

C
in
Depth

2nd
Revised &
updated Edition

S. K. Srivastava
Deepali Srivastava



BPB PUBLICATIONS

Preface to Second Edition

The first edition of this book was appreciated by the students for its simplicity. The second edition maintains this feature along with inclusion of new topics and errors of previous edition being rectified. Every topic has been explained in depth without compromising over the lucidity of the text and programs. This approach makes this book suitable for both novices and advanced programmers.

Development of logic and familiarity with the syntax and features of the language are the two pillars of excellent programming skills. The comprehensive contents of the chapters along with the numerous example programs helps you to develop your logic in a stepwise manner and makes you feel comfortable with the syntax of the language. Remember that you can't learn swimming by just reading a book on how to swim, you have to jump into water for that, similarly if you want to learn programming, it is essential for you to make your own programs. So start by understanding the programs given in the book, work on them, modify them and see the results and try to make similar programs.

Each chapter is complemented by exercises with solutions that act as a review of the chapter. We strongly recommend that you solve all the exercises before switching over to another topic. Exercises have been structured in such a way that you can test and implement the knowledge acquired from the chapter, and this is really important for getting full hold over a topic. We're sure if you understand the concepts, you'll enjoy doing the exercises.

Our aim of writing this book is to enable any student emerge as a full-fledged C programmer who can withstand the challenges of the industry. This is the reason for inclusion of chapters on project building, library development and code optimization.

We are thankful to our family and friends for their love and support.

If you have any problems or suggestions, please feel free to contact us at-

suresh_k_sri@yahoo.co.in

deepali_lko@yahoo.co.in

Suresh Kumar Srivastava

Deepali Srivastava

Preface to First Edition

Hello ! I am Suresh Kumar Srivastava. Firstly I want to tell you how the idea of writing a book on 'C' language came to my mind. When I was in 1st semester of 'B Level', C language was in my course. I didn't know anything about computers. So at that time learning of 'C' language was very difficult for me. I faced a lot of problems. After thorough studies of many standard and authentic books in 'C', it became convenient for me to work in 'C'. Presently I am in a position to say that that I have in-depth knowledge of 'C' and find myself in a position to help my juniors in making them comfortable with 'C'. This idea inspired me to write a book which is easily understandable by beginners and contains all theoretical concepts and their implementation in programming.

I was alone in this work. I was the initiator, visualizer and accomplisher for this work.

I am very thankful to my elder brother Raju Bhaiya and sister Reena didi for their love and care for me and my work. I am thankful to my friend Shailesh Raghuvanshi for proof reading of my book. I am also thankful to Mr. Manish Jain and Mr. Anil Tyagi of BPB Publications for considering my work.

Suresh Kumar Srivastava

Contents

1. Introduction to C	1-6
1.1 Design Methods	1
1.1.1 Top-Down Design	1
1.1.2 Bottom-Up Design	2
1.1.3 Modular Approach	2
1.2 Programming Languages	2
1.2.1 Low Level Languages	2
1.2.1.1 Machine Level Language	2
1.2.1.2 Assembly Language	2
1.2.2 High-Level Languages	3
1.3 Translators	3
1.4 History Of C	3
1.5 Characteristics Of C	4
1.6 Structure Of A C Program	4
1.7 Environment For C	5
1.7.1 Unix Environment	5
1.7.2 MS-DOS Environment	5
1.7.2.1 Command Line	5
1.7.2.2 Integrated Development Environment	6
2. Elements of C	7-16
2.1 C Character Set	7
2.1.1 Alphabets	7
2.1.2 Digits	7
2.1.3 Special characters	7
2.2 Execution Characters/Escape Sequences	8
2.3 Trigraph Characters	8
2.4 Delimiters	9
2.5 Reserved Words / Keywords	9
2.6 Identifiers	9
2.7 Data Types	10
2.8 Constants	10
2.8.1 Numeric Constants	11
2.8.1.1 Integer constant	11
2.8.1.2 Real (floating point) Constants	12
2.8.2 Character Constants	13
2.8.3 String Constants	13

