prime");
    else
        printf("not prime");
}
```

Explanation

We know that the prime number is divisible by one and itself only, that's why we divided the given number with every number within the range of one and itself and counted the number of factors it have. We know that prime number have only two factors and hence the condition.

Program 3.16 (b)
(Method-2)

```
void main()
{
    int i, n, count = 0;

    printf("Enter a number:");
    scanf("%d", &n);

    for(i = 2 ; i < n ; i++)
        if( n % i == 0)
            count++;

    if(count == 0)
        printf("prime");
    else
        printf("not prime");
}
```

Explanation

We know that any number is divisible by one and itself, that's why we have avoided dividing with those two numbers, thereby avoiding two runs.

Program 3.16 (c)

(Method-3)

```
void main()
{
    int i, n, count = 0;

    printf("Enter a number:");
    scanf("%d", &n);

    for(i = 2 ; i <= n / 2 ; i++)
        if( n % i == 0)
            count++;

    if(count == 0)
        printf("prime");
    else
        printf("not prime");
}
```

Explanation

We know that the factor of a number cannot be greater than half of that number and hence the code.

So this method is the most efficient of the three methods.

Program 3.16 (d)

(Method-4)

```
void main()
{

    int i, n, flag = 0;

    printf("Enter a number:");
    scanf("%d", &n);
```

```
    for(i = 2 ; i <= n / 2 ; i++)
    {
        if( n % i == 0)
        {
            flag = 1;
            break;
        }
    }

    if(flag == 0)
        printf("prime");
    else
        printf("not prime");
}
```

Explanation

As soon as a factor is encountered (between 2 and $n/2$), we can stop the iteration, there is no point in keep on counting the number of factors. To stop the loop we can use “break” statement. We will it more details in the Jump Statements section.

A program which finds the factorial value of a number

Program 3.17 (a)**(Method-1)**

```
void main()
{
    int n, i, f = 1;

    printf("Enter a number:");
    scanf("%d", &n);

    for(i = 1 ; i <= n ; i++)
        f = f * i;                                /*running product (initialize with 1)*/
    printf("Factorial : %d", f);
}
```

Output

```
Enter a number : 5
Factorial : 120
```

Program 3.17 (b)

(Method-2)

```
void main()
{
    int n, f = 1;

    printf("Enter a number:");
    scanf("%d", &n);

    while( n )
    {
        f = f * n;
        n--;
    }

    printf("Factorial = %d", f);
}
```

Explanation

In C language's point of view, any +ve or -ve value is treated as **TRUE** whereas 0(zero) is treated as **FALSE**. So the **while** loop runs as long as the value of 'n' is non – zero.

Program 3.17 (c)

(Method-3)

```
void main()
{
    int n, f = 1;

    printf("Enter a number:");
    scanf("%d", &n);

    while( n )
        f = f * n--;

    printf("Factorial = %d", f);
}
```

Explanation

This method is the most compact one. We know that, if there is only one instruction associated with a loop, then no need of enclosing it within the braces.

A Program which reads a number and prints that many numbers from the fibonacci series. Fibonacci series : 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ...

Program 3.18

```
void main()
{
    int n, i, a = 0, b = 1, c;

    printf("Enter a number:");
    scanf("%d", &n);

    printf("%d\t%d\t", a, b);

    for(i = 1 ; i <= n - 2 ; i++)
    {
        c = a + b;
        printf("%d\t", c);

        a = b;
        b = c;
    }
}
```

Infinite *while* loop**Program 3.19**

```
void main()
{
    int i;
    i = 1;
    while( 1 )
    {
        printf("%d\t", i);
        i++;
    }
}
```

Output

```
1      2      3      4      ... 10000 ... 30000 ... 32500 ... 32767      -32768 -32767 -
32766
-32765 ... -100 ... -1      0      1      2      3 ... infinite times
```

Explanation

We know that 'i' can hold a number max of **32767** and a min of **-32767**. Hence 'i' value oscillates between this range.

Now how to break the infinite loop?

Just press **Ctrl + break** to terminate the program.

There are many advantages of infinite loop, very soon you will understand it.

do – while loop

Syntax

```
i.c

do
{
    statements to be executed
    ...
    ...
    ...
    m.c
}while(t. c);
```

Note that there is a semicolon at the end of **while(t.c)** expression.

A program which prints 1 to 10 numbers

Program 3.20

```
void main()
{
    int i;

    i = 1;
```



```
do
{
    printf("%d\n", i);

    i++;

}while( i <= 10);

}
```

3.4 Jump statements

Whenever we need to transfer the control from one place to another (distance) place in a program then we use jump statements.

C supports three jump statements:

break
continue
goto

All three of them happens to be keywords.

***break* statement**

Sometimes there may be a situation where we want immediate termination of a loop, bypassing the testing condition expression and any remaining statements in the body of the loop. Under such circumstances we use a ***break*** statement. When a ***break*** statement is encountered inside a loop, the control comes out of that loop and resumes at the next statement following the loop.

Example 1

Program 3.21

```
void main()
{
    int n, max = 0;
    char ch;
    while( 1 )
    {
        printf("Enter a number:");
        scanf("%d", &n);
```

```
        if(n > max)
            max = n;

        printf("Another number[y/n]:");
        ch = getch();

        if(ch != 'y')
            break;
    }

    printf("Maximum : %d", max);
}
```

Explanation

a) The loop continues as long as the user presses 'y'. Now you would appreciate the use of infinite loop.

b) **ch = getch();**

getch() is an exclusive function for reading a character. Till now you might be using this just to avoid pressing **ALT + F5** to view the output.

getch() reads a character a character and stores it in 'ch'

c) **difference between getch(), getche(), getchar() functions**

getche() is same as **getch()** function, except that it echoes the pressed key. i.e., the pressed character is visible **getchar()** is same as **getche()** function, except that pressing the enter key is compulsory to finish the task So externally **scanf()** and **getchar()** performs the same job. The difference is the later executes faster than the former one as the later one tackles only character input.

Example 2

Program 3.22

```
void main()
{
    int n, i;

    for(i = 1 ; i <= 10 ; i++)
    {
        printf("Enter a number:");
        scanf("%d", &n);
    }
}
```

```
        if( n % 2 == 0)
            break;
    }
}
```

***continue* statement**

When a ***continue*** statement is encountered, in a loop, the control skips the execution of the remaining statements in that loop and continues with the next iteration of the loop.

Example 1

Program 3.23

```
void main()
{
    int i, j;

    for(i = 1 ; i <= 5 ; i++)
    {
        for(j = 1 ; j <= 3 ; j++)
        {
            if( i == j)
                continue;

            printf("%d, %d\n", i, j);
        }
    }
}
```

Output

```
1, 2
1, 3
2, 1
2, 3
3, 1
3, 2
4, 1
4, 2
4, 3
```

5, 1
5, 2
5, 3

Example 2

Program 3.24

```
void main()
{
    int i;

    for(i = 1 ; i <= 100 ; i++)
    {
        if( i % 9 == 0)
            continue;

        printf("%d\t", i);
    }
}
```

Output

Except the multiples of 9

We shall discuss *goto* statement in the later part of this chapter.

3.5 Conditional operator (?:)

It is also called as trinary operator. i.e., it has three operands. Conditional operator can be used as an alternative form of bi-directional decision control structure.

There are two forms of conditional operators.

Form 1

(expr1)? (expr 2) : (expr 3) ;

If **expr1** is a **TRUE** value, **expr2** gets executed otherwise **expr3** gets executed.

Form 2

Variable = (expr1) ? (expr 2) : (expr 3) ;

If **expr1** is a **TRUE** value, the variable gets the value from **expr2** otherwise from **expr3**

Example 1

Program 3.25

```
void main()
{
    int m;

    printf("Enter marks:");
    scanf("%d",&m);

    ( m >= 35 ) ? printf("Passed") : printf("Failed") ;
}
```

Example 2

Program 3.26

```
void main()
{
    int a, b, max;

    printf("Enter two numbers:");
    scanf("%d%d", &a, &b);

    max = ( a > b ) ? a : b;

    printf("Maximum : %d", max);
}
```

3.6 Logical operators

There are three logical operators in C

&& (AND)
|| (OR)
! (NOT)

The result of a logical operation is always either **TRUE** or **FALSE** . i.e., 1 (one) or 0 (zero)

Logical && operator

It is a binary operator, which takes two operands. The operands can be variables/ constants/expressions.

Truth table

A	B	A&&B
T	T	T
T	F	F
F	T	F
F	F	F

So a logical && (**AND**) operation can be **TRUE** only if both the operands are **TRUE** values.

Example

Program 3.27

```
void main()
{
    int m1, m2, m3;

    printf("Enter marks in three subjects:");
    scanf("%d%d%d", &m1, &m2, &m3);

    if( m1 >= 35 && m2 >= 35 && m3 >= 35 )
        printf("Passed");
    else
        printf("Failed");
}
```

Explanation

Remember that while performing a logical **AND** operation, if the left operand is a **FALSE** value, the right operand will not be evaluated (Since not required).

Logical || (OR) operator

It is a binary operator, which takes two operands. The operands can be variables/ constants/ expressions.

Truth table

A	B	A B
T	T	T
T	F	T
F	T	T
F	F	F

So a logical || (**OR**) operation can be **TRUE** if at least one of the operands is a **TRUE** value.

Example**Program 3.28**

```
void main()
{
    char ch;

    printf("Enter an alphabet:");
    scanf("%c", &ch);

    if( ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u' )
        printf("Vowel");
    else
        printf("Consonant");
}
```

Explanation

Remember that while performing a logical **OR** operation, if the left operand is a **TRUE** value, the right operand will not be evaluated (Since not required).

Note : Logical && and logical || are usually used to join more than one condition.

Logical ! (NOT) operator

It is a unary operator, So it takes only one operand. The operand can be variables/ constants/expressions.

Truth table

A	!A
T	F
F	T

If the operand is a **TRUE** value, it returns **FALSE** (0), otherwise returns **TRUE** (1).

Program 3.29

```
void main()
{
    int a = 10, b = -8, c = 0, i, j, k, l;

    i = !a;
    j = !b;
    k = !c;
    l = ! ( a == 10);

    printf("%d, %d, %d, %d", i, j, k, l);
}
```

Output

0, 0, 1, 0

3.7 switch - case

Whenever we need to select an option from a set of options, we use **switch- case** control structure.

It can be thought of as an alternative form to multi-directional decision control structure. However **switch-case** is a better alternative under the situations where we check a value against a set of values.

Syntax

```
switch ( expr )
{
    case constant1 : statements
        ...
        ...
        ...
        break;
    case constant2 : statements
        ...
        ...
        ...
        break;
    case constant3 : statements
        ...
        ...
        ...
        break;

    ...
    ...
    ...

    default : statements
}
```

First, the expression following the **switch** – keyword is evaluated. The resultant value is then matched, one by one, against the constant values that follow the **case** keywords, when a match is found the statements associated with that **case** gets executed and the control terminates from the entire **switch** structure (when **break** is encountered in between). If a match is not found, the statements associated with the **default**, gets executed.

Example 1**Program 3.30**

```
void main()
{
    int n;

    printf("Enter a number:");
    scanf("%d", &n);
```

```
switch( n )
{
    case 1 : printf("I am in case 1");
             break;
    case 2 : printf("I am in case 2");
             break;
    case 3 : printf("I am in case 3");
             break;
    case 4 : printf("I am in case 4");
             break;
    case 5 : printf("I am in case 5");
             break;
    default : printf("I am in default");
}
}
```

Output

Enter a number: 3
I am in case 3

Example 2

Program 3.31

```
void main()
{
    char ch;

    printf("Enter an alphabet:");
    scanf("%c", &ch);
    switch( ch )
    {
        case 'c' : printf("Chandigarh");
                    break;
        case 'b' : printf("Baroda");
                    break;
        case 'v' : printf("Vizag");
                    break;
        case 'n' : printf("Nagpur");
                    break;
        default : printf("Guesswhere?");
    }
}
```

3.8 Fall through

Lets consider the following example:

Program 3.32

```
void main()
{
    char ch;

    printf("Enter an alphabet:");
    scanf("%c", &ch);

    switch( ch )
    {
        case 'a' :
        case 'e' :
        case 'i' :
        case 'o' :
        case 'u' : printf("Vowel");
                  break;
        default : printf("Consonant");
    }
}
```

Example

Here, we are making use of a fact that once a **case** is satisfied, the control simply **falls through** the **case** till it doesn't encounter a **break** statement. So fall through is used, when in more than once **case** the task to be done is same.

3.9 goto

goto statement can be used to move the control from one place to another place in a function.

This keyword is derived from C language's predecessors.

goto statement is usually used to **abandon processing in some deeply nested structure**, such as breaking out of two or more loops at once. The break statement cannot be used directly, since it only exits from the innermost loop.

Lets consider the following program:

Example 1**Program 3.33**

```
void main()
{
    int i, j, k;

    for(i = 1 ; i <= 3 ; i++)
    {
        for(j = 1 ; j <= 3 ; j++)
        {
            for(k = 1 ; k <= 3 ; k++)
            {
                if( i == 2 && j == 2 && k == 2 )
                    goto allout;

                printf("%d, %d, %d\n", i, j, k);
            }
        }
    }

    allout:
        printf("out of all the nested loops");
}
```

Explanation

goto statement is usually associated with a label. In our program “**allout**” is called as label. A label has the same form as a variable name, & is followed by a colon. It can be attached to any statement in the same function where **goto** is present.

Example 2

goto can be used as an alternate to a loop.

Program 3.34

```
void main()
{
    int n, m, i, max = 0;

    printf("How many numbers do u want to enter:");
    scanf("%d", &n);

    i = 0;
```

```
readanumber :
    printf("Enter a number:");
    scanf("%d", &m);

    if( m > max)
        max = m;

    i++;

    if( i < n )
        goto readanumber;

    printf("Maximum : %d", max);
}
```

Explanation

While developing a program, if we use too many ***gotos*** (even if the need can be fulfilled with other control structures, more simpler), maintenance of the program becomes very difficult. We can never be sure how we got to a certain point in our code. Hence avoid using ***goto***.

3.10 Comments

Any message within `/*` and `*/` is a comment. The contents of a comment are ignored by the Compiler (i.e., not Compiled)

Uses

- a) Comments are used for the internal documentation (remarks) of a program.
- b) Comments are used to specify a heading for a program.
- c) If we want to find the result of a program when a certain set of instructions is absent then without removing those instructions, we can just put them in comments.

Ex 1 : `/* single line comment */`

Ex 2 : `/* multi line comment
multi line comment
multi line comment */`

3.11 What happens if ... ?

Usually we should not put any semicolon at the end of “for” loop expression. Lets see what happens if we put a semicolon at the end of a for loop expression.

Program 3.35

```
void main()
{
    int i;

    for(i = 1 ; i <= 10 ; i++) ;
    printf("%d\n", i);
}
```

Output

11

Explanation

a) If we put a semicolon at the end of a **for** loop expression, it means that there is nothing inside the **for** loop. So the immediate statement doesn't belong to the **for** loop.

b) The **for** loop runs as long as the condition is **TRUE**. When the control terminates the **for** loop, 'i' would be 11 and hence the output.

Lets see what happens if we put a semicolon at the end of an 'if' expression.

Example 1

Program 3.36

```
void main()
{
    int n;

    printf("Enter a number:");
    scanf("%d", &n);

    if( n == 10) ;
    printf("The entered number is 10");
}
```

Output

Enter a number : 15
The entered number is 10

Explanation

If we put a semicolon at the end of an **'if'** expression, it means that no statement is associated with it. So irrespective to the condition being **TRUE** or **FALSE**, the immediate statement (here **printf()**) will be executed.

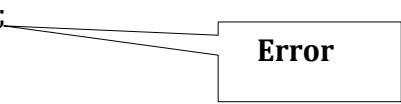
Example 2

Program 3.37

```
void main()
{
    int m;

    printf("Enter marks:");
    scanf("%d", &m);

    if( m >= 35 ) ;
        printf("Passed");
    else
        printf("Failed");
}
```



Explanation

As explained in the previous example, there is no statement associated with the **'if'** expression. So there is no connection between the **'if'** expression and the **printf("Passed");** statement.

The **Rule** is: **else** part should immediately follow the **'if'** block, otherwise the **else** part/block is called as hanging **else** which is illegal in **C**.

3.12 Generating random numbers

Generating random numbers is an important need in programming. In real-life there are many fields where random numbers are important.

Ex1 : In KBC (kaun Banega Crorepathi) game, removing randomly selected two incorrect answers

Ex2 : In Dice game : random face values will be obtained.

Ex3 : In Lottery tickets draw, a ticket is randomly picked.

/*Lets see how to generate random numbers*/

Program 3.38 (a)

(Method-1)

```
#include<stdlib.h>

void main()
{
    int i, r;

    randomize();

    for(i=1;i<=10;i++)
    {
        r = random(100);

        printf("%d\t", r);
    }
}
```

Output

82, 86, 72, 23, 45, 48, 88, 84, 21, 48

Explanation

- **random()** takes a number and returns a random number in the range 0 to one less than the given number.
- **randomize()** initializes the random number generator with a random value.
- However the above library is not portable across all compilers, In the next program we will see the portable method

Program 3.38 (b)**(Method-2)**

```
void main()
{
    int i, r;

    srand(time());

    for(i=1;i<=10;i++)
    {
        r = rand() % 100;

        printf("%d\t", r);
    }
}
```

Explanation

a) **srand()** : It seeds the random number generator used by the function rand(). It takes an integer value to be used as seed by the pseudo random number generator algorithm. Often the time() function is used as input for the seed

b) **time()**: It returns the time since January 1, 1970 in seconds, which, we know keeps on changing for execution of the program.

3.13 Difference between *for*, *while* & *do - while*

When to use what?

for

- When we know in advance, the number of times the loop should run.
- When we have all the three expression – initialization, testing and modification, directly.
- When, modification should take place at the end of a loop.
- When the modification expression is small, usually unary incrementation / decrementation.

While

- When we don't know in advance, the number of times the loop should run.
- When the modification expression has to occur in between the body of the loop.
- When the initialization and/or modification is/are big expressions

do - while

Similar to while loop, except that it runs atleast once.

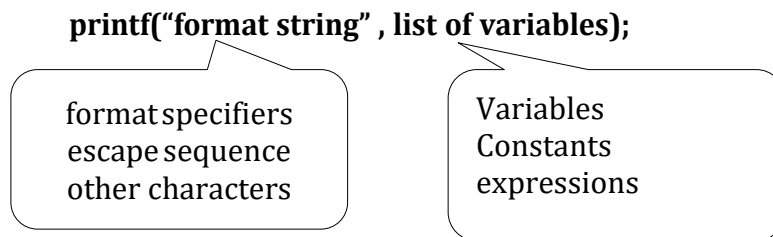
Note

Remember that a task that can be performed using one loop, can also be performed using the other two loops. However a particular loop would be more suitable for a particular requirement.

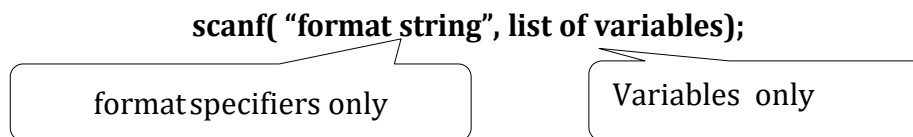
for, **while** loops are called as pre-checking loops, where as **do-while** loop is called as post-checking loop.

3.14 printf(), scanf() revisited

General form of **printf()** function is :



General form of **scanf()** function is:

**3.15 C instructions**

Till now we have seen several types of instructions. Lets classify them:

- | | |
|----------------------------------|---|
| a) Type declaration instructions | int a; |
| b) Arithmetic instructions | c = a + b; |
| c) Input/ Output instructions | printf("%d", a); |
| d) Control instruction | if(m >= 35)
printf("Passed");
else
printf("Failed"); |

3.16 pyramids

Loop control structures are very important in C programming, to understand the control flow in nested loops very well, lets generate some shapes which simulates the pyramids.

/* pyramid - example 1 */

/* A program which reads a number and generates the following pyramid */

```
1
12
123          if the given number is 5
1234
12345
```

Program 3.39

```
void main()
{
    int n, i, j;

    printf("Enter a number:");
    scanf("%d", &n);

    for(i = 1 ; i <= n ; i++)                since 'n' number of lines
    {
        for(j = 1 ; j <= i ; j++)            since each line 'i' number of columns
            printf("%d", j);

        printf("\n");                        each row should be printed in each
line.
    }
}
```

/* pyramid - example 2 */

/* A program which reads a number and generates the following pyramid */

```
1
22
333          if the given number is 5
4444
55555
```

Program 3.40

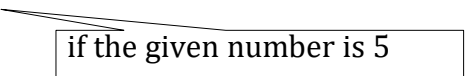
```
void main()
{
    int n, i, j;

    printf("Enter a number:");
    scanf("%d", &n);
    for(i = 1 ; i <= n ; i++)
    {
        for(j = 1 ; j <= i ; j++)
            printf("%d", i); /*we know that 'i' remains constant inside the 'j' loop*/
        printf("\n");
    }
}
```

/* pyramid - example 3 */

/* A program which reads a number and generates the following pyramid */

```
12345
1234
123
12
1
```



if the given number is 5

Program 3.41

```
void main()
{
    int n, i, j;

    printf("Enter a number:");
    scanf("%d", &n);

    for(i = n ; i >= 1 ; i--) /*since the pattern is 5, 4, 3, 2, 1 hence such a series is
generated*/
    {
        for(j = 1 ; j <= i ; j++)
            printf("%d", j);

        printf("\n");
    }
}
```

/* pyramid - example 4 */

/* A program which reads a number and generates the following pyramid */

```
54321
4321
321      if the given number is 5
21
1
```

Program 3.42

```
void main()
{
    int n, i, j;

    printf("Enter a number:");
    scanf("%d", &n);

    for(i = n ; i >= 1 ; i--)
    {
        for(j = i ; j >= 1 ; j--)
            printf("%d", j);

        printf("\n");
    }
}
```

/* pyramid - example 5 */

/* A program which reads a number and generates the following pyramid */

```
1
12
123      if the given number is 5
1234
12345
```

Program 3.43

```
void main()
{
    int n, i, j, k;
    printf("Enter a number:");
    scanf("%d", &n);
```

```
    for(i = 1 ; i <= n ; i++)
    {
        for(k = 1 ; k <= n - i ; k++)
            printf(" ");    to print ( n - i ) number of spaces

        for(j = 1 ; j <= i ; j++)
            printf("%d", j);

        printf("\n");
    }
}
/* pyramid - example 6 */
/* A program which reads a number and generates the following pyramid */
```

```
    1
   123
  12345      if the given number is 5
 1234567
123456789
```

Program 3.44

```
void main()
{
    int n, i, j, k;

    printf("Enter a number:");
    scanf("%d", &n);

    for(i = 1 ; i <= n ; i++)
    {
        for(k = 1 ; k <= n - i ; k++)
            printf(" ");

        for(j = 1 ; j <= 2 * i - 1 ; j++)
            printf("%d", j);

        printf("\n");
    }
}
```

```
/* pyramid - example 7 */
```

```
/* A program which reads a number and generates the following pyramid */
```

```
1
12
123          if the given number is 5
1234
12345
1234
123
12
1
```

Program 3.45

```
void main()
{
    int n, i, j, k;

    printf("Enter a number:");
    scanf("%d", &n);

    for(i = 1 ; i <= 2 * n - 1 ; i++)
    {
        k = (i <= n) ? i : n - i % n;

        for(j = 1 ; j <= k ; j++)
            printf("%d", j);

        printf("\n");
    }
}
```

Exercise

State True or False

- a) **break** is used to terminate a loop.
- b) A logic implemented using **for** loop cannot be implemented using **while** loop.
- c) **for** loop runs at-least once.
- d) Conditional operator has three operands.
- e) **default** is compulsory inside a **switch – case** control structure.

ARRAYS

Topic # 4

Atish Jain

4.1 What is an Array?

Arrays are variables capable of holding more than one value at a time.

For understanding the requirement of arrays let's consider the following two programs:

Program 4.1

/* reading/printing marks of five students - example 1*/

```
void main()
{
    int m, i;

    printf("Enter five students' marks:");
    for(i=1;i<=5;i++)
        scanf("%d", &m);

    printf("The marks are\n");
    for(i=1;i<=5;i++)
        printf("%d\n", m);
}
```

Output

Enter the student's marks:

39

47

63

45

56

The marks are

56

56

56

56

56

Explanation

Ordinary variables are capable of holding only one value at a time. That's why each time when we enter a new value into 'm', the existing value gets overwritten. So down the program when we want to access the marks of all the five students, we couldn't do so. So certainly this is not the solution for our problem.

Lets consider one more program:

/* reading/printing marks of five students - example 2*/

Program 4.2

```
void main()
{
    int m1, m2, m3, m4, m5;

    printf("Enter five students' marks:");
    scanf("%d%d%d%d%d", &m1, &m2, &m3, &m4, &m5);

    printf("The marks are\n");
    printf("%d, %d, %d, %d, %d", m1, m2, m3, m4, m5);
}
```

Output

Enter the student's marks:

39

47

63

45

56

The marks are

39

47

63

45

56

Explanation

No doubt, this program meets our requirement, but imagine a program which need to read 300 students' marks. In such a case we have two options to store these marks in memory:

- i) Define 300 variables to store marks obtained by 300 different students.
- ii) Define one variable capable of holding all the 300 marks.

Obviously, the second option is better.

Use of Arrays

- storing more than one value at a time under a single name.
- Reading, processing and displaying the array elements is far easier.
- Some logics can be implemented only through arrays.

Array

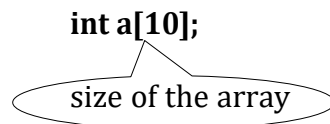
An array can be defined as a collection of similar elements stored in adjacent locations. These similar elements could be all ints or all floats, or all chars etc.

Let start with 1-D integer arrays:

4.2 1-D integer arrays

Defining a 1-D int array

Example :

`int a[10];`


The above statement tells the Compiler to reserve sufficient amount of space to store 10 integers (20 bytes) and specify a name 'a' to the entire collection.

Size : Denotes the maximum number of elements that can be stored in the array.

Organization of the elements in memory

	a[0]	a[1]				a[5]		a[7]		a[9]
elements	26	14	93	6	7	9	13	43	12	54
indices	0	1	2	3	4	5	6	7	8	9
addresses	100	102	104	106	108	110	112	114	116	118

The indices begin with 0 (zero) and ends with one less than the size of the array.

The array elements are accessed or referred to by its position in the group, called **index** or **subscript**.

/* a program which reads a set of integers into a 1-D int array and prints them */

Program 4.3

```
void main()
{
    int a[10], n, i;

    printf("How many numbers do u want to enter:");
    scanf("%d", &n);

    printf("Enter numbers:");
    for(i = 0 ; i < n ; i++)
        scanf("%d",&a[i]);

    printf("The entered numbers are\n");

    for(i = 0 ; i < n ; i++)
        printf("%d\t", a[i]);
}
```

Output

```
How many numbers do u want to enter: 5
Enter numbers:
39
47
63
45
56
The entered numbers are
39
47
63
45
56
```

Explanation

- a) Arrays are used only when the maximum size of a collection is known during the development of the program.
- b) When the program is executed, we get the prompt : How many :
Do remember that the number that we enter (the '**n**' value) should be less than or equal to the size of the array.
- c) **No bounds checking**

In 'C', there is no check to see if the subscript used for an array exceeds the size of the array. Data entered with a subscript exceeding the array size will simply be placed in memory outside the array (i.e., outside the reserved area of that array), probably on top of other data (reserved or unreserved). This will lead to unpredictable results:

- corrupting our program's data.
- corrupting our program (.exe file itself)
- corrupting other programs currently present in RAM.
- In some cases the computer may just hang. (on corruption of system files).

So checking the bounds of an array is programmer's responsibility.

Lets see some logic programs:

/* sum of array elements */

Program 4.4

```
void main()
{
    int a[10], n, i, s = 0;

    printf("How many numbers do u want to enter:");
    scanf("%d", &n);

    printf("Enter numbers:");
    for(i = 0 ; i < n ; i++)
    {
        scanf("%d",&a[i]);
        s = s + a[i];
    }

    printf("Sum = %d", s);
}
```

/* results sheet */

Program 4.5

```
void main()
{
    int rno[10], marks[10], n, i;

    printf("How many students details do u want to enter:");
    scanf("%d", &n);

    for(i = 0 ; i < n ; i++)
    {
        printf("Enter id and marks:");
        scanf("%d%d", &rno[i], &marks[i]);
    }

    printf("Results sheet\n");

    printf("-----\n\n");

    printf("First class\n");

    for(i = 0 ; i < n ; i++)
        if( marks[i] >= 60 )
            printf("%d -> %d\n", rno[i], marks[i]);

    printf("Second class\n");

    for(i = 0 ; i < n ; i++)
        if( marks[i] >= 50 && marks[i] < 60 )
            printf("%d -> %d\n", rno[i], marks[i]);

    printf("Third class\n");

    for(i = 0 ; i < n ; i++)
        if( marks[i] >= 35 && marks[i] < 50 )
            printf("%d -> %d\n", rno[i], marks[i]);
}
```

```
/* sorting a group of integers into ascending order */  
/* bubble/sink/interchange sorting method */
```

Program 4.6

```
void main()  
{  
    int a[10], n, i, j, t, flag;  
  
    printf("How many numbers do u want to enter:");  
    scanf("%d", &n);  
  
    printf("Enter the numbers:");  
    for(i = 0 ; i < n ; i++)  
        scanf("%d", &a[i]);  
  
    for(i = 0 ; i < n - 1 ;i++)  
    {  
        flag = 1;  
        for(j = 0 ; j < n - 1 - i ; j++)  
        {  
            if( a[j] > a[j+1] )  
            {  
                t = a[j];  
                a[j] = a[j+1];  
                a[j+1] = t;  
  
                flag = 0;  
            }  
        }  
  
        if(flag == 1)  
            break;  
    }  
  
    printf("In ascending order\n");  
    for(i = 0 ; i < n ; i++)  
        printf("%d\n", a[i]);  
}
```

Output

How many numbers do u want to enter: 5

Enter the numbers:

8

6

5

9

2

In ascending order

2

5

6

8

9

Explanation

Sorting any data is one of the most regular activities in computing. There are several sorting techniques. One of them is the bubble sorting technique.

Bubble sorting technique

a) The basic idea in this method is to scan the elements from left to right(or top to bottom) and sinking the elements with large keys (values) towards the right (bottom) side.

b) As an example, suppose that there are five elements in the array as following:

8	6	5	9	2
---	---	---	---	---

then, according to bubble sorting, a total of four passes (repetitions) are carried out.

c) At the end of pass one, the largest element among the five elements is set at the last position.

9

d) At the end of pass two, the largest element among the remaining four elements is set at the second last position.

8	9
---	---

e) Similarly at the end of all the four passes the array would be sorted completely.

2	5	6	8	9
---	---	---	---	---

Now the question is how to set the largest element among a set a elements at its appropriate position?

Lets consider the original array again:

8	6	5	9	2
---	---	---	---	---

During pass one

a) The elements in 0, 1 positions are compared, if the left element is greater than the right element then they are interchanged, then the sequence becomes:

6	8	5	9	2
---	---	---	---	---

b) Next, the elements in 1, 2 indices are compared, as the left element is greater than its right element they are interchanged, then the sequence becomes:

6	5	8	9	2
---	---	---	---	---

c) Next, the elements in 2, 3 indices are compared, and no interchange takes place. Now the sequence remains unchanged.

6	5	8	9	2
---	---	---	---	---

d) Next, the elements in 3,4 indices are compared and interchanged as 9 is greater than

2. Now the sequence becomes:

6	5	8	2	9
---	---	---	---	---

At the end of this **pass – one**, the element with the largest key value is set at the last position.

Similarly the sub-sequent passes are carried out.

In pass-one comparison is carried out 4 times

In pass-two comparison is carried out 3 times

In pass-three comparison is carried out 2 times

In pass-four comparison is carried out 1 times

So, for '**n**' elements (**n-1**) passes are carried out, and hence the outer **for** loop is repeated (**n-1**) times.

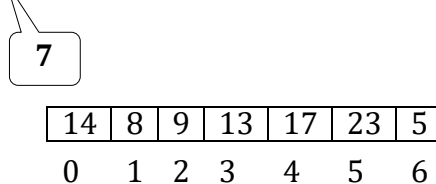
```
for(i = 0 ; i < n - 1 ; i++)
```

and each time (**n-1-i**) comparisons are carried out and hence the inner for loop is repeated (**n-1-i**) times.

```
for(j = 0 ; j < n - 1 - i ; j++)
```

1-D int array initialization

Example 1 `int a[] = { 14, 8, 9, 13, 17, 23, 5 };`



So when an array is initialized, mentioning its size becomes optional. The Compiler automatically sets its size with the number of elements initialized.

Example 2 `int a[10] = { 9, 13, 7, 6 };`

9	13	7	6	0	0	0	0	0	0
---	----	---	---	---	---	---	---	---	---

Note that, even if one location is initialized, the remaining locations of the array will be filled with zeros.

4.3 1-D character arrays (strings)

The way a group of integers can be stored in an integer array, similarly, a group of characters can be stored in a character array. Character arrays are used while dealing with names. Many a times 1-D character arrays are also called as **strings**.

Defining a 1-D character array

```
char st[20];
```

The above statement, tells the Compiler to reserve sufficient amount of space to store 20 characters (20 bytes) and specify a name 'st' to the entire collection.

Initializing a 1-D character array

method 1

```
char st[ ] = { 'v', 'i', 'k', 'r', 'a', 'm' };
```



6

Organization of the elements in memory

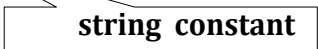
v	i	k	r	a	m
0	1	2	3	4	5
100	101	102	103	104	105

method 2

```
char st[ ] = "vikram";
```



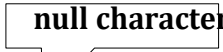
7



string constant

Organization of the elements in memory

v	i	k	r	a	m	\0
0	1	2	3	4	5	6
100	101	102	103	104	105	106



null character

Soon we will discuss the significance of this null character. Even though '\0' seems to be two characters only one byte is allocated for it. Note that whatever is followed after \ (backslash) is a special character.

/* reading/printing a group of characters - method 1*/

Program 4.7

```
#include<stdio.h>

void main()
{
    char st[30];
    int n, i;

    printf("How many characters do u want to enter:");
    scanf("%d", &n);

    printf("Enter the characters(name):");
    fflush(stdin);
    for(i = 0 ; i < n ; i++)
        scanf("%c", &st[i]);

    printf("The entered name is :");
    for(i = 0 ; i < n ; i++)
        printf("%c", st[i]);
}
```

Explanation

a) Reading/ printing a group of characters is similar to reading/printing a group of integers.

b) **fflush(stdin);**

It is safe to use the above statement while reading characters/ strings. **fflush()** happens to be one more library function. **stdin** stands for standard input, it is a **macro** defined in the file **stdio.h** that's why we have included this file. We will discuss macros in Chapter 6.

Whenever we press the enter key before reading any character, the entry key character remains in the keyboard buffer(temporary memory area where data is stored temporarily before getting stored in the variables in **RAM**) which is then retrieved as an input character thereby not allowing us to input a character. So, before inputting any character, just flush the keyboard buffer.

c) Reading/printing names can be accomplished much more easily. This is demonstrated in the next program.

/* reading/printing a group of characters - method 2*/

Program 4.8

```
void main()
{
    char st[30];

    printf("Enter the characters(name):");
    scanf("%s", st);

    printf("The entered name is : %s", st);
}
```

Output

Enter a name : Virendra Sehwag
Sehwag

Explanation

There are many issues to be discussed. Lets see one by one.

- a) for reading/printing names no need of using any loops
- b) The **%s** format specifier in the **scanf()** function keeps on reading the characters until it encounters a **whitespace character** (space/tab space/enter key) and places a special character called as null character at the end of the string (the null character indicates end of the string). That's why no need of loops for reading strings.
- c) Note that the **%s** format specifier in the **scanf()** can accept only single word (as the string gets terminated on encountering a whitespace character).
- d) The **%s** format specifier in the **printf()** function keeps on printing the characters until it encounters the null character (`'\0'`).

The next program illustrates reading of multi-word strings.

```
/* reading multi - word names */
```

Program 4.9

```
void main()
{
    char na[30];

    printf("Enter a name:");
    gets(na);

    printf("%s", na);
}
```

Output

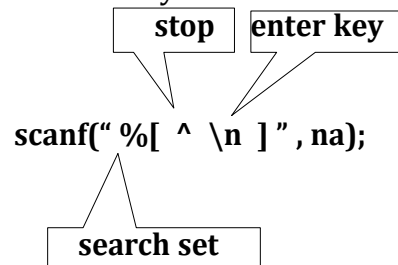
```
Enter a name : Virendra Sehwag
Virendra Sehwag
```

Explanation

a) `gets()` is an exclusive function for reading strings(it can read multi-word strings as well).

Also `gets()` function executes faster than **`scanf()`** function because relatively `gets()` function has less machine code than the machine code of **`scanf()`** function, because `gets()` function handles only strings whereas **`scanf()`** function can handle any type of data.

b) There is one more way to read multi-word strings:



The above instruction means that :

keep on searching for the enter key(`'\n'`) from the keyboard buffer area and stop reading the characters as soon as it is found.

Remember that when we enter any data from the keyboard, the data is initially stored in the keyboard buffer, when we press the enter key only then the characters are passed into **RAM**(because how can the Compiler know where our input value ends).

c) As we have **gets()**, exclusively for reading strings, similarly, we have **puts()**, exclusively for printing the strings.

eg: puts(st); or puts("Hello");

Note that using puts(), only one string can be printed at a time.
puts(), after printing the string, positions the cursor in the next line.

String handling is very often done in programming. For fulfilling the common requirements, we have a good collection of library functions. Lets see how to use them.

String handling using string library functions

/* finding string length */

Program 4.10

```
void main()
{
    char st[30];
    int l;

    puts("Enter a string:");
    gets(st);

    l = strlen(st);

    printf("Length = %d", l);
}
```

Output

```
Enter a string: B V Raju
Length = 8
```

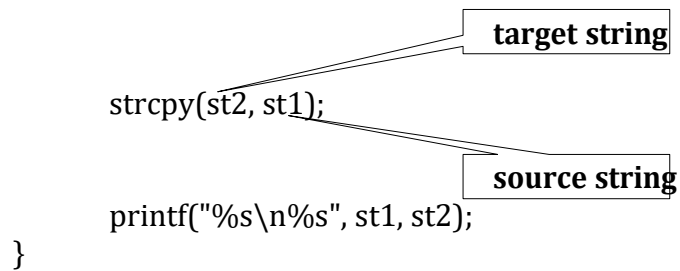
/* string copy */

Program 4.11

```
void main()
{
    char st1[30], st2[30];

    printf("Enter a string:");
    gets(st1);
}
```

```
        strcpy(st2, st1);
    printf("%s\n%s", st1, st2);
}
```



Output

```
Enter a string: India
India
India
```

/* string reverse */

Program 4.12

```
void main()
{
    char st1[30], st2[30];

    printf("Enter a string:");
    gets(st1);

    strcpy(st2, st1);
    strrev(st2);
    printf("Original : %s\\nReverse : %s", st1, st2);
}
```

Output

```
Enter a string : Marripalem
Original : Marripalem
Reverse : melapirraM
```

/* string comparision */

Program 4.13

```
void main()
{
    char st1[30], st2[30];
    int n;
```



```
    printf("Enter string 1:");

    gets(st1);
    printf("Enter string 2:");
    gets(st2);

    n = strcmp(st1, st2);

    if(n == 0)
        printf("same");
    else
        printf("not same");
}
```

Output

```
Enter string 1 : sachin
Enter string 2 : SACHin
not same
```

/* string concatenation */

Program 4.14

```
void main()
{
    char st1[30], st2[30], st3[60];
    int n;

    printf("Enter string 1:");
    gets(st1);
    printf("Enter string 2:");
    gets(st2);

    strcpy(st3, st1);
    strcat(st3, st2);

    printf("%s\n%s\n%s", st1, st2, st3);
}
```

Output

```
Enter string1 : shiva
Enter string2 : rama
```

```
shiva  
rama  
shivarama
```

Even though we have so many string library functions, they cannot fulfill our specific requirements.

So for that purpose we have to code them. First, let's imitate the above used library functions.

String handling without using string library functions

/* string length without using strlen() */

Program 4.15

```
void main()  
{  
    char st[30];  
    int i;  
  
    puts("Enter a string:");  
    gets(st);  
  
    for(i = 0 ; st[i] != '\0' ; i++);  
  
    printf("Length : %d", i);  
  
}
```

Explanation

a) As you know, semicolon at the end of a **for** loop statement indicates that there is nothing inside the **for** loop. The loop runs as long as the testing condition is TRUE.

b) **st[i] != '\0';**

We know that a string is terminated by the null character. That's why we keep on incrementing 'i' until the null character is encountered.

c) When the control comes out of the **for** loop, 'i' contains the length of the string.

/* string copy without using strcpy() */

Program 4.16

```
void main()  
{  
    char st1[30], st2[30];  
    int i;
```

```
    puts("Enter a string:");
    gets(st1);

    for(i = 0 ; st1[i] != '\0' ; i++)
        st2[i] = st1[i];

    st2[i] = '\0';

    printf("%s\n%s", st1, st2);
}
```

Explanation

- Within the loop the null character doesn't get copied into the target string. Hence this job is done explicitly outside the loop.
- '\0' can be represented as **0(zero)** which is its ASCII value or **NULL (a macro)**. To use this macro we have to include the file **stdio.h** at the top of the program as follows:

/* string reverse without using strrev() */

Program 4.17

```
void main()
{
    char st1[30], st2[30];
    int l, i, j;

    puts("Enter a string:");
    gets(st1);

    for(l = 0 ; st1[l] != 0 ; l++);

    j = l - 1;

    for(i = 0 ; i < l ; i++, j--)
        st2[i] = st1[j];

    st2[i] = 0;

    printf("Original : %s\nReverse : %s", st1, st2);
}
```

```
/* string comparison without using strcmp() */
```

Program 4.18

```
void main()
{
    char st1[30], st2[30];
    int l1, l2, i;

    puts("Enter string 1 :");
    gets(st1);

    puts("Enter string 2 :");
    gets(st2);

    for(l1 = 0; st1[l1] != 0 ; l1++);

    for(l2 = 0; st2[l2] != 0 ; l2++);

    if(l1 != l2)
    {
        printf("not same");
        exit();
    }

    for(i = 0 ; i < l1 ; i++)
        if( st1[i] != st2[i] )
            break;

    if(i == l1)
        printf("same");
    else
        printf("not same");
}
```

Explanation

a) Here the logic is if the lengths are not same, just print the message : “not same” and quit the program. **exit()** function is used to terminate the program. Remember that **break** is used to terminate a loop, whereas **exit()** is used to terminate a program.

b) If the lengths are same then run a loop(length number of times) and terminate the loop as soon as the compared characters are different. Outside the loop just check whether the loop has run length number of times or not.

/* string concatenation without using strcat() */

Program 4.19

```
void main()
{
    char st1[30], st2[30], st3[60];
    int i, j;

    puts("Enter string 1 :");
    gets(st1);

    puts("Enter string 2 :");
    gets(st2);

    for(i = 0 ; st1[i] != 0 ; i++)
        st3[i] = st1[i];

    st3[i] = '*'; /* to put a star between two strings */

    j = i + 1;

    for(i = 0 ; st2[i] != 0 ; i++, j++)
        st3[j] = st2[i];

    st3[j] = 0;

    printf("%s\n%s\n%s", st1, st2, st3);
}
```

Output

```
Enter string 1 : Megastar
Enter string 2 : Chiranjeevi
Megastar
Chiranjeevi
Megastar*Chiranjeevi
```

Lets see some logic programs:

/* finding number of words in a sentence - example 1*/

Program 4.20

```
void main()
{
    char st[128];
    int i, nwords = 0;

    puts("Enter a sentence:");
    gets(st);

    for(i = 0 ; st[i] != 0 ; i++)
        if( st[i] == ' ' )
            nwords++;

    printf("Number of words = %d", nwords + 1);
}
```

/* finding number of words in a sentence - example 2*/
/* if random number of spaces between the words */

Program 4.21

```
void main()
{
    char st[128];
    int i, nwords = 0;

    puts("Enter a sentence:");
    gets(st);

    for(i = 0 ; st[i] != 0 ; i++)
        if( st[i] == ' ' && st[i+1] != ' ' )
            nwords++;

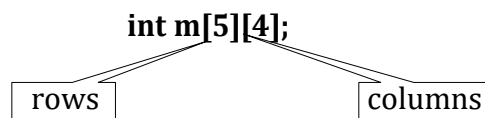
    printf("Number of words = %d", nwords + 1);
}
```

4.4 2-D integer arrays (matrices)

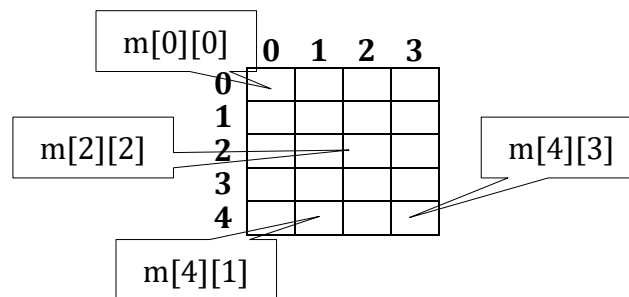
A 2-D array is nothing but a collection of 1-D arrays. 2-D arrays are used to represent any data in rows and cols format.

Lets begin our discussion with 2-D integer arrays.

Defining a 2-D int array



Here 40 bytes (20 integers) gets allocated to 'm'



To access an element of a 2-D int array we have to specify both the **row** and **column** subscripts.

Initializing a 2-D int array

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

Note that the first dimension size is optional.

One more way

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

Note

The size of second dimension(column) is compulsory because the Compiler should know the number of elements in each of the 1-D arrays. (note that its not compulsory to initialize all the elements of each of the rows).

Organization of 2-D array elements in memory.

Even though the 2-D array elements are represented in a tabular format, they are internally stored in contiguous (adjacent) memory locations.

int m[5][4];

Row major style (row by row)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
100	102	104	106	108	110	112	114	116	118	120	122	124	126	128	130	132	134	136	138

/* reading/printing a matrix */

Program 4.22