2.8.4	Symbolic Constants	13
2.9	Variables	14
2.9.1	Declaration of Variables	14
2.9.2	Initialisation of Variables	14
2.10	Expressions	15
2.11	Statements	15
2.11.1	Expression Statement	15
2.11.2	Compound Statement	15
2.12	Comments	16
3.	Input-Output In C	17-31
3.1	Conversion Specifications	17
3.2	Reading Input Data	18
3.3	Writing Output Data	21
3.4	Formatted Input And Output	24
3.4.1	Format For Integer Input	24
3.4.2	Format For Integer Output	25
3.4.3	Format For Floating Point Numeric Input	26
3.4.4	Format For Floating Point Numeric Output	27
3.4.5	Format For String Input	27
3.4.6	Format For String Output	28
3.5	Suppression Character in scanf()	28
3.6	Character I/O	29
3.6.1	getchar() and putchar()	29
Exercise		29
Answers		31
4.	Operators And Expressions	32-57
4.1	Arithmetic Operators	32
4.1.1	Unary Arithmetic Operators	32
4.1.2	Binary Arithmetic Operators	32
4.2	Integer Arithmetic	33
4.3	Floating-Point Arithmetic	34
4.4	Mixed Mode Arithmetic	34
4.5	Assignment Operators	35
4.6	Increment And Decrement Operators	35
4.6.1	Prefix Increment / Decrement	36
4.6.2	Postfix Increment / Decrement	36
4.7	Relational Operators	37
4.8	Logical Or Boolean Operators	39
4.8.1	AND (&&) Operator	39
4.8.2	OR () Operator	40
4.8.3	NOT (!) Operator	40
4.9	Conditional Operator	41
4.10	Comma Operator	42
4.11	sizeof Operator	43

4.12 Type Conversion	44
4.12.1 Implicit Type Conversions	44
4.12.2 Automatic Conversions	44
4.12.3 Type Conversion In Assignment	45
4.12.4 Explicit Type Conversion Or Type Casting	46
4.13 Precedence And Associativity Of Operators	47
4.14 Role Of Parentheses In Evaluating Expressions	50
4.15 Order Of Evaluation Of Operands	53
Exercise	53
Programming Exercise	56
Answers	56
5. Control Statements	58-109
5.1 Compound Statement or Block	58
5.2 if...else	59
5.2.1 Nesting of if...else	61
5.2.2 else if Ladder	63
5.3 Loops	65
5.3.1 while loop	65
5.3.2 do...while loop	69
5.3.3 for loop	71
5.3.4 Nesting Of Loops	75
5.3.5 Infinite Loops	77
5.4 break statement	78
5.5 continue statement	80
5.6 goto	82
5.7 switch	84
5.8 Some Additional Problems	90
5.9 Pyramids	99
Exercise	103
Programming Exercise	108
Answers	109
6. Functions	110-157
6.1 Advantages Of Using Functions	110
6.2 Library Functions	110
6.3 User-Defined Functions	111
6.4 Function Definition	112
6.5 Function Call	113
6.6 Function Declaration	114
6.7 return statement	116
6.8 Function Arguments	118
6.9 Types Of Functions	120
6.9.1 Functions With No Arguments And No Return Value	120
6.9.2 Function With No Arguments But A Return Value	121
6.9.3 Function With Arguments But No Return Value	121

6.9.4	Function With Arguments And Return Value	123
6.10	More About Function Declaration	124
6.11	Declaration Of Functions With No Arguments	124
6.12	If Declaration Is Absent	125
6.13	Order Of Evaluation Of Function Arguments	125
6.14	main() Function	125
6.15	Library Functions	126
6.16	Old Style Of Function Declaration	126
6.17	Old Style Of Function Definition	126
6.18	Local, Global And Static Variables	130
6.18.1	Local Variables	130
6.18.2	Global Variables	131
6.18.3	Static Variables	132
6.19	Recursion	132
6.19.1	Tower Of Hanoi	136
6.19.2	Advantages And Disadvantages Of Recursion	139
6.19.3	Local Variables In Recursion	139
6.20	Some Additional Problems	140
Exercise		149
Programming Exercise		155
Answers		156