```
void main()
{
    int m[5][5], r, c, i, j;

    printf("How many rows and cols:");
    scanf("%d%d", &r, &c);

    printf("Enter elements:");
    for(i = 0 ; i < r ; i++)
        for(j = 0 ; j < c ; j++)
            scanf("%d", &m[i][j]);

    printf("The entered elements are\n");
    for(i = 0 ; i < r ; i++)
    {
        for(j = 0 ; j < c ; j++)
            printf("%3d", m[i][j]);

        printf("\n");
    }
}
```


Explanation

a) As we have to access elements in rows and cols, we have used nested loops. The outer loops runs row number of times, whereas, the inner loop runs column number of times.

b) During printing the matrix elements, to print each row in each line we have used **printf("\n")** after the inner loop.

Lets see some logic programs:

/* sum of elements of a matrix */

Program 4.23

```
void main()
{
    int m[5][5], r, c, i, j, s = 0;

    printf("How many rows and cols:");
    scanf("%d%d", &r, &c);

    printf("Enter elements:");
    for(i = 0 ; i < r ; i++)
        for(j = 0 ; j < c ; j++)
        {
            scanf("%d", &m[i][j]);
            s = s + m[i][j];
        }
    printf("Sum of elements = %d", s);
}
```

```
/* sum of two matrices */
```

Program 4.24

```
void main()
{
    int m1[5][5], m2[5][5], m3[5][5], r, c, i, j;

    printf("How many rows and cols:");
    scanf("%d%d", &r, &c);

    printf("Enter elements into matrix 1:");
    for(i = 0 ; i < r ; i++)
        for(j = 0 ; j < c ; j++)
            scanf("%d", &m1[i][j]);

    printf("Enter elements into matrix 2:");
    for(i = 0 ; i < r ; i++)
        for(j = 0 ; j < c ; j++)
            scanf("%d", &m2[i][j]);

    for(i = 0 ; i < r ; i++)
        for(j = 0 ; j < c ; j++)
            m3[i][j] = m1[i][j] + m2[i][j];

    printf("Sum of two matrices:\n");
    for(i = 0 ; i < r ; i++)
    {
        for(j = 0 ; j < c ; j++)
            printf("%3d", m3[i][j]);

        printf("\n");
    }
}
```

Explanation

Rule : To add two matrices, their order must be same.

/* product of two matrices */

Program 4.25

```
void main()
{
    int m1[5][5], m2[5][5], m3[5][5] = {0}, r1, c1, r2, c2, i, j, k;

    printf("How many rows and cols for matrix 1:");
    scanf("%d%d", &r1, &c1);

    printf("How many rows and cols for matrix 2:");
    scanf("%d%d", &r2, &c2);

    if(c1 != r2)
    {
        printf("Multiplication rule violated - cannot be multiplied");
        exit();
    }

    printf("Enter elements into matrix 1:");
    for(i = 0 ; i < r1 ; i++)
        for(j = 0 ; j < c1 ; j++)
            scanf("%d", &m1[i][j]);

    printf("Enter elements into matrix 2:");
    for(i = 0 ; i < r2 ; i++)
        for(j = 0 ; j < c2 ; j++)
            scanf("%d", &m2[i][j]);

    /* coding the product */

    for(i = 0 ; i < r1 ; i++)
    {
        for(j = 0 ; j < c2 ; j++)
        {
            for(k = 0 ; k < c1 ; k++)
                m3[i][j] = m3[i][j] + m1[i][k] * m2[k][j];
        }
    }
}
```

```
printf("Product of two matrices:\n");
for(i = 0 ; i < r1 ; i++)
{
    for(j = 0 ; j < c2 ; j++)
        printf("%5d", m3[i][j]);

    printf("\n");
}
}
```

Explanation:

Rule : The number of columns of matrix 1 must be equal to the number of rows of matrix 2.

How to multiply?

Each row of matrix 1 must be multiplied with each column of matrix 2.

Now lets understand the product code:

Suppose that

the order of matrix 1 : r1 = 3, c1 = 4

the order of matrix 2 : r2 = 4, c2 = 5

then, the order of matrix 3 : rows = 3, columns = 5 (r1, c2)

matrix1

	0	1	2	3
0				
1				
2				

3x4

matrix2

	0	1	2	3	4
0					
1					
2					
3					

3x5

matrix3

	0	1	2	3	4
0					
1					
2					

4x5

/* coding the product */

```
for(i = 0 ; i < r1 ; i++)
{
    for(j = 0 ; j < c2 ; j++)
    {
        for(k = 0 ; k < c1 ; k++)
            m3[i][j] = m3[i][j] + m1[i][k] * m2[k][j];
    }
}
```

a) To visit every cell of matrix 3 we need two loops ('i' loop and 'j' loop).

b) To fill each cell of **m3**, we have to carry out the multiplication **four times** (in our example). and then add them (running sum). Hence we need another loop which runs four times (i.e., c1 or r2 times).

Now lets understand,

$$\mathbf{m1[i][k] * m2[k][j]}$$

We know that,

 'i' loop runs 3 times

 'j' loop runs 5 times

 'k' loop runs 4 times

So, the statement,

$$\mathbf{m3[i][j] = m3[i][j] + m1[i][k] * m2[k][j];}$$

runs $3 * 5 * 4 = 60$ times.

Lets see how multiplication is carried out manually

m3	m1	*	m2
0,0	0,0		0,0
	0,1		1,0
	0,2		2,0
	0,3		3,0
0,1	0,0		0,1
	0,1		1,1
	0,2		2,1
	0,3		3,1
0,2	0,0		0,2
	0,1		1,2
	0,2		2,2
	0,3		3,2
0,3	0,0		0,3
	0,1		1,3
	0,2		2,3
	0,3		3,3

0,4	0,0	0,4
	0,1	1,4
	0,2	2,4
	0,3	3,4

1,0	1,0	0,0
	1,1	1,0
	1,2	2,0
	1,3	3,0

From the above numbers sequence, it is clear that the 2nd index of m1 and 1st index of m2 changes every time. We know that 'k' changes every time. Hence the code : **m[][k] * m2[k][]** From the above numbers sequence, it is clear that the 2nd index of m2 changes every four times. We know that 'j' changes every four times.

Hence the code : **m[][k] * m2[k][j]**

From the above numbers sequence, it is clear that the 1nd index of m1 changes every twenty times. We know that 'i' changes every twenty times. Hence the code : **m[i][k] * m2[k][j]**

/* Transpose of a Matrix */

Program 4.26

```
void main()
{
    int m[5][5], t[5][5], i, j, r, c;

    printf("How many rows and cols:");
    scanf("%d%d", &r, &c);

    printf("Enter the matrix:");
    for(i=0;i<r;i++)
        for(j=0;j<c;j++)
            scanf("%d", &m[i][j]);

    /* Now finding transpose */

    for(i=0;i<r;i++)
        for(j=0;j<c;j++)
            t[j][i] = m[i][j];
```

```
printf("Original Matrix:\n");

for(i=0;i<r;i++)
{
    for(j=0;j<c;j++)
        printf("%5d", m[i][j]);
    printf("\n");
}

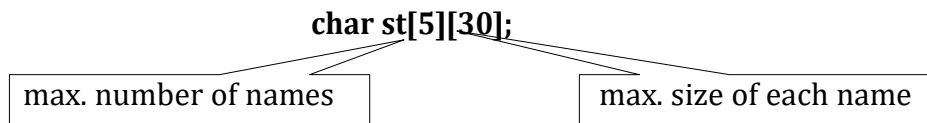
printf("\nIts Transpose:\n");
for(i=0;i<c;i++)
{
    for(j=0;j<r;j++)
        printf("%5d", t[i][j]);

    printf("\n");
}
}
```

4.5 2-D character arrays

2-D character arrays are nothing but an **array (collection) of strings**.

Defining a 2-D char array



Initializing a 2-D char array

```
char st[][15] = {"Lakshmi", "Ramya", "Jayanthi"};
```

Organization in memory

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	L	a	k	s	h	m	i	\0							
1	R	a	m	y	a	\0									
2	J	a	y	a	n	t	h	i	\0						

/* reading/printing a set of strings */

Program 4.27

```
#include<stdio.h>

void main()
{
    char st[10][30];
    int n, i;

    printf("How many strings do u want to enter:");
    scanf("%d", &n);

    printf("Enter strings:");

    for(i = 0 ; i < n ; i++)
    {
        fflush(stdin);
        gets(st[i]);
    }

    printf("The entered strings are\n");

    for(i = 0 ; i < n ; i++)
        printf("%s\n", st[i]);
}
```

Output

```
How many strings do u want to enter: 5
Enter strings: Ameerpet
ECIL
Gandipet
Secunderabad
Nalgonda

The entered strings are:
Ameerpet
ECIL
Gandipet
Secunderabad
Nalgonda
```


Lets see some logic programs
/* finding the longest string */

Program 4.28

```
#include<stdio.h>
void main()
{
    char st[10][30], t[30];
    int n, i, l, max = 0;

    printf("How many strings do u want to enter:");
    scanf("%d", &n);

    printf("Enter strings:");
    for(i = 0 ; i < n ; i++)
    {
        fflush(stdin);
        gets(st[i]);

        l = strlen(st[i]);

        if(l > max)
        {
            max = l;
            strcpy(t, st[i]);
        }
    }
    printf("Longest string is : %s", t);
}
/* finding the string which occurs first in alphabetical order without modifying the Array*/
```

Program 4.29

```
#include<stdio.h>
void main()
{
    char st[10][30], t[30];
    int n, i;

    printf("How many strings do u want to enter:");
    scanf("%d", &n);
```

```
    printf("Enter strings:");

    for(i = 0 ; i < n ; i++)
    {
        fflush(stdin);
        gets(st[i]);
    }

    for(i = 0 ; i < n ; i++)
    if(i == 0 || strcmp(t, st[i]) > 0 )
        strcpy(t, st[i]);

    printf("The string which occurs first in alphabetical order is : %s", t);
}

/* string sorting */
```

Program 4.30

```
#include<stdio.h>

void main()
{
    char st[10][30], t[30];
    int n, i, j, flag;

    printf("How many strings do u want to enter:");
    scanf("%d", &n);

    printf("Enter strings:");
    for(i = 0 ; i < n ; i++)
    {
        fflush(stdin);
        gets(st[i]);
    }

    for(i = 0 ; i < n - 1 ; i++)
    {
        flag = 1;

        for(j = 0 ; j < n - 1 - i ; j++)
        {
```

```
        if( strcmp(st[j], st[j+1]) > 0 )
        {
            strcpy(t, st[j]);
            strcpy(st[j], st[j+1]);
            strcpy(st[j+1], t);

            flag = 0;
        }
    }

    if(flag == 1)
        break;
}

printf("In alphabetical order\n");
for(i = 0 ; i < n ; i++)
    printf("%s\n", st[i]);
}
```

Output

How many strings do u want to enter: 7

Enter strings : shiva

sreenu

mouli

sagar

mridula

sumitra

sujana

In alphabetical order

mouli

mridula

sagar

shiva

sreenu

sujana

sumitra

Exercise

State true or false

- e) Array elements are stored in adjacent locations.
- f) Array elements must be of similar datatype.
- g) 2-D array elements are stored in adjacent locations.
- h) ASCII value of NULL character is zero.
- i) `gets()` can read multi- word strings.

Functions (Basic)

Topic # 5

Atish Jain

5.1 What are functions?

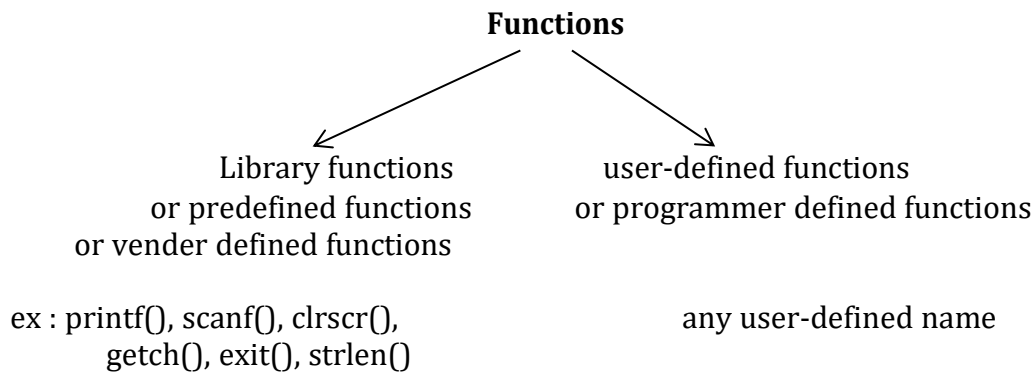
A 'C' language program is nothing but a collection of functions. These are the building blocks of a 'C' program.

Now its time to discuss it in detail. Let's discuss part by part.

Generally, function means a task.
In programming terms:

We know that array is a collection of data items, similarly, **function is a collection of logically related instructions which performs a particular task.**

Types of functions



Use of functions

Suppose that in a program, a task is getting repeated several times, so instead of repeating the code, it would be nice if we could write it once and use it whenever and wherever required. So, in short we can say that functions are used to avoid code repetition.

Benefits of avoiding code repetition

- time is saved
- thereby money is saved
- frustration is saved(because code the same thing again and again leads to frustration)
- Testing can be avoided (because we use the proved code (already tested))

5.2 How to create a function?

Syntax

```
<return value type> <function name> (parameters list)
{
    body of the function
    [return <value> ] ;           ← optional
}
```

5.3 Calling a function

/* calling a function */

Program 5.1

```
void main()
{
    int a, b, c;

    printf("Enter two numbers:") ← calling a function
    scanf("%d%d",&a, &b) ← calling a function

    c = a + b;

    printf("Sum = %d", c) ← calling a function

    getch(); ← calling a function
}
```

Explanation

a) Whenever we call a function, the control temporarily transfers to that function and after completion of the execution of that function, the control returns back to the function from where the call has been made.

b) Till now we have been calling the library functions.(which are already available in machine code).

Let's see how to create our own functions.

Let's first understand the requirement of writing our own functions:

Even though we have a huge collection of library functions, they may not meet our specific requirements. So under these situations we create our own functions and use them whenever or wherever required.

```
/* creating a user-defined function */
```

Program 5.2

```
void fun()
{
    printf("Hello");
}

void main()
{
    fun();
}
```

function definition

function calling

Output

Hello

Explanation

- a) **Function definition** : It contains the function's return type, name, parameters list and body.
- b) **Function calling** : Transferring control from one function to another function.
- c) **Called function** : A function which gets called. In our example **fun()** is the called function.
- d) **Calling function** : A function which calls an other function. In our example **main()** is the calling function.
- e) Note that a function cannot be defined in any other function.

5.4 Returning a value from a function

```
/* returning a value from a function - example 1 */
```

Program 5.3

```
int sum(int a, int b)
{
    int c;
    c = a + b;

    return c;
}
```

formal arguments

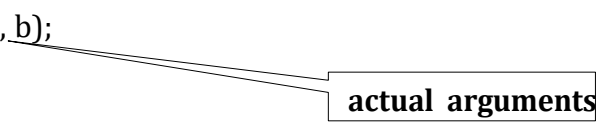
returning a value from a function


```
void main()
{
    int a, b, c;

    printf("Enter two numbers:");
    scanf("%d%d", &a, &b);

    c = sum(a, b);

    printf("Sum = %d", c);
}
```

**Explanation**

- a) Actual arguments : The variables used in the function calling statement.
- b) Formal arguments : The variables used in the receiving section of the called function.
- c) The name of actual arguments and formal arguments may or may not be same. Both of them will have different memory locations.
- d) The keyword **return** is used to return a value from a function.

/* returning a value from a function - example 2 */

Program 5.4

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

int diff(int a, int b)
{
    return (a - b);
}

int prod(int a, int b)
{
    return (a * b);
}

int quot(int a, int b)
{
    return (a / b);
}
```

```
void main()
{
    int a, b, result;
    printf("Enter two numbers:");
    scanf("%d%d", &a, &b);
    result = sum(a, b);
    printf("Sum = %d\n", result);
    result = diff(a, b);
    printf("Difference = %d\n", result);
    result = prod(a, b);
    printf("Product = %d\n", result);
    result = quot(a, b);
    printf("Quotient = %d", result);
}
```

5.5 Where to define a function? - function prototyping

Have a look at the first program of this chapter. Guess what happens if we define the **fun()** function below the **main()** function? **Error occurs**. Why? To understand the reason, you should know some points:

- a) A program's **execution** begins from the opening brace of the **main()** function and ends at the closing brace of the **main()** function.
- b) However a program's **compilation** begins from the first line of the program and ends at the last line of the program.
- c) By default a functions return type is an integer. It means that while defining a function, if we don't mention its return type, the compiler assumes it to be an integer type.
- d) Now let's discuss the error : In this case **main()** gets compiled first, when the statement **fun();** gets compiled, the compiler assumes that its return type is an integer(as by that time the compiler doesn't know about its definition). But down the program, we have mentioned its return type as **void**. Hence the compiler reports an error message.
- e) An ideal place to define a function is above its caller (i.e., its calling function).

Conclusion

If a function's return type is an integer then it can be defined any where (above or below its caller).

However if we want to define a non-integer return type function, below its calling function, then we have to provide the called **function's prototype** (here **fun()** 's) in its calling function (here **main()**) at the place where we define the variables.

/* illustrates function prototype */

Program 5.5

```
void main()
{
    void fun();           function prototype/declaration
    fun();
}

void fun()
{
    printf("Hello");
}
```

Note

The ideal place to specify a function's prototype is outside and above all the functions, so that it can be called by any other function of the program.

5.6 A menu driven program

/* to illustrate that a function can be called any number of times */

Program 5.6

```
int fact(int n)
{
    int f = 1;

    if(n > 7)
    {
        printf("Sorry! can't find a factorial for a number > 7");
        return -1;
    }

    while( n )
        f = f * n--;

    return f;
}
```

```
int reverse(int n)
{
    int r, rev = 0;

    while( n )
    {
        r = n % 10;
        rev = rev * 10 + r;
        n = n / 10;
    }

    return rev;
}

void prime(int n)
{
    int i, flag = 1;

    for(i=2; i<=n/2;i++)
        if(n%i == 0)
        {
            flag = 0;
            break;
        }

    if(flag == 1)
        printf("prime");
    else
        printf("not prime");
}

void armstrong(int n)
{
    int r, s = 0, t;

    t = n;
    while( n )
    {
        r = n % 10;

        s = s + r * r * r;
        n = n / 10;
    }
}
```

```
        if(t == s)
            printf("armstrong");
        else
            printf("no");
    }
    void main()
    {
        char ch;
        int n1, n2;

        printf("Enter a number:");
        scanf("%d", &n1);

        while( 1 )
        {
            printf("1.Factorial\n");
            printf("2.Reverse\n");
            printf("3.Prime\n");
            printf("4.Armstrong\n");
            printf("5.Exit\n\n");

            printf("Enter u'r choice:");
            ch = getche();
            printf("\n");
            switch( ch )
            {
                case '1' : n2 = fact(n1);
                           if(n2 != -1)
                               printf("Factorial : %d", n2);
                           break;
                case '2' : n2 = reverse(n1);
                           printf("Reverse = %d", n2);
                           break;
                case '3' : prime(n1);
                           break;
                case '4' : armstrong(n1);
                           break;
                case '5' : exit();
                default : printf("Invalid choice");
            }
        }
    }
```

5.7 Creating our own header files

In the previous program, you have created four different functions and used it that program.

Suppose that, in a different program, you again require all these (or some) functions, what would you do? copy and paste in the new program? No. Instead it would be nice to separate those user- defined functions and save in a separate file and use them in all the programs which requires them (all or some). Let's see how this can be achieved.

/* header- file which consists of fact(), reverse(), armstrong() and prime() functions definitions */

Program 5.7

step1 : Create all the user-defined functions

Save this file as **myfuns.h** (we cannot link/execute this file as **main()** is absent)

```
int fact(int n)
{
    int f = 1;
    if(n > 7)
    {
        printf("Sorry! can't find a factorial for a number > 7");
        return -1;
    }

    while( n )
        f = f * n--;

    return f;
}

int reverse(int n)
{
    int r, rev = 0;
    while( n )
    {
        r = n % 10;
        rev = rev * 10 + r;
        n = n / 10;
    }
    return rev;
}
```

```
void prime(int n)
{
    int i, flag = 1;

    for(i=2; i<=n/2;i++)
        if(n%i == 0)
        {
            flag = 0;
            break;
        }

    if(flag == 1)
        printf("prime");
    else
        printf("not prime");
}
```

```
void armstrong(int n)
{
    int r, s = 0, t;

    t = n;

    while( n )
    {
        r = n % 10;

        s = s + r * r * r;

        n = n / 10;
    }

    if(t == s)
        printf("armstrong");
    else
        printf("no");
}
```

step2

```
/* creating a file which contains the main() function */
#include"myfuns.h"
void main()
{
    char ch;
    int n1, n2;

    printf("Enter a number:");
    scanf("%d", &n1);

    while( 1 )
    {
        printf("1.Factorial\n");
        printf("2.Reverse\n");
        printf("3.Prime\n");
        printf("4.Armstrong\n");
        printf("5.Exit\n\n");

        printf("Enter u'r choice:");
        ch = getche();
        printf("\n");
        switch( ch )
        {
            case '1' : n2 = fact(n1);
                        if(n2 != -1)
                            printf("Factorial : %d", n2);
                        break;
            case '2' : n2 = reverse(n1);
                        printf("Reverse = %d", n2);
                        break;
            case '3' : prime(n1);
                        break;
            case '4' : armstrong(n1);
                        break;
            case '5' : exit();
            default : printf("Invalid choice");
        }
        getch();
    }
}
```


Explanation

In the statement:

#include"myfuncs.h"

Note that we have used double quotations instead of the conventional angular brackets. Why is this discrimination for our header file?

Let's discuss both the formats:

a) **#include<myfuncs.h>**

searching for the file takes place only in the **include path** : c:\turboc2\include (it depends where you have installed your header file)

b) **#include"myfuncs.h"**

searching for the file takes place first in the **current working directory**, if not found searching takes place in the **include path**.

Conclusion

If we want to use angular brackets format for a header file, it should be present in the include path.

Exercise

State true or false

- a) **printf()** is a library function.
- b) Default return type of a function is void
- c) A function can be called only a maximum of three times.
- d) To return a value from a function, the keyword **return** is used.
- e) Non-integer return type functions must be defined only above its calling function(s).

Macros and Storage Classes

Topic # 6

Atish Jain

6.1 The Preprocessor

It is exactly what its name implies. Preprocessor is a program which processes our source code before passing it to the Compiler.

A statement which begins with a # symbol is called as a **preprocessor directive**. There are several preprocessor directives. In this material we would discuss only two of them.

#include directive
#define directive

Lets discuss one by one.

#include directive

You are already aware of this directive. We have used this whenever we required to include any file.

Eg : #include<stdio.h>
 #include<conio.h>
 #include<myfuncs.h>

Whenever the preprocessor encounters this directive, it inserts the specified file in the current file.

#define directive

This will be discussed in the next topic.

Note that the preprocessor directives are not 'C' language instructions. They are replaced by an appropriate 'C' code.

6.2 Macros

A macro is a constant defined by using the preprocessor directive : **#define**.

Macros are of two types:

- simple macros
- macros with arguments

Defining a macro

Syntax

#define < macro template > < macro expansion >

There are several uses of macros:

- Increases program readability.
- Performs automatic replacement.
- Reduces typing burden.
- Increases program execution speed.

Very soon you will understand these uses.

/* macros - example 1 */

Program 6.1

```
#define NULL 0

void main()
{
    char st[20];
    int i;
    puts("Enter a string:");
    gets(st);
    for(i=0; st[i] != NULL ; i++);
    printf("Length = %d", i);
}
```

Explanation

a) Here we have defined a word **NULL** as 0 (zero). So in our program, where ever there is NULL, it is replaced by 0 (zero). Isn't NULL more readable than 0 (zero)?.

b) What happens before Compilation?

When we compile a program, before the source code is passed to the Compiler, it is examined by a preprocessor program for any macro definition. When it sees the #define directive, it goes through the entire program in search of the macro template, wherever it finds one, it replaces the macro template with its corresponding expansion. Only after completion of this procedure, the expanded program is handed over to the Compiler.

```
/* macros - example 2 */
```

Program 6.2

```
#define TRUE 1
void main()
{
    while( TRUE )
    {
        printf("This loop runs infinite times\n");
    }
}
```

```
/* macros - example 3 */
```

Program 6.3

```
#define PI 3.142857

void main()
{
    float r, a;

    printf("Enter radius of a circle:");
    scanf("%f", &r);

    a = PI * r * r;

    printf("Area of circle : %f", a);
}
```

Note

It is just customary to use capital letters for a macro template. This makes it easy for the programmer to identify all the macros while reading through the program. Usually variable names and function names are defined in lowercase, whereas, macros are defined in uppercase.

A macro expansion can extend more than one line, except for the last line, there should be a backslash (\) at the end of every line.

6.3 Macros with arguments

The macros that we have seen so far are called as simple macros. Macros can have arguments, just as functions can.

```
/* macros with arguments - example 1 */
```

Program 6.4

```
#define SUM(a, b) a+b
void main()
{
    int a, b;
    printf("Enter two numbers:");
    scanf("%d%d", &a, &b);
    printf("Sum = %d", SUM(a,b) );
}
```

Explanation

In the above example, a macro was used to find the sum of two integers. As we know, even a function can be written to do this job. Then why have we used a macro?

Here, the preprocessor replaces the macro template with its expansion. i.e., **SUM(a,b)** in the **printf()** is replaced by **a+b**. So there is no such thing as macro call during execution. Usually macros makes the program execute faster but increases the program size (if macro expansion is a big one), whereas functions make the program slower but smaller and compact.

```
/* macros with arguments - example 2 */
```

Program 6.5

```
#define PI 3.142857
#define AREA(x) PI * x * x

void main()
{
    float r, a;
    printf("Enter radius of a circle:");
    scanf("%f", &r);
    a = AREA(r);
    printf("Area of the circle = %f", a);
}
```

Importance of enclosing the variables in the macro expansion within parenthesis**/* macros with arguments - example 3 */****Program 6.6**

```
#include<math.h>
#define MAC(x, y) sqrt(x*x + y*y)
float fun(int x, int y)
{
    return sqrt(x*x + y*y);
}

void main()
{
    int a = 3, b = 4;
    float p, q, r, s;
    p = fun(a, b);
    q = fun(a+1, b+1);
    r = MAC(a, b);
    s = MAC(a+1, b+1);
    printf("Using functions : %f, %f\n", p, q);
    printf("Using macros : %f, %f", r, s);
}
```

Output

```
Using functions : 5.000000, 6.403124
Using macros : 5.000000, 4.000000
```

Explanation

Surprised? As you have observed, the values of 'q' and 's' are different but we have expected them to be same.

Macros employs blind replacement strategy. i.e.,

s = MAC(a+1, b+1);

will be replaced by

s = sqrt(a+1*a+1 + b+1*b+1);

which leads to, s = sqrt(a + a + 1 + b + b + 1)

Hence it is always safe to enclose the variables in the macro expansion within parenthesis.

i.e., **#define MAC(x, y) sqrt((x)*(x) + (y)*(y))**

Had we used this style, 'q' and 's' would have been same.

6.4 Storage classes

Till now, while defining a variable we mentioned only its data type. Even though we haven't mentioned anything more than that, something is assumed by default. To fully define a variable we need to mention not only its data type, but also its **storage class**. If we don't specify the storage class of a variable in its definition, the Compiler will assume a default storage class depending on the context where the variable is defined.

In 'C' there are four types of storage classes:

- auto (automatic)
- static
- register (CPU register)
- extern (external)

The storage class tells us four things:

Default initial value: What will be the default initial value of the variable as soon as it is defined.

Location: Where the variable would be stored.

Scope: Where the variable can be accessed.

Life: How long the memory remains reserved for the variable.

The following table shows the features of various storage classes:

	auto	static	register	extern
Default initial value	garbage	0 (zero)	garbage	0 (zero)
Location	RAM	RAM	CPU registers	RAM
Scope	Local to the block where the variable is defined	Local to the block where the variable is defined	Local to the block where the variable is defined	Entire program
Life	As long as the control is within the block where the variable is defined	As long as the program is under execution	As long as the control is within the block where the variable is defined	As long as the program is under execution

`/* nameless blocks */`

Program 6.7

```
void main()
{
    int a = 10;

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

    {
        printf("St Anns School - Secunderabad");
    }
}
```

Output

```
10
St Anns School - Secunderabad
```

Explanation

In 'C', nameless blocks are possible within a function block. The use is memory can be conserved.

How? Very soon you will understand.

`/* where to define a variable */`

Program 6.8

```
void main()
{
    int a = 10;
    printf("%d\n", a);

    {
        int b = 20;
        printf("%d", b);
    }
}
```

Explanation

In 'C' a variable must be defined before the first executable statement in a block.

/* where can we access a variable */

Program 6.9

```
void main()
{
    int a = 10;
    printf("%d\n", a);

    {
        int b = 20;
        printf("%d, %d\n", a, b);
    }

    printf("%d, %d", a, b); /* error: 'b' not accessible */
}
```

Explanation

A variable defined in a block can be accessed in that block and all its inner blocks (i.e., a variable defined in a block cannot be accessed in its outer block). It is called as the **scope of a variable**.

auto storage class specifier

/* auto storage class */

Program 6.10

```
void fun()
{
    int a = 10;
    printf("%d\n", a);
    a++;
}

void main()
{
    fun();
    fun();
    fun();
}
```

Output

10
10
10

Explanation

- a) The default storage class of a variable inside any block is ***auto***.
- b) We know that ***auto*** variable's life is as long as the control remains within the block where the variable is defined.

Whenever the statement,

int a = 10;

gets executed, memory gets allocated to 'a' and the location is initialized with 10.

Whenever the control comes out of the **fun()** functions' block, the variable 'a' dies. i.e., memory allocated to 'a' gets unreserved, and hence the output.

static storage class specifier

/* static storage class */

Program 6.11

```
void fun()
{
    static int a = 10;
    printf("%d\n", a);
    a++;
}

void main()
{
    fun();
    fun();
    fun();
}
```

Output

10
11
12

Explanation

- a) We know that **static** variable's life is as long as the program is under execution.
- b) **static** and global (very soon you will understand its meaning) variables are also called as **load time variables**. i.e., for these variables memory gets allocated as soon as the program gets loaded into memory(before the execution of **main()** function) and memory gets de-allocated just before the termination of the program from the memory.
- c) Remember that, the statement,

static int a = 10;

gets executed only once. So, no matter how many times the control enters the **fun()** functions' block, only once this statement gets executed and rest of the times it gets ignored. Also remember that the latest value of the static variable persists (remains undestroyed) between different function calls, and hence the output.

register storage class specifier

The **register** storage class specifier indicates to the Compiler that the value of the variable should reside in a machine register (CPUs memory area). The Compiler is not required to honor this request. Because of the limited size and number of registers available on most systems, few variables can actually be put in registers. If the Compiler does not allocate a machine register for a **register** variable, the variable is treated as having the storage class specifier **auto**. A **register** storage class specifier indicates that the variable, such as a loop control variable, is heavily used and that the programmer hopes to enhance performance by minimizing access time (register accessing is faster than RAM accessing).

To understand this concept better, just consider the following example:

Cooking becomes faster if all the food items are put near to the stove, rather than at a distance place in the kitchen.

/* register storage class */

Program 6.12

```
void main()
{
    register int i;

    for(i=1;i<=100;i++)
        printf("I Love India\n");
}
```

Explanation

We know that here, 'i' gets accessed around 300 times. So, if it is made to reside in the CPU registers, program execution would be faster.

Remember that, we cannot use register storage class for all types of variables.

Eg:

```
register long a;
register float b;
register double c;  } all are illegal
```

This is because the CPU registers in a microcomputer are usually 16-bit registers, therefore cannot hold a float value or a double value. The Compiler would treat the variables to be auto storage class.

A limitation of register variables is that you cannot apply **address of (&)** operator. This is because the CPU registers do not have memory addresses.

/* global variables */

Program 6.13

```
int i = 78;

void fun1()
{
    printf("%d\n", i);
}

void fun2()
{
    int i = 45;
    printf("%d\n", i);
}

void main()
{
    printf("%d\n", i);

    fun1();
    fun2();
}
```

Explanation

The variables, which are required in more than one function, are defined globally i.e., outside and above all the functions.

Use

- No need of passing them as arguments to a function.
- No need of call by address to change the values of a variable (there by avoiding the usage of pointers).

Note that global is not a keyword. It is just a concept.

extern storage class specifier

Usually a big program is divided into a number of files, so that maintenance of the program becomes easy. A variable defined in a file can be accessed by other files as well, such a variable is called as ***extern*** (external) variable.

Step 1 : Create a file **f1.h** and just define a variable in this file.

```
/* external variable */
```

```
int i = 26;                variable definition (memory is allocated)
```

step 2 : Create another file

```
/* extern storage class */
```

Program 6.14

```
#include "f1.h"
```

```
void main()
```

```
{
```

```
    extern int i;           /*variable declaration (memory is not allocated)*/
```

```
    printf("%d", i);
```

```
}
```

Explanation

The statement,

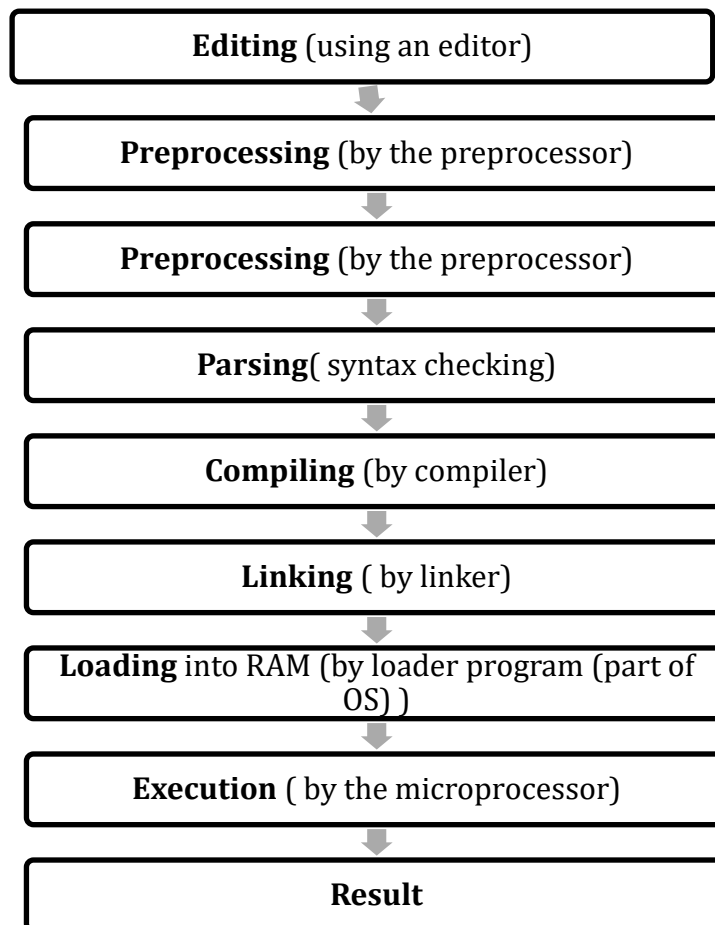
```
    extern int i;
```

indicates that the variable 'i' is defined externally in a separate file(sometimes in the same file but outside and below that function where this statement is present)

When to use what?

register	:	For frequently used variables
static	:	If variable is to live across function calls
extern	:	If variable is required by more than one file.
global	:	If variable is required in more than one function.
auto	:	All other cases

6.5 A typical C Environment (Steps before execution)



Exercise

State True or False

- a) A variable can be defined any where in a block.
- b) **static** variable is also called as load time variable.
- c) When a variable is required in more than one function, it is made global.
- d) Default initial value of a **static** variable is one.
- e) Macros must be defined in uppercase only.

Pointers & Advanced Functions

Topic # 7

Atish Jain

7.1 What is a pointer?

A Pointer is a variable that contains the address of a variable. Pointers are much used in C, partly because they are sometimes the only way to perform a particular operation, and partly because they usually lead to more compact and efficient code than can be obtained in other ways. They provide much power and utility for the programmer to access and manipulate data in ways not seen in some other languages.

After going through this chapter you will surely appreciate the use of pointers.

There are many issues to be discussed. Let's begin our journey with how to print an address.

7.2 How to print an address?

/ to print address of a variable */*

Program 7.1

```
void main()
{
    int a = 10;

    printf("%d\n", a);
    printf("%u", &a);
}
```

Output

```
10
65494
```

Explanation

- a) We know that every variable has an address. If we want, we can know what address is assigned to a variable (by the compiler).
- b) To print the address of an integer variable just use **&a** in the **printf()** function
- c) **%u** In a **16-bit compiler** like **Turbo C**, a variable can be assigned an address in the range of **1 to 65535**. The maximum positive value that can be printed using **%d** is **32767**. Hence we use **%u** for printing the addresses of variables. Note that address is always a positive value.

7.3 Defining an integer pointer

Let's begin with an integer pointer.

```
int a = 420;
int* p;
```

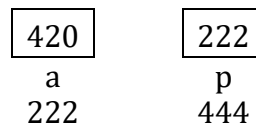
p is an integer pointer

Pointers are defined by using the * in front of the variable. Now '**p**' gets the capability to hold an integer variable's address.

```
p = &a;
```

The above statement stores the address of '**a**' in '**p**';

This is illustrated in **figure 7.1**



7.4 Address of and value at operators

/ address of and value at operators - example 1*/*

Program 7.2

```
void main()
{
    int a = 35;
    int* p;

    p = &a;

    printf("%u\n", a);
    printf("%u\n", &a);
    printf("%u\n", p);
    printf("%u\n", &p);
    printf("%u", *p);
}
```

Output

```
35
222
222
444
35
```

Explanation

a) & **address of operator**
* **value at /de-referencing / indirection operator**

b)



c) ***p** should be read as **value at 'p'**

d) How to evaluate ***p**?

First find the address in '**p**' and then find value at that address.

In our example '**p**' is 222. value at 222 is 35.

/* address of and value at operators - example 2*/

Program 7.3

```
void main()
{
    int a = 35;
    int* p;
    int** q;

    p = &a;
    q = &p;

    printf("%u\n", p);
    printf("%u\n", q);

    printf("%u\n", *q);
    printf("%u\n", *p);
    /* printf("%u\n", *a); illegal */

    printf("%u\n", **q);
    /* printf("%u\n", **p); illegal */
    printf("%u", *&p);
}
```

Output

222
444
222
35
35
222

Explanation

a)

35	222	444
a	p	q
222	444	666

b) `printf("%u\n", *a);` is illegal because only a pointer can be de-referenced

c) `printf("%u\n", **p);` is illegal because, as it is a single pointer it can be de-referenced only once i.e., only one `'*'` can be applied to `'p'`.

7.5 Size of a pointer

`/* size of a pointer */`

Program 7.4

```
void main()
{
    int a = 10;
    char ch = 'x';
    float f = 9.7;

    int* p = &a;
    char* q = &ch;
    float* r = &f;

    printf("%d, %d, %d\n", sizeof(a), sizeof(ch), sizeof(f) );
    printf("%d, %d, %d", sizeof(p), sizeof(q), sizeof(r) );
}
```

Output

2, 1, 4
2, 2, 2

Explanation

What ever may be the type of a pointer, only **two bytes** (if 16 – bit compiler) are allocated as it is going to store an address of a variable When you execute this program in C-Free Compiler you get pointer size as 4 bytes because it is 32-bit compiler.

7.6 Why different types for pointers?

/* why different types for pointers */

Program 7.5

```
void main()
{
    int a = 4873;
    char ch = 'p';
    float f = 2173.964;
    int* p;

    p = &a;
    printf("%u\n", *p); /* dereferencing */

    p = &ch;
    printf("%u\n", *p); /* dereferencing */

    p = &f;
    printf("%u", *p);    /* dereferencing */
}
```

Output

```
4873
garbage
garbage
```

Explanation

While de-referencing, the compiler can know from how many bytes a value should be extracted by looking at the pointer type.

If it is an **integer pointer**, a value from **two bytes** is extracted.

If it is a **character pointer**, a value from **one byte** is extracted.

If it is a **float pointer**, a value from **four bytes** is extracted.

Hence to store a particular type of data suitable pointer should be used.

7.7 Pointer arithmetic

/* pointer arithmetic - example 1 */

Program 7.6

```
void main()
{
    int a = 15;
    char ch = 'x';
    float f = 9.7;

    int* p = &a;
    char* q = &ch;
    float* r = &f;

    printf("Before incrementation : %d, %c, %f\n", a, ch, f);
    a++;
    ch++;
    f++;
    printf("After incrementation : %d, %c, %f\n\n", a, ch, f);
    printf("Before incrementation : %u, %u, %u\n", p, q, r);
    p++;
    q++;
    r++;
    printf("After incrementation : %u, %u, %u", p, q, r);
}
```

Output

```
Before incrementation:    15, x, 9.7
After incrementation :    16, y, 10.7

Before incrementation :   100, 200, 300
After incrementation :    102, 201, 304
```

Explanation

When a pointer is incremented it moves to the next location of its type. So,

When an integer pointer is incremented , the change is +2

When a character pointer is incremented , the change is +1

When a float pointer is incremented , the change is +4

You will appreciate this logic in dynamic memory management.

/* pointer arithmetic - example 2 */

Program 7.7

```
void main()
{
    int a = 10, b = 20;
    int* p = &a, *q = &b;

    printf("%u, %u\n", p, q);

    printf("%u\n", p + 2);
    printf("%u\n", p - 2);
    /* printf("%u\n", p * 2); illegal */

    printf("%u\n", q - p);
    /* printf("%u", p + q); illegal */
}
```

Output

```
65498, 65500
65502
65494
1
```

Explanation

10	20	65498	65500
a	b	p	q
65498	65500	65502	65504

Only the following pointer arithmetic is possible:

- a) Adding an integer to a pointer (pointer + integer or integer + pointer)
- b) Subtracting an integer from a pointer.(pointer – integer or integer – pointer)
- c) The only arithmetic operation that can be made **between two pointers** is **subtraction(pointer – pointer)** to find the distance between two pointers.

Remaining arithmetic is not possible because it would not make any sense (eg : adding two addresses is meaningless).

7.8 Types of function calls

There are two types of function call mechanisms :

- Call by value
- Call by address/reference

Call by value

Calling a function by passing the **value of an actual argument** is called as call by value(till now we have been dealing with this style).

/ call by value */*

Program 7.8

```
void swap(int a, int b)
{
    int t;

    t = a;
    a = b;
    b = t;
}

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

    printf("Before swapping : %d, %d\n", a, b);
    swap(a, b);
    printf("After swapping : %d, %d\n", a, b);
}
```

Output

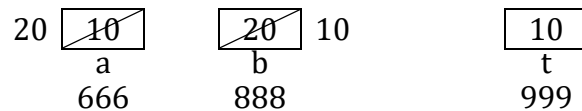
Before swapping: 10, 20
After swapping: 10, 20

Explanation

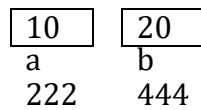
Surprised?

The **a**, **b**, in **main()** and **swap()** functions are different.

swap()



main()



So the changes made to **a** and **b** in the **swap()** function has no affect on **a** and **b** of **main()**.

Call by address/reference

Calling a function by passing the **address of an actual argument** is called as call by address/reference.

/ call by address - example 1 */*

Program 7.9

```
void swap(int* a, int* b)
{
    int t;

    t = *a;
    *a = *b;
    *b = t;
}
void main()
{
    int a = 10, b = 20;

    printf("Before swapping : %d, %d\n", a, b);
    swap(&a, &b);
    printf("After swapping : %d, %d\n", a, b);
}
```

Output

Before swapping: 10, 20

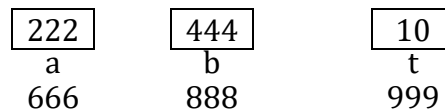
After swapping: 20, 10

Explanation

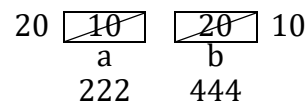
Even now the **a**, **b**, in **main()** and **swap()** functions are different, but still swapping is possible.

Let's see how.

swap()



main()



So through call by address we can change the values of actual arguments by de-referencing the formal arguments.

/* call by address - example 2 */

Program 7.10

```
void changeMe(int* a)
{
    *a = 72;
}

void main()
{
    int a = 145;

    printf("Before : %d\n", a);
    changeMe(&a);
    printf("After : %d", a);
}
```

Output

Before : 145

After : 72

7.9 Array and pointer notationsLet's consider an array: `int a[] = {4, 8, 10, 19, 7};`

Suppose that it is organized as :

4	8	10	19	7
100	102	104	106	108

We know that `a[0]` is 4
 `&a[0]` is 100

then what is **a**?Name of an array is nothing but the address of its first element. i.e., **a is 100**

Putting it all together

elements in pointer notation	----->	<code>*(a+0)</code>	<code>*(a+1)</code>	<code>*(a+2)</code>	<code>*(a+3)</code>	<code>*(a+4)</code>
elements in array notation	----->	<code>a[0]</code>	<code>a[1]</code>	<code>a[2]</code>	<code>a[3]</code>	<code>a[4]</code>
elements	----->	4	8	10	19	7
index/subscript	----->	0	1	2	3	4
addresses	----->	100	102	104	106	108
addresses in array notation	----->	<code>&a[0]</code>	<code>&a[1]</code>	<code>&a[2]</code>	<code>&a[3]</code>	<code>&a[4]</code>
addresses in pointer notation	----->	<code>(a+0)</code>	<code>(a+1)</code>	<code>(a+2)</code>	<code>(a+3)</code>	<code>(a+4)</code>

Conclusion**Array notation Pointer notation**

<code>a[i]</code>	<code>*(a + i)</code>
<code>& a[i]</code>	<code>(a + i)</code>

7.10 Dynamic memory management – malloc(), calloc(), free(), realloc()

Difficulty with arrays

Let's consider the following program:

```
/* using arrays */
```

Program 7.11

```
void main()
{
    int a[100], n, i;

    printf("How many numbers do u want to enter:");
    scanf("%d", &n);

    printf("Enter numbers:");
    for(i = 0 ; i < n ; i++ )
        scanf("%d", &a[i]);

    printf("The entered numbers are\n");
    for(i = 0 ; i < n ; i++)
        printf("%d\n", a[i]);
}
```

Explanation

Let's consider the following cases:

- a) Suppose that, during runtime, the requirement is only 20 integers, then the memory allocated for the other 80 integers gets wasted.
- b) Suppose that, during runtime, the requirement is 115 integers, then we cannot change the size of the array.

So there is no flexibility in arrays. Arrays are used only when we know in advance (at the time of program development) the approximate size of the array. If we don't know its (the collection) size in advance we go for **dynamic memory management(DMM)**. Using DMM exact amount of memory can be allocated. Let's see how this can be achieved.

```
/* dynamic memory management */
```

Program 7.12

```
void main()
{
    int* a, n, i;

    printf("How many numbers do u want to enter:");
    scanf("%d", &n);

    a = (int*)malloc( n * sizeof(int) );

    printf("Enter numbers:");
    for(i = 0 ; i < n ; i++ )
        scanf("%d", a + i );

    printf("The entered numbers are\n");
    for(i = 0 ; i < n ; i++)
        printf("%d\n", *(a + i) );

    free(a);
}
```

Explanation

a) **malloc()** is a library function which allocates specified amount of memory from the **heap** and returns the base address(starting address) of the allocated memory. **Heap** is a dynamically allocated area of memory where the program runs.

b)

typecasting

a = (int*)malloc(n * sizeof(int));

malloc() returns a **void** pointer. So it is necessary to typecast it to an appropriate type depending on the type of pointer we have on the left side of the assignment operator. Very soon we will discuss **void** pointers.

c) Can't we use **2** in place of **sizeof(int)** ? We can, but then the program will not be portable to higher – bit compilers where an integer occupies 4 bytes (32- bit compilers)

d) Instead of **malloc()**, we can even use **calloc()** for **dynamic memory allocation(DMA)**.

```
a = (int*)calloc(n, sizeof(int));
```

Difference

calloc() function :

- i) takes two arguments
- ii) initializes the reserved memory with zeros

e) **free()**

We know that when a variable goes out of its scope memory allocated for that variable is automatically de-allocated by the compiler. However dynamically allocated memory remains allocated as long as the program is under execution. Hence when its need is over we have to explicitly de-allocate it(otherwise it could lead to memory leaks(very soon we will discuss this)) so that this memory can to be re used for some other purpose. To accomplish this job, we use **free()** function which de-allocates the dynamically allocated memory. Assume a situation where there are many instructions to be executed, then you will appreciate the need of freeing the memory.

The following program illustrates the use of **free()** function :

```
/* use of free */
```

Program 7.13

```
void main()
{
    int* a, n, i, *b;

    printf("How many numbers do u want to link to 'a' : ");
    scanf("%d", &n);

    a = (int*)malloc( n * sizeof(int) );

    printf("Enter numbers:");
    for(i = 0 ; i < n ; i++ )
        scanf("%d", a + i );

    printf("The entered numbers and their addresses are\n");
    for(i = 0 ; i < n ; i++)
        printf("%d [ %u ]\n", *(a + i), a + i );

    free(a);
}
```

```
    printf("How many numbers do u want to link to 'b' : ");
    scanf("%d", &n);

    b = (int*)malloc( n * sizeof(int) );

    printf("Enter numbers:");
    for(i = 0 ; i < n ; i++ )
        scanf("%d", b + i );

    printf("The entered numbers and their addresses are\n");
    for(i = 0 ; i < n ; i++)
        printf("%d [ %u ]\n", *(b + i), b + i );
}
```

Output

Case i) without using free()

```
How many numbers do u want to link to 'a' : 5
Enter numbers: 1    2    3    4    5
The entered numbers and their addresses are
1 [3462]
2 [3464]
3 [3466]
4 [3468]
5 [3470]
```

```
How many numbers do u want to link to 'b' : 5
Enter numbers: 6    7    8    9    10
The entered numbers and their addresses are
6 [3478]
7 [3480]
8 [3482]
9 [3484]
10 [3486]
```

Case ii) with using free()

```
How many numbers do u want to link to 'a' : 5
Enter numbers: 1    2    3    4    5
The entered numbers and their addresses are
1 [3462]
2 [3464]
3 [3466]
```


4 [3468]

5 [3470]

How many numbers do u want to link to 'b' : 5

Enter numbers: 6 7 8 9 10

The entered numbers and their addresses are

6 [3462]

7 [3464]

8 [3466]

9 [3468]

10 [3470]

Note that when we use **free()** there is a chance that the same memory will be reused(which can be observed in the second case).

/* use of realloc */

Program 7.14

```
void main()
{
    int* a, n, i, m;

    printf("How many numbers do u want to link to 'a' : ");
    scanf("%d", &n);

    a = (int*)malloc( n * sizeof(int) );

    printf("Enter numbers:");
    for(i = 0 ; i < n ; i++ )
        scanf("%d", a + i );

    printf("The entered numbers and their addresses are\n");
    for(i = 0 ; i < n ; i++)
        printf("%d [ %u ]\n", *(a + i), a + i );

    /* ----- */

    printf("How many more do u want to link to 'a' : ");
    scanf("%d", &m);

    a = (int*)realloc(a, (n+m) * sizeof(int) );
```

```
printf("Enter numbers : ");
for(i = 0 ; i < m ; i++)
scanf("%d", a + n + i);

printf("All the numbers and their addresses are\n");
for(i = 0 ; i < n + m ; i++)
    printf("%d [ %u ]\n", *(a + i), a + i );

}
```

Explanation

At times we may need to adjust the size of the allocated block (allocated using **malloc()** or **calloc()**). Then we use the library function **realloc()**.

realloc() function adjusts the size of the allocated block to the specified new size and returns the address of the reallocated block, which might be different than the address of the original block. If the base address of the newly allocated block is different than the existing base address (returned by the **malloc()** function) then the **realloc()** function automatically copies the existing contents to the new location.

Static memory allocation Vs Dynamic memory allocation

SMA	DMA
<ol style="list-style-type: none">1. Memory is allocated during execution.2. Arrangement is done at compilation.3. Memory is allocated from stack area.4. Memory is de-allocated by the compiler.	<ol style="list-style-type: none">1. Memory is allocated during execution.2. Arrangement is done during execution.3. Memory is allocated from heap area.4. Memory is de-allocated using free()

7.11 Passing arrays to functions

Passing array elements to functions is a regular activity in C programming (for reading/displaying/processing the array elements).

Array elements can be passed to a function in several ways. Let's discuss one by one.

Suppose that, our requirement is to print the array elements by a user-defined function.

/* passing arrays to functions - method 1 */

Program 7.15

```
void fun(int x)
{
    printf("%d\t", x);
}

void main()
{
    int a[ ] = {9, 43, 2, 6, 3}, i;

    for(i = 0 ; i < 5 ; i++)
        fun( a[i] );
}
```

Explanation

Here, we are passing an individual array element at a time to the function **fun()** and getting it printed in the **fun()** function.

This method is not an efficient one because the **fun()** function gets called five times and each time the control has to be passed to the **fun()** function and return to the **main()** function. So a lot of time gets wasted.

/* passing arrays to functions - method 2 */

Program 7.16

```
void fun(int a, int b, int c, int d, int e)
{
    printf("%d\t%d\t%d\t%d\t%d", a, b, c, d, e);
}

void main()
{
    int a[ ] = {9, 43, 2, 6, 3}, i;

    fun( a[0], a[1], a[2], a[3], a[4] );
}
```

Explanation

Here we have passed all the array elements at once into the **fun()** function. In this method we have avoided the wastage of time by calling the function only once. But imagine an array of 300 elements, is it not difficult to pass the information in this fashion. Hence this method too is not an appropriate one.

/* passing arrays to functions - method 3 */

Program 7.17

```
void fun(int* p, int n)
{
    int i;
    for(i = 0 ; i < n ; i++)
        printf("%d\t", *(p + i) );
}

void main()
{
    int a[ ] = {9, 43, 2, 6, 3}, i;

    fun(a, 5);
}
```

Explanation

To get the good of both methods, we pass the entire array by passing the base address of the array and a number representing the total number of elements. This method is **efficient** and **easy** to use as well.

7.12 To return more than one value from a function

We can return only one value from a function. However, indirectly we can return more than one value from a function. There are three methods:

- using a dummy array
- using pointers
- using dynamic memory allocation

Let's discuss one by one...

```
/* return more than one value from a function - method 1 */
/* using a dummy array */
```

Program 7.18

```
void more(int a[], int n, int b[])
{
    int i, sum = 0, max = 0, min = 32767;

    for(i=0;i<n;i++)
    {
        sum += a[i];

        if(a[i] > max)
            max = a[i];

        if(a[i] < min)
            min = a[i];
    }

    b[0] = sum;
    b[1] = max;
    b[2] = min;
}

void main()
{
    int a[ ] = { 24, 62, 875, 21, 75}, b[3];

    more(a, 5, b);

    printf("Sum : %d\n", b[0]);
    printf("Max : %d\n", b[1]);
    printf("Min : %d\n", b[2]);
}
```

Explanation

main()

a

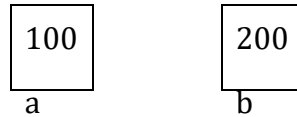
24	62	875	21	75
100	102	104	106	108

b

1057	875	21
gar	gar	gar
200	202	204

swap()

int a[] is nothing but another way of representing **int* a**. **int b[]** is nothing but another way of representing **int* b**



b[0] = sum; is same as *(b+0) = sum; is same as value at 200 = 1057

b[1] = max; is same as *(b+1) = max; is same as value at 202 = 875

b[2] = min; is same as *(b+2) = min; is same as value at 204 = 21

/* return more than one value from a function - method 2 */
/* using pointers */

Program 7.19

```
void more(int* a, int n, int* x, int* y, int* z)
{
    int i, sum = 0, max = 0, min = 32767;
    for(i=0;i<n;i++)
    {
        sum += a[i];

        if(a[i] > max)
            max = a[i];

        if(a[i] < min)
            min = a[i];
    }

    *x = sum;
    *y = max;
    *z = min;
}

void main()
{
    int a[ ] = { 24, 62, 875, 21, 75}, x, y, z;

    more(a, 5, &x, &y, &z);
    printf("Sum : %d\n", x );
    printf("Max : %d\n", y );
    printf("Min : %d\n", z );
}
```

/* return more than one value from a function - method 3 */
/* using dynamic memory allocation */

Program 7.20

```
int* more(int* a, int n)
{
    int i, sum = 0, max = 0, min = 32767;
    int* m;

    for(i=0;i<n;i++)
    {
        sum += a[i];

        if(a[i] > max)
            max = a[i];

        if(a[i] < min)
            min = a[i];
    }

    m = (int*)malloc( 3 * sizeof(int) );

    *(m + 0) = sum;
    *(m + 1) = max;
    *(m + 2) = min;

    return m;
}

void main()
{
    int a[ ] = { 24, 62, 875, 21, 75}, *p;

    p = more(a, 5);

    printf("Sum : %d\n", *(p + 0) );
    printf("Max : %d\n", *(p + 1) );
    printf("Min : %d\n", *(p + 2) );

    free(p);
}
```

Explanation

```
m = (int*)malloc( 3 * sizeof(int) );
```

As this is a dynamically allocated memory it persists as long the program is under execution.

7.13 Dangling pointer and memory leak

Dangling pointer (Wild pointer)

It is a pointer to a memory location that is freed. In short, we can say object destroyed, path present.

Let's discuss it in detail:

Dangling pointers in programming are pointers whose objects have since been deleted or de-allocated, without modifying the value of the pointer.

In many languages (particularly the C/C++ programming languages) deleting(freeing) an object from memory doesn't alter any associated pointers. The pointer still points to the location in memory where the data/object was, even though the object/data has since been deleted and the memory may now be used for other purposes. A pointer in such a situation is called a dangling pointer.

Using a dangling pointer under the assumption that the object it points to is still valid can cause unpredictable behavior. The use of dangling pointer can also result in the silent corruption of unrelated data.

To avoid bugs (logical errors) of this kind, one common programming technique is to set pointers to the NULL (0) pointer. When the NULL pointer is de-referenced/freed there is no potential for data corruption or unpredictable behavior.

Note that de-referencing a NULL pointer is compiler dependent. Most of the Compilers terminate the program on de-referencing a NULL pointer.

/* dangling pointer - example 1 */

Program 7.21

```
void main()
{
    int *p;

    {
        int x = 10;
        p = &x;
    }

    printf("%d", *p);
}
```

Output

May not be 10

Explanation

We know that the 'x' dies as soon as the control comes out of the inner block, but still 'p' points to 'x'. Hence 'p' is called as dangling pointer.

/* dangling pointer - example 2 */

Program 7.22

```
int* fun()
{
    int a = 7214; /* use static to avoid dangling pointer */

    return &a; /* returning the address of a local auto variable leads to dangling
pointer */
}

void main()
{
    int *p;

    p = fun();
    printf("some piece of code\n");
    printf("%d", *p);
}
```

Output

Some piece of code
May not be 7214

Explanation

We know that 'a' dies as soon as the control comes out of the **fun()** function, but 'p' points to 'a'.
Hence 'p' is called as dangling pointer.

Moral : We should not return the address of a local *auto* variable.

/ dangling pointer - example 3 */*

Program 7.23

```
#define NULL 0

int* fun()
{
    int* a, n, i;

    printf("How many numbers do u want to enter:");
    scanf("%d", &n);

    a = (int*)malloc( n * sizeof(int) );

    printf("Enter numbers:");
    for(i = 0 ; i < n ; i++ )
        scanf("%d", a + i );

    printf("The entered numbers are\n");
    for(i = 0 ; i < n ; i++)
        printf("%d\n", *(a + i) );

    free(a); /* now 'a' is a dangling pointer */

    a = NULL; /* now 'a' is a safe pointer */

    return a;
}
```

```
void main()
{
    int *b, n , i;
    b = fun();

    printf("How many no.s:");
    scanf("%d", &n);

    printf("Enter numbers:");
    for(i = 0 ; i < n ; i++ )
        scanf("%d", b + i ); /* overwriting into unreserved area [ may be disastrous ]
*/

    printf("The entered numbers are\n");
    for(i = 0 ; i < n ; i++)
        printf("%d\n", *(b + i) ); /* accessing the unreserved area */
}
```

Output

Some output...

Null pointer assignment

Explanation

a) Had we not used **a = NULL;** , 'b' would have still pointed to the de-allocated location,
which would have further lead to overwriting into unreserved area.

b) Null pointer assignment

It is a runtime error. It occurs when our program tries to access an illegal memory location. Illegal location means either the location is in the OS address space or in the other processes memory space.

Memory leak

It is a situation where allocated memory is not freed although it is never used again. In short, we can say object present, path destroyed.

/* memory leak - example 1 */**Program 7.24**

```
void main()
{
    int* a, n, i, b = 536;

    printf("How many numbers do u want to enter:");
    scanf("%d", &n);

    a = (int*)malloc( n * sizeof(int) );
    printf("Enter numbers:");

    for(i = 0 ; i < n ; i++ )
        scanf("%d", a + i );

    printf("The entered numbers are\n");

    for(i = 0 ; i < n ; i++)
        printf("%d\n", *(a + i) );

    a = &b; /* memory leak */

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

    printf("Elements are\n");

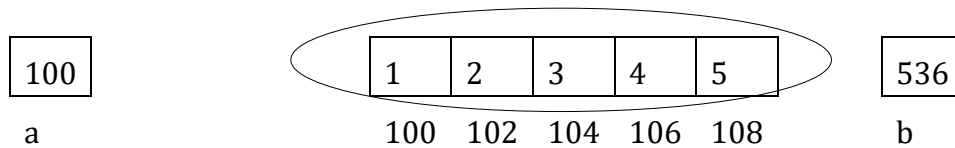
    for(i = 0 ; i < n ; i++)
        printf("%d\t", *( a + i ) ); /* unexpected results */
}
```

Output

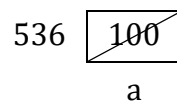
```
How many numbers do u want to enter: 5
Enter numbers: 1 2 3 4 5
The entered numbers are
1
2
3
4
5
536
Elements are
536  -28  285  1  -30
```

Explanation

```
a = (int*)malloc( n * sizeof(int) );
```



As soon as the statement **a = &b;** gets executed, address of 'b' gets stored in 'a'



As the circled block is dynamically allocated memory, it persists as long the program is under execution. Since 'a' holds 536 there is no way to de-allocate the dynamically allocated memory, resulting in memory leak.

To avoid memory leak, we should say **free(a);** just before changing the value of 'a'.

/* memory leak - example 2 */

Program 7.25

```
#define NULL 0
void main()
{
    char* st1, *st2;

    st1 = (char*)malloc( 100 * sizeof(char) );
    st2 = (char*)malloc( 100 * sizeof(char) );

    puts("Enter string 1 : ");
    gets(st1);

    st2 = st1;    /* memory leak */

    printf("%s\n%s", st1, st2);
    free(st1);    /* will be successfull */
    free(st2);    /* will fail */
}
```

Explanation

As **st2**'s original value is lost, it would result in memory leak.

```
/* memory leak - example 3 */
```

Program 7.26

```
#define NULL 0
void fun3()
{
    int* a, n, i;

    printf("How many numbers do u want to enter:");
    scanf("%d", &n);
    a = (int*)malloc( n * sizeof(int) );
    printf("Enter numbers:");
    for(i = 0 ; i < n ; i++ )
        scanf("%d", a + i );

    printf("The entered numbers are\n");
    for(i = 0 ; i < n ; i++)
        printf("%d\n", *(a + i) );

    /* memory leak as "free(a);" is absent*/
}

void fun2()
{
    /* some code here */
    fun3();
    /* some code here */
}

void fun1()
{
    /* some code here */
    fun2();
    /* some code here */
}

void main()
{
    /* some code here */
    fun1();
    /* some code here */
}
```

Explanation

Even though the dynamically allocated memory can be used only in the **fun()** function, this memory persists as long as the program is under execution, resulting in memory leak.

To avoid memory leak, we should say **free(a)**, somewhere in the **fun()** function when it is no more required.

7.14 Recursion

Calling function by itself is called as recursion. Let's understand it through programs:

```
/* meaningless recursion */
```

Program 7.27

```
void fun()
{
    printf("Hello\t");
    fun();
}

void main()
{
    fun();
}
```

Output

```
Hello
Hello
Hello
...
```

Explanation

The **fun()** function gets executed infinite times. If the **fun()** function consists some local data the program will go out of memory, because at some stage the memory would get exhausted (because the local data of a function block dies only after the complete execution of the function). So this type of recursion is useless and dangerous. Useful recursion is demonstrated in the following programs.

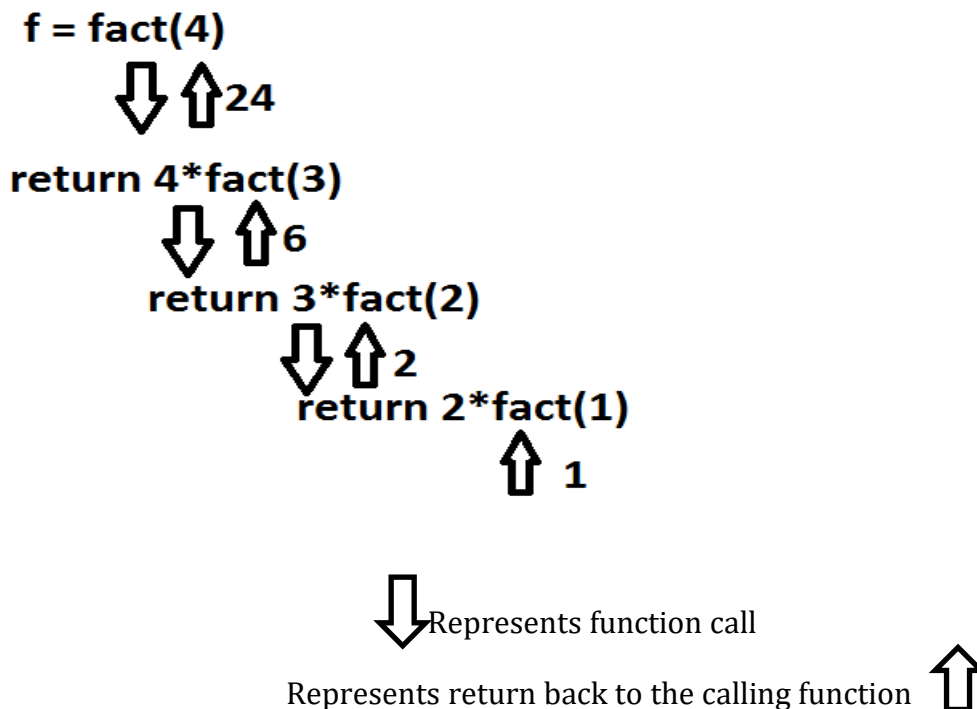
```
/* recurrSION - example 1 */
/* factorial of a number*/
```

Program 7.28

```
int fact(int n)
{
    if( n <= 1) <----- base case
        return 1;
    return n * fact(n-1);<----- recursive case
}

void main()
{
    int n, f;
    printf("Enter a number : ");
    scanf("%d", &n);
    f = fact(n);
    printf("Factorial = %d", f);
}
```

Explanation- Analysis



- a) For every recursive function the following two properties must be satisfied:
There must be a **base criteria** for which the function doesn't call itself. Each time the function calls itself(**recursion**) it must be closer to the base criteria.
- b) For each function call separate stacks will be created which contains the local data, return pointer(calling functions address, so that control can return back to its caller) of the function. In our program as there are four **fun()** function calls four different stacks will be created, they will be destroyed as soon as the control returns to their respective calling functions.

```
/* recursion - example 2 */  
/* binary equivalent of a decimal number */
```

Program 7.29

```
void binary(int n)  
{  
    if( n/2 )  
        binary( n/2 );  
  
    printf("%d\t", n % 2);  
}  
  
void main()  
{  
    int n;  
  
    printf("Enter a number : ");  
    scanf("%d", &n);  
  
    binary( n );  
}
```

Explanation**Analysis**

binary(25)	
↓	25%2 will be printed(i.e., 1)
binary(12)	↑
↓	12%2 will be printed(i.e., 0)
binary(6)	↑
↓	6%2 will be printed(i.e., 0)
binary(3)	↑
↓	3%2 will be printed(i.e., 1)
binary(1)	↑
→	1%2 will be printed(i.e., 1)

Note that the **printf()** statement will be executed in the reverse direction of function call sequence.

```
/* recurrSION - example 3 */
/* sum of array elements */
```

Program 7.30

```
int sumOfArrayElements(int* a, int n)
{
    if(n == 1)
        return a[0];
    return a[n-1] + sumOfArrayElements(a, n - 1);
}

void main()
{
    int a[10], n, i, s;

    printf("How many:");
    scanf("%d", &n);

    for(i=0; i<n; i++)
        scanf("%d", &a[i]);

    s = sumOfArrayElements(a, n);
    printf("Sum = %d", s);
}
```

```
/* recurrision - example 4 */
/* nth fibonacci number */
```

Program 7.31

```
int fib(int n)
{
    if(n == 0 || n == 1)
        return n;

    return fib(n-2) + fib(n-1);
}

void main()
{
    int n1, n2;

    printf("Enter a number:");
    scanf("%d", &n1);
    n2 = fib(n1);
    printf("nth fibonacci number is : %d", n2);
}
```

Explanation

Note that first the left function call gets executed and then the right function call gets executed.

Analysis

$n2 = \text{fib}(7);$

$\text{fib}(5) \quad + \quad \text{fib}(6)$

$\text{fib}(3) \quad + \quad \text{fib}(4) \quad \quad \text{fib}(4) \quad + \quad \text{fib}(5)$

$\text{fib}(1) + \text{fib}(2) \quad \text{fib}(2) \quad + \quad \text{fib}(3)$ similar to the left part
of this figure

Note that a problem that can be solved recursively can also be solved iteratively (using loops).

Advantages of recursion

- Sometimes the code is compact than its looping counterpart
- Sometimes the recursive logic mirrors the real problem.
- Some problems are much easier to solve recursively than iteratively.

Disadvantages of recursion

- Understanding the working is difficult than its looping counterpart.
- Program execution is far slower.
- There is no portable way to tell how deep recursion can go without causing trouble (depends on how much “stack space” the machine has) and there is no way to recover from too-deep recursion (a “stack overflow”).

7.15 Array of pointers

The way there can be an array of integers or an array of character, similarly there can be an array of pointers. Since a pointer variable contains an address, an array of pointers would be nothing but a collection of addresses. Let's understand this through programs:

/* array of pointers - example 1*/

Program 7.32

```
void main()
{
    int a = 10, b = 20, c = 30, d = 40, i;
    int* p[4];
    p[0] = &a;
    p[1] = &b;
    p[2] = &c;
    p[3] = &d;

    for(i = 0 ; i < 4 ; i++)
        printf("%d\t", *( p[i] ) );
}
```

Output

10 20 30 40

```
/* array of pointers - example 2 */
```

Program 7.33

```
void main()
{
    char* elec[] = {"Panasonic", "Samsung", "Philips", "LG", "Sony"};
    int i;
    for(i=0;i<5;i++)
        printf("%s\n", elec[i]);
}
```

Output

```
Panasonic
Samsung
Philips
LG
Sony
```

Explanation

The important advantage of the pointer array is that the rows of the array may be of different lengths.

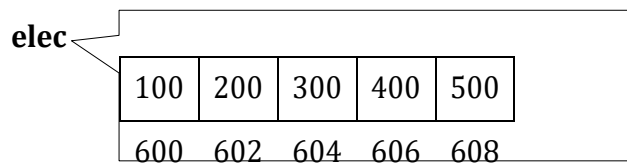
P	a	n	a	s	o	n	i	c	\0
100	101	102	103	104	105	106	107	108	109

S	a	m	s	u	n	g	\0
200	201	202	203	204	205	206	207

P	h	i	l	i	p	s	\0
300	301	302	303	304	305	306	307

L	G	\0
400	401	402

S	o	n	y	\0
500	501	502	503	504



No. of bytes allocated : **44 bytes**(34 (10+8+8+3+5) + 10 (5 * 2)) with those for a 2-D array:

char elect[][15] = {"Panasonic", "Samsung", "Philips", "LG", "Sony"};

No. of bytes allocated : **75 bytes**(5 * 15)

While dealing with matrices, an array of **int** pointers provides flexibility, which is not there in 2-D **int** arrays. Let's understand this through an example:

/* array of pointers - example 3 */

/* Dynamic memory management for a matrix */

Program 7.34

```
void main()
{
    int *m[10], r, c, i, j;

    printf("How many rows and cols : ");
    scanf("%d%d", &r, &c);

    /* memory allocation */

    for(i = 0 ; i < r ; i++)
        m[i] = (int*)malloc( c * sizeof(int) );

    printf("Enter elements into the matrix\n");

    for(i = 0 ; i < r ; i++)
        for(j = 0 ; j < c ; j++)
            scanf("%d", m[i] + j);

    printf("The matrix is\n");

    for(i = 0 ; i < r ; i++)
    {
        for(j = 0 ; j < c ; j++)
            printf("%3d", *(m[i] + j) );

        printf("\n");
    }
}
```

```
/* memory de-allocation */
for(i=0;i<r;i++)
    free(m[i]);
}
```

Explanation

Let's consider the following two cases:

case i)

int m[10][10];

We know that **200 (100 * 2) bytes** gets reserved.

During execution, say, $r = 3$ and $c = 4$ then only **24 (12 * 2) bytes** would be utilized.

Conclusion

Total memory allocated : 200 bytes

Total memory utilized : 24 bytes

Total memory wasted : 186 bytes

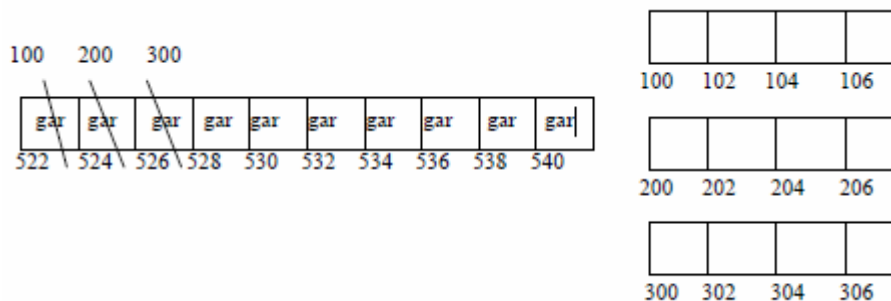
case ii)

int* m[10];

We know that **20 (10 * 2) bytes** gets reserved for 'm'.

During execution, say, $r = 3$ and $c = 4$ then another **24 (12 * 2) bytes** gets reserved (for storing the matrix elements).

Let's understand the statement : **scanf("%d", m[i] + j);**



We know that to the `scanf()` we have to specify the addresses of the memory locations where the values have to be placed.

$m[0] + 0$	$100 + 0$	100
$m[0] + 1$	$100 + 1$	102
$m[0] + 2$	$100 + 2$	104
$m[0] + 3$	$100 + 3$	106
$m[1] + 0$	$200 + 0$	200
$m[1] + 1$	$200 + 1$	202
$m[1] + 2$	$200 + 2$	204
$m[1] + 3$	$200 + 3$	206

and so on

Conclusion

Total memory allocated : 44 (20 + 24) bytes
Total memory utilized : 24 bytes
Total memory wasted : 20 bytes

Out of these **20 bytes** , **6 (3 * 2)** bytes have been utilized indirectly. So, actual wastage : **14 bytes only**

Finally all the dynamically allocated memory is de- allocated using the **free()** function.

7.16 Pointer to a pointer

Using pointer to a pointer, least amount of memory can be wasted while dealing with matrices.

```
/* pointer to a pointer */  
/* Dynamic memory management for a matrix */
```

Program 7.35

```
void main()  
{  
    int **m, r, c, i, j;  
  
    printf("How many rows and col.s : ");  
    scanf("%d%d", &r, &c);  
  
    /* memory allocation */  
    m = (int**)malloc( r * sizeof(int) );  
  
    /* memory allocation */  
    for(i = 0 ; i < r ; i++)  
        *(m+i) = (int*)malloc( c * sizeof(int) );
```

Because 'm' is a pointer to an integer pointer.

or m[i]


```
printf("Enter elements into the matrix\n");

for(i = 0 ; i < r ; i++)
    for(j = 0 ; j < c ; j++)
        scanf("%d", m[i] + j);

printf("The matrix is\n");

for(i = 0 ; i < r ; i++)
{
    for(j = 0 ; j < c ; j++)
        printf("%3d", *(m[i] + j) );

    printf("\n");
}

/* memory de-allocation */

for(i=0;i<r;i++)
    free(m[i]);

/* memory de-allocation */

free(m);
}
```

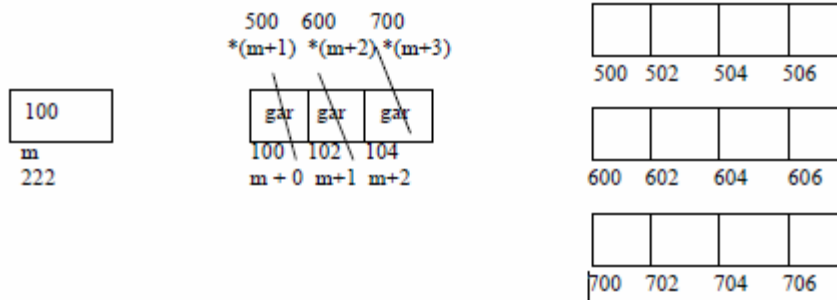
Explanation

int m;**

We know that **2 bytes** gets reserved for '**m**'.

During execution, say, $r = 3$ and $c = 4$ then

First **6 ($3 * 2$) bytes** gets reserved for storing the base addresses of each of the rows,
next **24 ($12 * 2$) bytes** gets reserved (for storing the matrix elements).



Conclusion

Total memory allocated : 32 (2 + 6 + 24) bytes
 Total memory utilized : 24 bytes
 Total memory wasted : 8 bytes

Even these 8 (2 + 6 (3 * 2)) bytes have been utilized indirectly.
 So, actual wastage: **0 bytes only**

Finally all the dynamically allocated memory is de- allocated using the **free()** function.

7.17 Constant pointers

```
/* constant pointers - example 1 */
/* non - constant pointer to a non-constant data */
```

Program 7.36

```
void main()
{
    int a = 10, b = 20;
    int *p;
    p = &a;

    printf("%u, %u\n", a, p);

    *p = 56;      /* valid */
    p = &b;      /* valid */

    printf("%u, %u\n", a, p);
}
```

Explanation

***p** and **p** both can be changed.

```
/* constant pointers - example 2 */  
/* non - constant pointer to a constant data */
```

Program 7.37

```
void main()  
{  
    int a = 10, b = 20;  
    const int *p;  
  
    p = &a;  
  
    printf("%u, %u\n", a, p);  
  
    *p = 56;    /* invalid */  
    p = &b;    /* valid */  
  
    printf("%u, %u\n", a, p);  
}
```

Explanation

***p** cannot be changed
p can be changed

```
/* constant pointers - example 3 */  
/* constant pointer to a non - constant data */
```

Program 7.38

```
void main()  
{  
    int a = 10, b = 20;  
    int* const p = &a;  
  
    printf("%u, %u\n", a, p);  
  
    *p = 56;    /* valid */  
    p = &b;    /* invalid */  
  
    printf("%u, %u\n", a, p);  
}
```

Explanation

***p** can be changed
p cannot be changed

```
/* constant pointers - example 4 */  
/* constant pointer to a constant data */
```

Program 7.39

```
void main()  
{  
    int a = 10, b = 20;  
    const int* const p = &a;  
  
    printf("%u, %u\n", a, p);  
  
    *p = 56;      /* invalid */  
    p = &b;      /* invalid */  
  
    printf("%u, %u\n", a, p);  
}
```

Explanation

***p** cannot be changed
p cannot be changed

7.18 void pointer

```
/* void pointers */
```

Program 7.40

```
void main()  
{  
    int a = 423;  
    char ch = 'm';  
    float f = 12.5;  
  
    void* p;  
  
    printf("%d\n", sizeof(p) );  
  
    p = &a;  
    printf("%d\n", *(int*)p); /* extracts data from two bytes */
```

```
p = &ch;
printf("%c\n", *(char*)p); /* extracts data from one byte */

p = &f;
printf("%f\n", *(float*)p); /* extracts data from four bytes */
}
```

Explanation

- a) Ordinary variables cannot be of **void** type, because during compilation, the compiler must know number of bytes to be allocated for a variable. However pointer variables can be of **void** type because we know that for any pointer variable **2** bytes gets allocated.
- b) A **void** pointer(also called as generic pointer) is capable of holding any type of variable's address.
- c) A void pointer cannot be de-referenced directly because while de-referencing, the compiler should know from how many bytes, data has be extracted. So typecasting is required while de-referencing a **void** pointer.

7.19 Function pointers

We know that every variable (except a **register** variable) has an address in **RAM**, and also we can store that address in a pointer variable. Similarly every function has an address in **RAM**, and also we can store that address. To store an address of a function, we have to use a function pointer. Let's begin with printing the addresses of functions.

/* function pointers - example 1 */

Program 7.41

```
void fun1()
{
    printf("Ameerpet\n");
}

void fun2()
{
    printf("KPHB\n");
}
```

```
void fun3()
{
    printf("Gachibowli\n");
}

void main()
{
    printf("%u\n", main);
    printf("%u\n", fun1);
    printf("%u\n", fun2);
    printf("%u\n", fun3);
}
```

Output

```
545
506
519
532
```

Explanation

Note that to obtain the address of a function we just have to mention the name of the function, as has been done in the **printf()** function above. This is similar to mentioning the name of an array to get its base address.

/* function pointers - example 2 */

Program 7.42

```
void fun()
{
    printf("I am fun\n");
}

void main()
{
    void (*fp)();

    fp = fun;
    fp();
}
```

Output

```
I am fun
```

Explanation

a) **void (*fp)();**

this is the way we define a function pointer. It means, that **fp** is a function pointer, which can hold an address of a function whose return type is **void** and which doesn't take any arguments.

b) Why is **fp** enclosed in parenthesis?

Had we not enclosed **fp** in parenthesis it would have looked like:

void* fp();

doesn't it look like a function prototype? where **void*** is the return type, and **fp** is the name of the function. Hence parentheses are required while defining a function pointer.

c) **fp = fun;**

The above statement assigns the address of **fun()** function to the function pointer **fp**.

d) To invoke the function we are just required to write the statement,

fp(); or **(*fp)();**

Uses of function pointers

- a) In passing functions as arguments to other functions.
- b) In writing viruses, or vaccine.
- c) In the implementation of dynamic binding in C++
- d) Used very heavily in C#.NET as delegates etc

/* function pointers - example 3 */

Program 7.43

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

void main()
{
    float (*fp)(float, float);
    float x;
```

```
    fp = sum;

    x = fp(2.4, 1.1);

    printf("%f", x);
}
```

Explanation

fp is a function pointer which is capable of holding an address of a function whose return type is *float* and which takes two *float* type of arguments.

```
/* function pointers - example 4 */
/* array of function pointers */
```

Program 7.44

```
void fun1()
{
    printf("Sunami 2004 December\n");
}

void fun2()
{
    printf("Vizag HudHud Cyclone 2014 October\n");
}

void fun3()
{
    printf("Chennai Rains 2015 November\n");
}

void main()
{
    void (*fp[3])() = {fun1, fun2, fun3};
    int i;

    for(i = 0 ; i < 3 ; i++)
        fp[i]();
}
```

Explanation

As we have array of integers, array of integer pointer, we can have array of function pointers as well. An array of function pointers is nothing but a collection of similar type of functions (same return type and same type of argument list)

7.20 Nested function calls

/* nested function calls */

Program 7.45

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

void main()
{
    int a, b;

    printf("Enter two numbers :");
    scanf("%d %d",&a, &b);

    printf("Sum = %d", sum(a,b) );
}
```

Explanation

- a) A function can be nested, provided it's return type is other than ***void*** type.
- b) The inner functions get executed first in the direction of left to right.

7.21 Array of character pointers

/* array of character pointers */

Program 7.46

```
#include<stdio.h>

char* getString()
{
    char st1[100], *p;

    printf("Enter a string:");
    fflush(stdin);

    gets(st1);
}
```

```
        p = (char*)malloc( strlen(st1) + 1 );

        strcpy(p, st1);
        return p;
}

void main()
{
    char* st[10];
    int n, i;

    printf("How many strings do u want to enter:");
    scanf("%d", &n);

    for(i = 0 ; i < n ; i++)
        st[i] = getString();

    printf("The strings are\n");

    for(i = 0 ; i < n ; i++)
        printf("%s\n", st[i]);
}
```

Output

```
How many strings do u want to enter: 5
Enter a string : Hyderabad
Enter a string : Visakhapatnam
Enter a string : Vijayawada
Enter a string : Tirupathi
Enter a string : Warangal
```

```
The strings are
Hyderabad
Visakhapatnam
Vijayawada
Tirupathi
Warangal
```

Explanation

Even though **100 bytes** gets allocated for **st1**, we know that **st1** dies as soon as the control returns from the **getString()** function. However the dynamically allocated memory for each of the strings persists as long as the program is under execution. So we were able to allocate only required amount of memory for each of the strings.

7.22 scanf() revisited

Very often we have been using the `scanf()` function. Let's discuss it in detail...

```
/* scanf() revisited */
```

Program 7.47

```
void main()
{
    int n;
    char st[20];

    printf("Enter a number:");
    scanf("%d", &n); /* call by address */

    printf("Enter a string:");
    scanf("%s", st); /* call by address */

    printf("%d\n%s", n, st);
}
```

Explanation

a) While reading integers, why ampersand(&) is required in the **scanf()** function? We know that to change the actual argument's value through the called function (here **scanf()**), we have to pass the argument by address.

b) While reading strings, why ampersand(&) is not required in the **scanf()** function? We know that name of the array (**st**) is nothing but an address (base address), so inherently it is call by address. Hence ampersand (&) is not required for reading the strings.

7.23 Changing array address

/ changing array address */*

Program 7.48

```
void main()
{
    int a[] = {13, 9, 64, 53, 18}, i;
    int* p;

    p = a;
    for(i=0;i<5;i++)
    {
        printf("%d\t", *p);
        p++;
    }

    printf("\n");
    for(i=0;i<5;i++)
    {
        printf("%d\t", *a);
        a++;
    }
}
```



error

Explanation

Array is a **constant pointer**. We know that name of an array contains its base address. This value cannot be changed.

Use of pointers

Some of the simple applications of pointers are listed below:

- Dynamic memory management is possible
- Passing arrays to functions
- Call by reference(to change the actual argument's value or to avoid a copy of the actual argument)
- Pointer notation is faster than array notation.
- To return more than one value from a function.

Note

Pointers are a fundamental part of 'C'. If you cannot use pointers properly then you have basically lost all the power and flexibility that 'C' allows. The secret to 'C' is in its use of pointers.

'C' is unusual in that it allows pointer to point to anything. Pointers are sharp tools, and like any such tool, used well they can be delightfully productive but used badly they can do great damage.

Lot's of new programmers think they hate pointers. Don't hate pointers. **Love pointers.** Pointers are the difference between having someone's address written down in a notebook and carrying their house around with you wherever you go.

Exercise

State True or False

- a) Pointer is a variable, which holds an address.
- b) %a is used to print an address.
- c) a[i] is equivalent to (a+i)
- d) Subtraction between two pointers is possible.
- e) void pointers are possible.

Structures

Topic # 8

Atish Jain

8.1 When to use a structure?

We know that array is a collection of elements of similar type. However, many a times we need to group multiple types of data into a single entity. To meet this type of requirement we use structures.

Usually real-world data consists of multiple types of characteristics:

Ex1 student details : rno, age, name, course, fee etc.

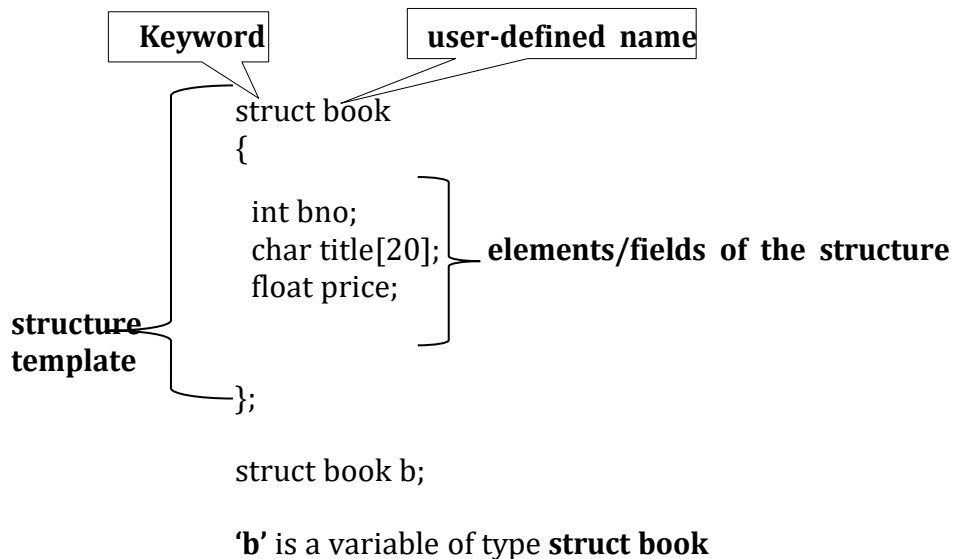
Ex2 employee details : eno, age, name, designation, salary etc.

Ex3 book details : bno, title, author, price etc.

Structure A structure is a collection of variables under a single name. These variables can be of different types (*int / char / float*).

How to create a structure

Creating a structure is nothing but creating our own datatype.



Number of bytes allocated for a structure variable is the total amount of memory required for all its elements. So here **'b'** occupies 26 (2 + 20 + 4) bytes.

Note that memory is allocated for a structure only after defining a variable of its type, but not as soon a structure template is created.

8.2 Reading & printing

Program 8.1

```
/* reading and printing */

struct book
{
    int bno;
    char title[20];
    float price;
};

void main()
{
    struct book b;

    printf("Enter book details:");
    scanf("%d%s%f", &b.bno, b.title, &b.price);

    printf("%d, %s, %f", b.bno, b.title, b.price);
}
```

Explanation

- a) Structure elements are accessed using the dot (.) operator.
- b) In memory the book structure is organized as follows:

134	How to program C	545.00
bno	title	price
100	102	122

b
100

8.3 Array of structures?

Program 8.2

```
/* array of structures */

struct book
{
    int bno;
    char title[20];
    float price;
};

void main()
{
    struct book b[5];
    int n, i;
    printf("How many books details do u want to enter:");
    scanf("%d", &n);

    for(i = 0 ; i < n ; i++)
    {
        printf("Enter book details:");
        scanf("%d%s%f", &b[i].bno, b[i].title, &b[i].price);
    }

    printf("The books details are\n");
    for(i = 0 ; i < n ; i++)
        printf("%d\t%s\t%f\n", b[i].bno, b[i].title, b[i].price);
}

void bradman(float* p)
{
    float q = 1.5;
    bradman( &q );
}
```

Explanation

Purpose of the dummy function **bradman()**:

It is called as a link function, which is used to link the **Floating- point Emulator** software.

Why should we link it?

Today's computer has a math-coprocessor which performs the floating-point operations. But 20 years ago there was no math-coprocessor, all the floating-point operations were carried by a program called as **floating-point emulator**. As our Compiler is DOS-based (developed 25 years ago), it too uses the **floating-point emulator** program for floating-point calculations. So this program along with our program gets loaded into memory. But for efficiency, the creators of Turbo C Compiler, created it in such a way that the **floating-point emulator** program gets loaded only when required. However there is a bug in the Compiler, because of which sometimes even though necessary, it doesn't get loaded into memory.

The Bug is: reading float member of a structure, using array of structures or pointer to a structure.

Had we not used this dummy function, we would have got the runtime error:

scanf : floating point formats not linked
abnormal program termination.

8.4 Structures and pointers

Program 8.3

```
/* structures and pointers */
```

```
struct student
```

```
{
```

```
    int sno, age;
```

```
    char sna[20], course[20];
```

```
    float fee;
```

```
};
```

```
void main()
```

```
{
```

```
    struct student s, *p;
```

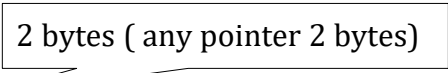
```
    p = &s;
```

```
    printf("Enter student's details:");
```

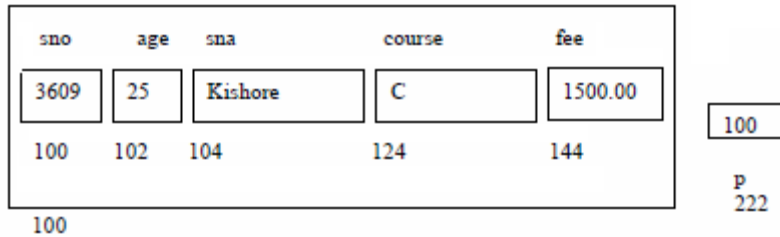
```
    scanf("%d%d%s%s%f", &s.sno, &s.age, s.sna, s.course, &s.fee);
```

```
    printf("%d, %d, %s, %s, %f", p->sno, p->age, p->sna, p->course, p->fee);
```

```
}
```



2 bytes (any pointer 2 bytes)

Explanation**48 bytes**

Arrow operator (->) is used to access a field through a pointer to a structure.

p->sno means : student number at 100

8.5 Dynamic memory allocation / de-allocation**Program 8.4**

```
/* dynamic memory management - example 1 */
/* allocating memory to one structure */

struct student
{
    int sno, age;
    char sna[20], course[20];
    float fee;
};

void main()
{
    struct student *p;
    p = (struct student*)malloc( sizeof(struct student) );
    printf("Enter student's details:");
    scanf("%d%d%s%s%f", &p->sno, &p->age, p->sna, p->course, &p->fee);
    printf("%d, %d, %s, %s, %f", p->sno, p->age, p->sna, p->course, p->fee);
}

void bradman(float* p)
{
    float q = 8.2;
    bradman(&q);
}
```

Explanation

sno	age	sna	course	fee
3609	25	Kishore	C	1500.00
100	102	104	124	144

100

100

P
222

&p->fee means : passing the address of fee at 100 (i.e., 144)

```
/* dynamic memory management - example 2 */  
/* allocating memory to a set of structures */  
100
```

```
p  
222
```

Program 8.5

```
struct student  
{  
    int sno, age;  
    char sna[20], course[20];  
    float fee;  
};  
  
void main()  
{  
    struct student *p;  
    int n, i;  
    printf("How many students details do u want to enter:");  
    scanf("%d", &n);  
  
    p = (struct student*)malloc( n * sizeof(struct student) );  
  
    for(i = 0 ; i < n ; i++)  
    {  
        printf("Enter student's details:");  
        scanf("%d%d%s%s%f",&(p+i)->sno, &(p+i)->age, &(p+i)->sna,  
            &(p+i)->course, &(p+i)->fee);  
    }  
}
```

```
printf("The students details are\n");

for(i = 0 ; i < n ; i++)
    printf("%d\t%d\t%s\t%s\t%f\n", (p+i)->sno, (p+i)->age, (p+i)->sna,
    (p+i)->course, (p+i)->fee);
}

void bradman(float* p)
{
    float q = 8.2;

    bradman(&q);
}
```

8.6 Initialization and assignment

Program 8.6

```
/* initialization and assignment */

struct employee
{
    int eno, age;
    char ena[30], desig[30];
    float sal;
};

void main()
{
    struct employee e1 = { 11, 27, "Sambhu Singh", "Commissioner of Police",
    35000.00 }, e2;

    e2 = e1;

    printf("%d, %d, %s, %s, %f\n", e1.eno, e1.age, e1.ena, e1.desig, e1.sal);
    printf("%d, %d, %s, %s, %f\n", e2.eno, e2.age, e2.ena, e2.desig, e2.sal);
}
```

Explanation

Note that structure elements (fields) cannot be initialized inside the structure template, because a structure template is only a general form, memory is not allocated for structue template.

8.7 Passing structures to functions

Program 8.7

```
/* passing structures to functions */

struct employee
{
    int eno, age;
    char ena[30], desig[30];
    float sal;
};

void swap(struct employee* p, struct employee* q)
{
    struct employee t;

    t = *p;
    *p = *q;
    *q = t;
}

void main()
{
    struct employee e1 = { 11, 27, "Kishore", "Marketing Manager", 11000.00 };
    struct employee e2 = { 12, 27, "Vinodh", "Deputy Centre Manager", 13000};

    swap(&e1, &e2);

    printf("%d, %d, %s, %s, %f\n", e1.eno, e1.age, e1.ena, e1.desig, e1.sal);
    printf("%d, %d, %s, %s, %f\n", e2.eno, e2.age, e2.ena, e2.desig, e2.sal);
}
```

8.8 typedef

typedef is a keyword which is used to define an alias name for a datatype(usually structure type of data type).

We know that alias names are easier to refer, than the original names.

Eg : Original name : Konedela Shiva Sankar Vara Prasad
 Alias name : Chiranjeevi

Program 8.8

```
/* use of typedef */
struct employee
{
    int eno, age;
    char ena[30], desig[30];
    float sal;
};

typedef struct employee employee;

void swap(employee* p, employee* q)
{
    employee t;
    t = *p;
    *p = *q;
    *q = t;
}

void main()
{
    employee e1 = { 11, 27, "Kishore", "Marketing Manager", 11000.00 };
    employee e2 = { 12, 27, "Vinodh", "Deputy Centre Manager", 13000};

    swap(&e1, &e2);

    printf("%d, %d, %s, %s, %f\n", e1.eno, e1.age, e1.ena, e1.desig, e1.sal);
    printf("%d, %d, %s, %s, %f\n", e2.eno, e2.age, e2.ena, e2.desig, e2.sal);
}
```

8.9 Nested structures

Structure with a structure (group of multiple types, within a group of multiple types) is called as nested structure.

Program 8.9

```
/* nested structures */

struct address
{
    int dno;
    char street[20], area[20], city[20];
};
```

```
struct student
{
    int sno;
    char sna[20], course[20];
    float fee;

    struct address a;
};

void main()
{
    struct student s;

    printf("Enter student details:");

    scanf("%d%s%s%f%d%s%s%s", &s.sno, s.sna, s.course, &s.fee, &s.a.dno,
s.a.street, s.a.area, s.a.city);

    printf("%d, %s, %s, %f, %d, %s, %s, %s", s.sno, s.sna, s.course, s.fee, s.a.dno,
s.a.street, s.a.area, s.a.city);
}
```

8.10 Date and time structures


Program 8.10

```
/* date and time structures */

#include<dos.h>

void main()
{
    struct date d;

    struct time t;

    getdate(&d);
    gettime(&t); 
    printf("Current Date : %02d - %02d - %d\n", d.da_day, d.da_mon, d.da_year);
    printf("Current Time : %02d : %02d : %02d", t.ti_hour, t.ti_min, t.ti_sec);
}
```


Explanation

date and time are pre-defined structures (defined in dos.h)

8.11 unions

unions are similar to structures, except the fact that memory allocated for a **union** variable is memory required for the largest field.

Program 8.11

```
/* unions - example 1 */

union sample
{
    int n;
    char st[20];
    float c;
    double d;
};

void main()
{
    union sample s;

    printf("%d", sizeof(s) );
}
```

Output

20

Program 8.12

```
/* unions - example 2 */

#include<stdio.h>

union sample
{
    double a;
    float b[2];
    char c[8];
};
```

```
void main()
{
    union sample s;
    int i;

    s.a = 1345.672;
    printf("%lf\n", s.a);

    s.b[0] = 1.43;
    s.b[1] = 82.6;
    printf("%f, %f\n", s.b[0], s.b[1]);

    printf("Enter 8 charactes (string):");
    for(i = 0 ; i < 8 ; i++)
        scanf("%c", &s.c[i]);

    for(i = 0 ; i < 8 ; i++)
        printf("%c", s.c[i]);
}
```

Explanation

At a time only one field's value can be stored. Note that the memory allocated for a **union** can be accessed in several formats, i.e., as **ints** / **chars** / **floats** / **double** etc.

8.12 enum

There is one other kind of constant, the **enumeration constant**. An enumeration is a list of constant integer values.

Ex 1 enum boolean { NO, YES};

0 1

Ex 2 enum condition {FALSE, TRUE};

0 1

Ex 3 enum months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC};

1 2 3 4 5 6 7 8 9 10 11 12

Ex 4 enum liquids {WATER = 5, OIL = 7, MILK};

5 7 8 1

The first name is an enum, has value 0, the next 1, and so on, unless explicit values are specified. If not all values are specified, unspecified values continue the progression from the last specified value.

Remember that the constant names in different enumerations must be distinct, values need not be distinct in the same enumeration.

Program 8.13

```
/* enum - example 1 */  
  
enum week{ SUN, MON, TUE , WED, THU, FRI, SAT };  
  
void main()  
{  
    printf("%d", THU);  
}
```

Output

4

<i>enum</i>	Vs	macros (#define)
1. values are generated automatically		1. not generated
2. more than one constant can be defined at a time		2. only one

8.13 Dice game

Rule : A player rolls two dice. Each dice has 6 faces. These faces contains : 1, 2, 3, 4, 5, 6 spots. After the dice have come to rest, the sum of the spots on the two upward faces is calculated.

If the sum is 7 or 11 on the first throw, the player wins. If the sum is 2, 3 or 12 on the first throw, the player loses. If the sum is 4,5, 6, 8, 9 or 10 on the first throw, then the sum becomes the player's point. To win, the player must continue rolling the dice until the player makes his point.

The player loses by rolling 7 before making the point.

Program 8.14

```
/* enum - example 2 */

/* also usage of random number generators, fall through in switch - case */

#include<stdlib.h>

enum gamestatus{ WON, LOST, CON };

int rollDice()
{
    int d1, d2, d3;

    printf("Press any key to roll the dice...\n");
    getch();

    d1 = 1 + random( 6 );
    d2 = 1 + random( 6 );

    d3 = d1 + d2;

    printf("Player rolled : %d\n", d3);

    return d3;
}

void main()
{
    int s, gs, mypoint;

    randomize();

    s = rollDice();

    switch( s )
    {
        case 7 :
        case 11:
            gs = WON;
            break;
        case 2:
```

```
        case 3:
        case 12:
            gs = LOST;
            break;
        default :
            gs = CON;
            mypoint = s;
            printf("Player's point is %d\n", mypoint);
    }

    while( gs == CON )
    {
        s = rollDice();

        if(s == mypoint)
            gs = WON;
        else if( s == 7)
            gs = LOST;
    }

    if(gs == WON)

        printf("Player won");

    else

        printf("Player lost");
}
```

Exercise

State True or False

- a) Structure is a collection of multiple type of variables.
- b) dot operator is used to access a field through a structure object.
- c) Arrow operator is used to access a field through a pointer to a structure object.
- d) Size of an union is the size of the largest element.

Files (Disk I/O)

Topic # 9

Atish Jain

Round-1

9.1 Console I/O and Disk I/O

Till now we have been doing console I/O. i.e., reading the data from the console input device (keyboard) and sending the output to the console output device (screen).

Reading	:	keyboard	→	program
Writing	:	program	→	screen

Disk I/O

Reading the data from a disk file(usually Hard disk) and sending the output to a disk file (usually Hard disk).

Disk I/O means storing the data on a secondary storage device using files.

Disk I/O is used for developing database applications.

File is a collection of bytes stored on a secondary storage device.

9.2 Use of Disk I/O

Use

There are several uses of Disk I/O:

- a) Suppose that today you have written a program, which reads a hundred names, sorts them into alphabetical order and prints them. You would undergo the following steps:
- step1 : Write the program
 - step2 : Compile it
 - step3 : Execute it
 - step4 : Enter all the hundred names
 - step5 : Data is processed
 - step6 : See the output

Suppose that after few days you want to view the same output again. What would you do? Execute that program once again and enter the hundred names all over again. Instead won't it be nice if we could store the result in a separate disk file and whenever required just open that file and see the result. To accomplish this task we have to perform **disk output** operations.

b) Suppose that you want to find the maximum of a set of numbers. What would you do?

```

step1 : Write the program
step2 : Compile it
step3 : Execute it
step4 : Enter a set of numbers (say, hundred)
step5 : Data is processed
step6 : See the output

```

Suppose that after few days you want to perform a different operation (say, sum of numbers) on the same data. What would you do?

```
step1 : Write a new program
step2 : Execute it
step3 : Enter the same data again
```

Instead won't it be nice if you could read the data (hundred numbers) from a disk file. To accomplish this task we have to perform **disk input** operations.

c) For any **Database applications** we have to use Disk I/O.

Reading : Disk----->----- Program----->-----
Screen/Disk
Writing : Disk/keyboard->---- Program-->----- Disk

File operations:

- Creating a new file and inserting the data.
- Opening the existing file for append/update/delete.

There are 2 types of files

1. Text files

- Text files stores character data
- Text files are readable
- Text files are processed sequentially (character by character)

2. Binary files

- Binary files stores data in binary format
- Binary files are non readable
- Random access

9.3 Writing data to a disk file

/* writing data to a disk file */

/* Saving the output of a program permanently in a disk file */

Program 9.1

Step1: Create the following program:

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

```
void main()
{
    char st[10][30], temp[30];
    int n, i, j;
    FILE *fp;

    fp = fopen("sort2.txt", "w");

    printf("How many strings:");
    scanf("%d", &n);

    printf("Enter the strings:");
    fflush(stdin);
    for(i=0;i<n;i++)
        gets(st[i]);

    for(i=0;i<n-1;i++)
    {
        for(j=0;j<n-1-i;j++)
        {
            if( strcmp(st[j], st[j+1]) > 0 )
            {
                strcpy(temp, st[j]);
                strcpy(st[j], st[j+1]);
                strcpy(st[j+1], temp);
            }
        }
    }

    fprintf(fp, "In alphabetical order:\n");
    for(i=0;i<n;i++)
        fprintf(fp, "%s\n", st[i]);

    fclose(fp);
}
```

Step2: open notepad and view the file sort2.txt

Explanation

a) After executing the program, goto command prompt and view the contents of file1.txt.

b) FILE *fp;
Creates a FILE pointer '**fp**'.
FILE is a special structure (pre-defined). While performing disk I/O this FILE structure should be used.

c) **fopen()**
Before reading/writing into a file, first that file should be opened. This task is done by the **fopen()** library function.
fopen() takes two arguments - file to open and file mode.

d) Lets discuss **fopen()** in detail:

If read mode

step 1 : searches for the file on the disk
step 2 : if absent return NULL to 'fp', if present, reserves a buffer area for transfer of data.
step 3 : Loads the file contents into the buffer area.
step 4 : Sets the FILE structure attributes with necessary values
step 5 : returns the starting address of the FILE structure to 'fp'. Now 'fp' is associated with the specified file.

If write mode

step 1 : searches for the file on the disk.
step 2 : if absent, creates a new one with the specified name
step 3 : reserves a buffer area for data transfer
step 4 : sets the FILE structure elements with necessary values
step 5 : returns the starting address of the FILE structure to 'fp'. Now 'fp' is associated with the file.

e) **fclose()**

Every file opened should be closed. This job is done by the **fclose()** function. Even though we don't close the file, it gets closed at the end of the program, However, during the execution of the program, if it needs to be opened again, it has to be closed.(just like, a door can be opened only if it is in closed state)

Note: If buffer area is not created for I/O operations, then the program execution becomes very slow (at the speed of disk rather than at the speed of processor)

9.4 Reading data from a disk file

/ taking input from a disk file*/*

Program 9.2

Step1: open notepad and create a file input1.txt as follows:

```
8
56
23
831
82
2
568
75
238
```

Step2: now create the following program:

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

```
void main()
```

```
{
    int a[10], n, i, j, t;
    FILE *fp;

    fp = fopen("input1.txt", "r");

    fscanf(fp, "%d", &n);

    for(i=0;i<n;i++)
        fscanf(fp, "%d", &a[i]);

    for(i=0;i<n-1;i++)
    {
        for(j=0;j<n-1-i;j++)
        {
            if( a[j] > a[j+1])
            {
                t = a[j];
                a[j] = a[j+1];
                a[j+1] = t;
            }
        }
    }
}
```

```
    printf("In ascending order:\n");

    for(i=0;i<n;i++)
        printf("%d\n", a[i]);

    fclose(fp);

    getch();
}
```

/*writing the content that we enter on the console into a file */

Program 9.3

```
#include<stdio.h>

void main()
{
    FILE *fp;
    char ch;

    fp = fopen("data1.txt", "w");

    while( ( ch = getchar() ) != EOF )
        fputc(ch, fp);

    fclose(fp);
}
```

Explanation

- a) **fputc()** writes data character by character.
- b) **EOF** is a macro whose value is -1. This program keeps on reading until the user presses the end of input key (CTRL + Z). So when this key is encountered, **getchar()** function returns -1.

/* reading data from a disk file - character by character*/

Program 9.4

```
#include<stdio.h>

void main()
{
    FILE *fp;
    char ch;
    fp = fopen("file1.txt", "r");
    while( 1 )
    {
        ch = fgetc(fp);

        if(ch == EOF)
            break;
        printf("%c", ch);
    }

    fclose(fp);
}
```

Explanation

- a) Every text mode file has a special character at its end called as end of the file character (ASCII VALUE 26).
- b) **fputc()** returns -1 when end of the file character is encountered.

Round-2

9.5 Writing and reading infinite number of records

/* writing infinite number of records to a disk file */

Program 9.5

```
#include<stdio.h>

struct student
{
    int rno, age;
    char sna[20], course[20];
    float fee;
};
```

```
void main()
{
    FILE *fp;
    char ch;
    struct student s;

    fp = fopen("stud1.txt", "w");

    while( 1 )
    {
        printf("\nEnter student details : ");
        scanf("%d%d%s%s%f", &s.rno, &s.age, s.sna, s.course, &s.fee);

        fprintf(fp, "%d\t%d\t%s\t%s\t%f\n", s.rno, s.age, s.sna, s.course, s.fee);

        fflush(stdin);
        printf("Another record?[y/n]:");
        ch = getche();

        if(ch != 'y')
            break;
    }

    fclose(fp);
}
```

Explanation

fprintf() is used to write record by record, where as **fputc()** is used to write character by character.

/* reading records from a disk file */

Program 9.6

```
#include<stdio.h>

struct student
{
    int rno, age;
    char sna[20], course[20];
    float fee;
};
```

```
void main()
{
    FILE *fp;
    struct student s;

    fp = fopen("stud1.txt", "r");

    while( fscanf(fp, "%d%d%s%s%f", &s.rno, &s.age, s.sna, s.course, &s.fee ) != EOF
)
        printf("%d\t%d\t%s\t%s\t%f\n", s.rno, s.age, s.sna, s.course, s.fee);

    fclose(fp);
}
```

Output

```
111  22   Ravi  C    1500
112  21   Shiva C++  1500
113  22   Jagan VC++ 3000
114  18   Sekhar Java 3000
```

Explanation

fscanf() returns -1 when end of the file character is encountered.

9.6 Filtering the records

```
/* filtering the records */
/* displaying the records of the students of a particular course */
```

Program 9.7

```
#include<stdio.h>

struct student
{
    int rno, age;
    char sna[20], course[20];
    float fee;
};
```



```
void main()
{
    FILE *fp;
    char ch, st[20];
    struct student s;

    fp = fopen("stud1.txt", "r");

    printf("Enter course name:");
    scanf("%s", st);

    while( fscanf(fp, "%d%d%d%s%s%f", &s.rno, &s.age, s.sna, s.course, &s.fee ) != EOF
)
    {
        if(strcmp(st, s.course) == 0 )

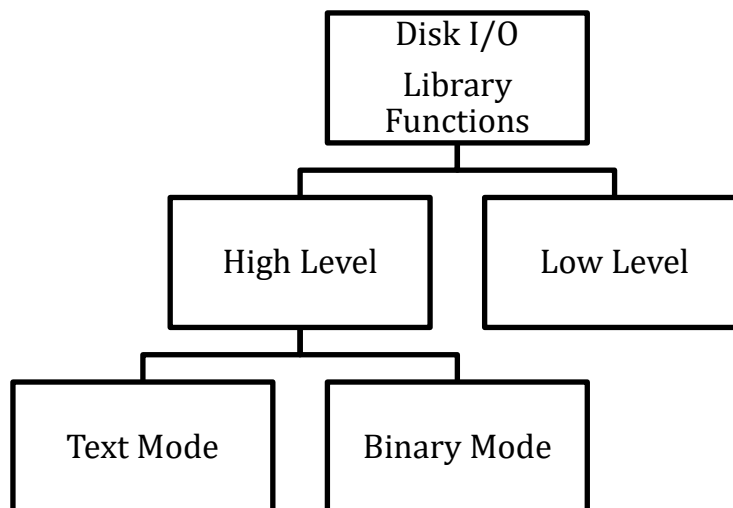
            printf("%d\t%d\t%s\t%s\t%f\n", s.rno, s.age, s.sna, s.course,
s.fee);
    }

    fclose(fp);
}
```

Round-3

9.7 Binary mode file handling

Disk I/O library functions can be classified into two types:



Till now we discussed high-level text mode file handling. Lets now discuss high-level binary mode file handling, later on we will disk low-level file handling in the next chapter.

The binary mode library functions, handle the data in binary form. This means that the data is stored in the disk in the same format (same number of bytes) as is stored in the primary memory (RAM).

Advantages

- a) There are no conversions while saving/ retriving the data and therefore file handling is much faster.
- b) Remains encrypted (unreadable by humans) – One cannot understand the contents through the *type* command.

Note that in **text** mode file handling, an *int* occupies 2 bytes whereas in the disk it depends upon the number of digits a particular *int* value has. Similar is the case with *float* values.

There are three main areas where text and binary mode functions are different:

- a) Handling of new line character (text mode-two bytes)
- b) Representation of end of file (no end of file character for binary mode)
- c) Storage of *int* / *floats*

/* binary mode - writing records to a disk file */

Program 9.8

```
#include<stdio.h>

struct student
{
    int rno, age;
    char sna[20], course[20];
    float fee;
};

void main()
{
    FILE *fp;
    char ch;
    struct student s;

    fp = fopen("stud2.bin", "wb");
```

```
while( 1 )
{
    printf("\nEnter student details : ");
    scanf("%d%d%s%s%f", &s.rno, &s.age, s.sna, s.course, &s.fee);

    fwrite(&s, sizeof( s ), 1, fp);

    fflush(stdin);
    printf("Another record?[y/n]:");
    ch = getche();

    if(ch != 'y')
        break;
}

fclose(fp);
}
```

Explanation

fwrite(&s, sizeof(s), 1, fp);

&s : Base address to transfer bytes from
sizeof(s) : Number of bytes to be transfered
1 : For arrays, number of elements to transfer. In this case since no arrays, so only one.
fp : Write to the file associated with 'fp'

/* binary mode - reading records from a disk file */

Program 9.9

```
#include<stdio.h>

struct student
{
    int rno, age;
    char sna[20], course[20];
    float fee;
};
```

```
void main()
{
    FILE *fp;
    struct student s;

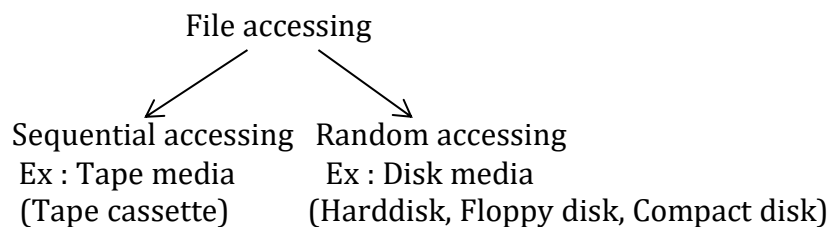
    fp = fopen("stud2.bin", "rb");

    while( fread( &s, sizeof( s ), 1, fp) )
        printf("%d\t%d\t%s\t%s\t%f\n", s.rno, s.age, s.sna, s.course, s.fee);

    fclose(fp);
}
```

Round-4

9.8 Random file accessing



Till now we have been doing sequential file accessing. i.e., to access a record (or a particular byte) we have to go through all its previous records (or bytes). However in random file accessing to access a record (or a particular byte) we can directly reach that point. Hence, random file accessing is much faster than sequential file accessing. Lets see how this can be accomplished.

/* random file accessing – finding number of records */

Program 9.10

```
#include<stdio.h>

typedef struct student
{
    int rno;
    char sna[20], course[20];
    float fee;
}student;
```

```
void main()
{
    student s;
    FILE *fp;
    int bytenumber;

    fp = fopen("student1.obj", "rb");

    fseek(fp, 0 , SEEK_END);

    bytenumber = ftell(fp);

    printf("Number of records = %d", bytenumber/sizeof(s) );

    fclose(fp);

    getch();
}

/* random file accessing - getting nth record */
```

Program 9.11

```
#include<stdio.h>

struct student
{
    int rno, age;
    char sna[20], course[20];
    float fee;
};

void main()

{
    FILE *fp;
    struct student s;
    int recordno, position;

    fp = fopen("stud2.bin", "rb");

    printf("Enter the record number:");
    scanf("%d", &recordno);
```

```
    position = (recordno - 1) * sizeof( s ) ;

    fseek(fp, position , SEEK_SET);

    fread( &s, sizeof( s ), 1, fp);

    printf("%d\t%d\t%s\t%s\t%f\n", s.rno, s.age, s.sna, s.course, s.fee);

    fclose(fp);
}
```

Explanation

- a) **fseek()** happens to be one more library function. It moves the file pointer by a quantity of specified number of bytes.
- b) **SEEK_SET** is a macro. It indicates, to move the file pointer from the beginning of the file.
There are two more macros:
SEEK_CUR : It indicates, to move the file pointer from the current location of the file.
SEEK_END : It indicates, to move the file pointer from the end of the file.
- c) Suppose that, we want to view record number five. So we have to move to the starting byte number of record number five. Using **fseek()** we move to the end of fourth record Hence the code : **position = (recordno - 1) * sizeof(s) . fseek()** automatically positions the cursor at the specified byte (+ 1) location . i.e., the starting address of record number 5.
- d) We know that **fread()** function extracts the data from the current file pointer position (i.e., record number five)
- e) **ftell()** is another library function. It returns the byte position of the pointer.

Round-5

9.9 Deleting a record

Actually we cannot delete anything form the memory (RAM/ DISK). It is only the technique of memory management, i.e., reserving/ unreserving the memory.

/* deleting a record */**Program 9.12**

```
#include<stdio.h>

struct student
{
    int rno, age;
    char sna[20], course[20];
    float fee;
};

void main()
{
    FILE *fp1, *fp2;
    struct student s;
    int n;

    fp1 = fopen("school1.bin", "rb");
    fp2 = fopen("temp.bin", "wb");

    printf("Enter roll number:");
    scanf("%d", &n);

    while( fread(&s, sizeof( s ), 1, fp1) )
    {
        if(n != s.rno)
            fwrite( &s, sizeof( s ), 1, fp2 );
    }

    fclose(fp1);
    fclose(fp2);

    remove("school1.bin");
    rename("temp.bin", "school1.bin");
}
```

Explanation

a) Before removing/ renaming a file, we have to disassociate the link between the file pointer and the file being accessed. Hence we have closed the files.

9.10 Modifying a record

/* modifying a record */

Program 9.13

```
#include<stdio.h>

struct student
{
    int rno, age;
    char sna[20], course[20];
    float fee;
};

void main()
{
    FILE *fp1, *fp2;
    struct student s;
    int n;

    fp1 = fopen("school1.bin", "rb");
    fp2 = fopen("temp.bin", "wb");

    printf("Enter roll number:");
    scanf("%d", &n);

    while( fread(&s, sizeof( s ), 1, fp1) )
    {
        if(n == s.rno)
        {
            printf("Enter roll no, age, name, course and fee:");
            scanf("%d%d%s%s%f", &s.rno, &s.age, s.sna, s.course, &s.fee);
        }

        fwrite( &s, sizeof( s ), 1, fp2 );
    }

    fclose(fp1);
    fclose(fp2);

    remove("school1.bin");
    rename("temp.bin", "school1.bin");
}
```


Round-6**9.11 Putting it all together**

```
/* putting it all together */  
/* a menu driven student database management program */
```

Program 9.14**Step 1 : Create a file struct.h**

```
struct student  
{  
    int rno, age;  
    char sna[20], course[20];  
    float fee;  
};
```

```
typedef struct student student;
```

Step 2 : Create a file append.c

```
void append()  
{  
    FILE *fp;  
    student s;  
    char ch;           append mode  
  
    fp = fopen("studdata.bin", "ab");  
  
    while( 1 )  
    {  
        printf("Enter student details:");  
        scanf("%d%d%s%s%f", &s.rno, &s.age, s.sna, s.course, &s.fee);  
  
        fwrite( &s, sizeof( s ), 1, fp);  
  
        fflush(stdin);  
        printf("Another record?[y/n]:");  
        ch = getche();  
  
        if(ch != 'y')  
            break;  
    }  
  
    fclose( fp );  
}
```

Step 3 : Create a file display.c

```
void display()
{
    FILE *fp;
    student s;

    fp = fopen("studdata.bin", "rb");

    printf("\n");

    while( fread( &s, sizeof( s ), 1, fp) )
        printf("%d\t%d\t%s\t%s\t%f\n", s.rno, s.age, s.sna, s.course, s.fee);

    fclose(fp);
}
```

Step 4 : Create a file filter.c

```
void filter()
{
    FILE *fp;
    student s;
    char st[20];

    fp = fopen("studdata.bin", "rb");

    printf("Enter course name:");
    scanf("%s", st);

    printf("\n");

    while( fread( &s, sizeof( s ), 1, fp) )
    {
        if(strcmp(st, s.course) == 0)
            printf("%d\t%d\t%s\t%s\t%f\n", s.rno, s.age, s.sna, s.course,
s.fee);
    }

    fclose(fp);
}
```

Step 5 : Create a file delete.c

```
void delete()
{
    FILE *fp1, *fp2;

    student s;
    int n;

    fp1 = fopen("studdata.bin", "rb");
    fp2 = fopen("temp.bin", "wb");

    printf("Enter roll number:");
    scanf("%d", &n);

    while( fread(&s, sizeof( s ), 1, fp1) )
    {
        if(n != s.rno)
            fwrite( &s, sizeof( s ), 1, fp2 );
    }

    fclose(fp1);
    fclose(fp2);

    remove("studdata.bin");
    rename("temp.bin", "studdata.bin");
}
```

Step 6 : Create a file modify.c

```
void modify()
{
    FILE *fp1, *fp2;

    student s;
    int n;

    fp1 = fopen("studdata.bin", "rb");
    fp2 = fopen("temp.bin", "wb");

    printf("Enter roll number:");
    scanf("%d", &n);
```

```
while( fread(&s, sizeof( s ), 1, fp1) )
{
    if(n == s.rno)
    {
        printf("Enter roll no, age, name, course and fee:");
        scanf("%d%d%s%s%f", &s.rno, &s.age, s.sna, s.course, &s.fee);
    }

    fwrite( &s, sizeof( s ), 1, fp2 );
}

fclose(fp1);
fclose(fp2);

remove("studdata.bin");
rename("temp.bin", "studdata.bin");
}
```

Step 7 : putting it all together**Create a file menu.c**

```
#include<stdio.h>
#include"struct.h"
#include"append.c"
#include"display.c"
#include"filter.c"
#include"delete.c"
#include"modify.c"
void main()
{
    char ch;

    while( 1 )
    {
        printf("*** Student database management system ***\n\n");
        printf("1. Append\n");
        printf("2. Display\n");
        printf("3. Filter\n");
        printf("4. Delete\n");
        printf("5. Modify\n");
        printf("6. Exit\n\n");
        printf("Enter u'r choice:");
        ch = getche();
    }
}
```

```
        switch( ch )
        {
            case '1' : append();
                        break;
            case '2' : display();
                        break;
            case '3' : filter();
                        break;
            case '4' : delete();
                        break;
            case '5' : modify();
                        break;
            case '6' : exit();
            default : printf("Invalid choice");
        }
    }
}
```

Explanation

Whenever we want to add any data at the end of existing data, we have to open the file in append mode.

File modes**wt/wb**

- a) If file is absent, new file is created with the specified name.
 - b) If file is present, existing file is overwritten.
- Operation: writing

rt/rb

- a) If file is absent, returns NULL
 - b) If file is present, loads the file into memory and sets pointer pointing to first character in the file.
- Operation: reading

at/ab

- a) If file is absent, new file is created.
 - b) If file is present, loads the file into memory and sets pointer beyond last character in file
- Operation : writing at the end

wt+/wb+

- c) If file is absent, new file is created with the specified name.
- d) If file is present, existing file is overwritten.

Operation: reading/ writing

rt+/rb+

- c) If file is absent, returns NULL
- d) If file is present, loads the file into memory and sets pointer pointing to first character in the file.

Operation: reading/writing

at+/ab+

- c) If file is absent, new file is created.
- d) If file is present, loads the file into memory and sets pointer beyond last character in file

Operation : writing at the end/ reading

Exercise

State True or False

- a) `fopen()` is used to open a file.
- b) Binary mode file accessing is faster than text mode file accessing.
- c) Sequential file accessing is faster than random file accessing.
- d) Console means keyboard + screen.
- e) When we delete any data from a disk, only memory gets unreserved.

Command Line Arguments

Topic # 10

Atish Jain

10.1 What are Command line arguments?

As the name indicates, Command line arguments are nothing but the arguments passed to the program (**main()**) from the command prompt.

We know that when we link a program (F9), **.exe** file gets created. This file (program) can also be used as a command, which can be executed at the **DOS** command prompt.

While typing the command, we can also pass some values to our program. These values are received by the **main()**.

Passing arguments to the **main()** from command prompt is known as command line arguments.

The two arguments which are accepted by **main()** are **argc** and **argv**.

argc – argument count, which is **int** type, contains count of the arguments.

argv – argument value, which is array of character pointers, maintains list of arguments.

Lets understand it through programs.


10.2 sum command

Lets create our own **sum** command

```
/* sum command */
```

Program 10.1

save this program as **mysum.c**



```
void main(int argc, char* argv[] )  
{  
    int i, s = 0;  
  
    for(i = 1 ; i < argc ; i++)  
        s += atoi( argv[i] );  
  
    printf("Sum is %d", s);  
}
```

Explanation

- a) Don't run this program by pressing Ctrl + F9. This must be executed at the command prompt.
eg : c:\ahcareer\sum 10 20 30
- b) '**argc**' contains the number of arguments passed from the command prompt (including the program name). It is just for readability that the parameter name is '**argc**', it can be any variable name.
- c) '**argv**' is an array of character pointers which contains the base addresses of each of the arguments.
- d) **atoi()** is a library function which converts a string into an integer.

10.3 type command

Lets create our own **type** command (which displays the contents of a file, on the screen)

```
/* type command */
```

Program 10.2

save this program as mytype.c

```
#include<stdio.h>
void main(int argc, char* argv[] )
{
    FILE *fp;
    char ch;

    fp = fopen( argv[1], "r");
    while( 1 )
    {
        ch = fgetc(fp);
        if(ch == EOF)
            break;

        printf("%c", ch);
    }
    fclose(fp);
}
```

Explanation

How to run this program?

c:\ahccareer\mytype file1.txt

10.4 copy command

Lets create our own copy command (which copies the contents of a file into another file)

/* copy command */

Program 10.3

save this program as mycopy.c

```
#include<stdio.h>
void main(int argc, char* argv[] )
{
    FILE *fp1, *fp2;
    char ch;

    fp1 = fopen( argv[1], "r");
    fp2 = fopen( argv[2], "w");

    while( 1 )
    {
        ch = fgetc(fp1);

        if(ch == EOF)
            break;

        fputc(ch, fp2);
    }

    fclose(fp1);
    fclose(fp2);
}
```

Explanation

How to run this program?

c:\ahccareer\mycopy file1.txt file2.txt

10.5 special copy command

Lets create a special copy command (which copies the contents of a file into multiple files)

```
/* special copy command */  
/* copying into multiple files */
```

Program 10.4

save this program as **mulcopy.c**

```
#include<stdio.h>  
  
void main(int argc, char* argv[] )  
{  
    FILE *fp1, *fp2;  
    char ch;  
    int i;  
  
    fp1 = fopen( argv[1], "r");  
  
    for(i = 2; i < argc ; i++)  
    {  
        fp2 = fopen( argv[i], "w");  
  
        while( 1 )  
        {  
            ch = fgetc(fp1);  
            if(ch == EOF)  
                break;  
            fputc(ch, fp2);  
        }  
        rewind(fp1);  
    }  
  
    fclose(fp1);  
    fclose(fp2);  
  
    printf("File copied into %d files successfully", argc - 2 );  
}
```

Explanation

How to run this program?

c:\ahcareer\mycopy file1.txt f1.txt f2.txt f3.txt f4.txt f5.txt

Purpose of rewind() function

After the completion of copying of **file1.txt** contents into **f1.txt** the **file1.txt** pointer (**fp1**) would be at the end of the file. So we have to reposition the **f1.txt** file pointer (**fp1**) at the beginning of the file so that it can be copied into another file. This job can be accomplished using **rewind()** function which happens to be another library function.

10.6 Copying other than text file / low-level disk I/O

Using High level disk I/O functions we could copy only text files but not the **.exe / .obj / .lib / .com** files. The following program can copy even these type of files.

Low-Level Disk I/O

In Low-level disk I/O data is written/read as **buffer full of bytes**, rather than int / float / char / strings / structures as is done using high level disk I/O functions.

Usually mass data I/O is done using low-level disk I/O.

```
/* low - level disk i/o */  
/* copying .exe / .obj / .com files */
```

Program 10.5

save this program as anycopy.c

```
#include<stdio.h>  
#include<fcntl.h>  
  
void main(int argc, char* argv[] )  
{  
    int in, out, bytes;  
    char buff[512];  
  
    in = open(argv[1], O_RDONLY | O_BINARY);  
    out = open(argv[2], O_CREAT | O_WRONLY | O_BINARY);
```

```
while( 1 )
{
    bytes = read(in, buff, 512);
    if(bytes > 0)
        write(out, buff, bytes);
    else
        break;
}
close(in);
close(out);
}
```

Explanation

How to run this program?

c:\ahcareer\anycopy prog1.exe prog2.exe

- a) **open()** : We know that **fopen()** returns a file structure pointer, but in Low – level disk I/O, **open()** returns an integer value called file handle. This is a number assigned to a particular file, which is used thereafter to refer to the file. If **open()** returns -1 it means that the file couldn't be opened successfully.
- b) **O_RDONLY** , **O_BINARY** , **O_CREAT** , **O_WRONLY** all of them are macros. To use these macros we have to include the file **fcntl.h** (|) is the bitwise **OR** operator which is used to concatenate more than one mode.
- c) **read()** functions reads a maximum of specified number of bytes into the specified char array from the specified file and returns a number which contains the number of bytes actually read.
- d) **write()** function writes specified number of bytes from the specified char array into the specified file.

Exercise**State True or False**

- a) Arguments can be passed to the **main()** function.
- b) The first argument contains the count of arguments.
- c) The second argument contains the values of arguments.
- d) **rewind()** function re-positions the pointer at the beginning of a file.
- e) **write()** function is a low-level disk input function.

OTHERS

Topic # 11

Atish Jain

1. Characteristics of C

(That defines the language & also has lead to its popularity)

- Extensive use of function calls
- Procedural programming language (converting a program into a number of functions)
- Structured programming language (sequencing, selection, iteration – are sufficient to describe the instruction cycle of a CPU)
- Doesn't require extensive runtime support (as soon as we link/build the program we get an executable code which can run independently, whereas Java/VB/VC++ uses a lot of runtime support to make the programs smaller)
- File handling is provided by library functions
- Aggregate (heterogeneous) datatypes are allowed in C – through structures
- A standardized C preprocessor for
 - Macro definition
 - Source code files inclusion etc.
- Loosely/weak typed language (C++/Java Stronger typed language) eg: characters can be used as integers
- Low-level (bit-wise) programming readily available
- Extensive use of pointers for low-level access to memory
- Can develop efficient programs than with High level Languages like Java/ VB
- A large number of compound operators, such as ++, += etc.
- 32 keywords

2. Features of C

- Programming language (syntax) is portable .i.e., independent of CPU architecture
- Source code is portable across all Operating Systems with little or no changes(when we use Compiler specific library)
- Compilers are available on a wide variety of CPUs
- It has high level constructs
- It can handle low-level activities
- It produces efficient programs
- It can be compiled on a variety of computers

- It influenced many languages like C++, C#, Java, Perl, Python, PHP etc
- Its use of modularity. Sections of code can be stored in libraries for re-use in future programs
- First programming language developed by a programmer.

3. Compilation Vs Interpretation

Compilation: conversion of the entire source code into machine code.

Pros

- As the entire machine code which has to be executed is completely available, so faster execution

Cons

- Any changes to the source code requires complete recompilation

Interpretation: conversion of one instruction into machine code and execute it. The process

continues till the end of the program. So execution is a part of interpretation

Pros

- The primary advantage is that we can run the source code program to test its operation, make a few changes & run it again directly. This can enormously speed up the development/testing process.
- Although compiled programs execute faster than interpreted programs, interpreters are popular in program development environment in which programs are changed frequently as new features are added & logical/runtime errors are corrected.
- Interpreters are much easier to build than a compiler.

Cons

- As source code has to be first converted into machine code and then executed, so execution would be slower
- Interpreter s/w has to be loaded into RAM along with the program, so more memory is required
- No copy of the machine code (executable code) exists on the disk.

4. Where is C widely used?

C is widely used in system programming. (i.e., for developing system s/w)

System S/w: is a s/w which directly interacts/controls the computer h/w

Examples

Operating Systems, Compilers/ Interpreters (also called as Language processors), Device Drivers, utility software, Graphics libraries, firmware

Device Driver: is a s/w corresponding to a h/w. When we buy any h/w like printer/scanner and plug it to the Computer, it won't work directly. First the corresponding s/w (device driver) should be loaded. Note that for some h/w, device driver s/w comes along with the Operating System (like Windows XP)

Examples: printer drivers, scanner drivers, keyboard drivers, mouse drivers, pen drive driver, webcam drivers, LAN card drivers, sound drivers etc

Utility Software: also known as service program is a type of computer s/w specifically designed to help manage and tune the computer h/w, OS.

Examples: disk defragmentation, system profiles, virus scanners, compression, network managers etc

Firmware: software stored on integrated circuits eg. ROM

5. Popular s/w developed used C

- Windows Operating System
- UNIX Operating System
- Linux Operating system
- OS/2
- Oracle Database
- DB/2 Database
- MySQL Database
- Dbase Database
- Apache HTTP Server
- PHP libraries
- Sun's Java Virtual Machine
- Ruby's Virtual Machine

Appendix

A. Standard Library Functions

I. GENERAL

- | | | |
|---------------------|---------------------|-----------------------|
| a) printf() | i) putch() | q) randomize() |
| b) scanf() | j) putchar() | r) random() |
| c) clrscr() | k) exit() | s) srand() |
| d) getch() | l) delay() | t) rand() |
| e) getche() | m) sound() | u) toupper() |
| f) getchar() | n) nosound() | v) tolower() |
| g) puts() | o) gettime() | w) sprintf() |
| h) gets() | p) getdate() | x) sscanf() |

II. STRING

- | | | |
|--------------------|---------------------|--------------------|
| a) strlen() | d) strcmp() | h) strset() |
| b) strcpy() | e) strcat() | i) strupr() |
| c) strrev() | f) stricmp() | j) strlwr() |
| | g) strncpy() | |

III. FILES

- | | | |
|-----------------------|--------------------|-------------------|
| a) fprintf() | g) fopen() | m) write() |
| b) fscanf() | h) rewind() | n) read() |
| c) fputc() | i) remove() | o) fseek() |
| d) fgetc() | j) rename() | p) ftell() |
| e) fclose() | k) open() | |
| f) fcloseall() | l) close() | |
| q) | | |

B. IMPORTANT SHORT CUT KEYS (FOR TURBOC2)

- | | | |
|-----|--------------|--------------------------------|
| 1. | F1 | General Help |
| 2. | F2 | Save |
| 3. | F3 | Load / Open |
| 4. | ALT+F3 | Pick |
| 5. | ALT+F9 | Compile |
| 6. | F9 | Link |
| 7. | CTRL+F9 | Execute |
| 8. | ALT+F5 | User Screen / To View A Result |
| 9. | CTRL+F8 | Break Point |
| 10. | F7 | Tracing |
| 11. | F8 | Tracing – Step Over |
| 12. | CTRL+F1 | Topic Help |
| 13. | CTRL + Y | Delete A Line |
| 14. | COPY & PASTE | |
| | | i. Ctrl + k + b |
| | | ii. Ctrl + k + k |
| | | iii. Ctrl + k + c |
| 15. | CUT & PASTE | |
| | | i. Ctrl + k + b |
| | | ii. Ctrl + k + k |
| | | iii. Ctrl+ k + v |

C. BIBLIOGRAPHY

Title	Author
1. Let Us C	Yeshavant Kanetkar
2. Understanding Pointers In C	Yeshavant Kanetkar
3. The C Programming Language	Kernighan & Ritchie
4. Programming with ANSI – C	E. Balagurusamy
5. Programming With C	Venugopal & Prasad

Feed Back Form

Your valuable suggestions are solicited for enabling us to improve the presentational qualities of the book, which incidentally is the Hallmark of “AHCAREER”. Please express your views in brief in the space below and hand it over to our front office

Name : _____

Course Attended _____

Address with Phone No. _____