7. Arrays**158-195**

7.1	One Dimensional Array	158
7.1.1	Declaration of 1-D Array	158
7.1.2	Accessing 1-D Array Elements	159
7.1.3	Processing 1-D Arrays	160
7.1.4	Initialization of 1-D Array	162
7.1.5	1-D Arrays And Functions	165
7.1.5.1	Passing Individual Array Elements to a Function	165
7.1.5.2	Passing whole 1-D Array to a Function	165
7.2	Two Dimensional Array	167
7.2.1	Declaration and Accessing Individual Elements of a 2-D array	167
7.2.2	Processing 2-D Arrays	168
7.2.3	Initialization of 2-D Arrays	169
7.3	Arrays With More Than Two Dimensions	173
7.3.1	Multidimensional Array And Functions	174
7.4	Introduction To Strings	175
7.4.1	Input and output of strings	175
7.5	Some Additional Problems	175
Exercise		191
Programming Exercise		193
Answers		194

8. Pointers**196-252**

8.1	About Memory	196
-----	--------------	-----

8.2	Address Operator	197
8.3	Pointers Variables	197
8.3.1	Declaration Of Pointer Variables	198
8.3.2	Assigning Address To Pointer Variables	198
8.3.3	Dereferencing Pointer Variables	199
8.4	Pointer Arithmetic	201
8.5	Precedence Of Dereferencing Operator And Increment/Decrement Operators	204
8.6	Pointer Comparisons	206
8.7	Pointer To Pointer	206
8.8	Pointers and One Dimensional Arrays	208
8.9	Subscripting Pointer Variables	211
8.10	Pointer to an Array	212
8.11	Pointers And Two Dimensional Arrays	213
8.12	Subscripting Pointer To An Array	216
8.13	Pointers And Three Dimensional Arrays	217
8.14	Pointers And Functions	219
8.15	Returning More Than One Value From A Function	221
8.16	Function Returning Pointer	222
8.17	Passing a 1-D Array to a Function	223
8.18	Passing a 2-D Array to a Function	225
8.19	Array Of Pointers	227
8.20	void Pointers	229
8.21	Dynamic Memory Allocation	231
8.21.1	malloc()	231
8.21.2	calloc()	233
8.21.3	realloc()	233
8.21.4	free()	234
8.21.5	Dynamic Arrays	235
8.22	Pointers To Functions	238
8.22.1	Declaring A Pointer To A Function	239
8.22.2	Calling A Function Through Function Pointer	240
8.22.3	Passing a Function's Address as an Argument to Other Function	240
8.22.4	Using Arrays Of Function Pointers	242
	Exercise	244
	Answers	251
9.	Strings	253-287
9.1	String Constant or String Literal	253
9.2	String Variables	255
9.3	String Library Functions	257
9.3.1	strlen()	257
9.3.2	strcmp()	258
9.3.3	strcpy()	259
9.3.4	strcat()	261
9.4	String Pointers	262
9.5	Array Of Strings Or Two Dimensional Array Of Characters	264

9.6	Array Of Pointers To Strings	267
9.7	sprintf()	272
9.8	sscanf()	273
9.9	Some Additional Problems	274
	Exercise	280
	Programming Exercise	284
	Answers	286
10.	Structure And Union	288-333
10.1	Defining a Structure	288
10.2	Declaring Structure Variables	289
	10.2.1 With Structure Definition	289
	10.2.2 Using Structure Tag	289
10.3	Initialization Of Structure Variables	290
10.4	Accessing Members of a Structure	290
10.5	Assignment of Structure Variables	292
10.6	Storage of Structures in Memory	292
10.7	Size of Structure	293
10.8	Array of Structures	293
10.9	Arrays Within Structures	295
10.10	Nested Structures (Structure Within Structure)	296
10.11	Pointers to Structures	298
10.12	Pointers Within Structures	299
10.13	Structures And Functions	299
	10.13.1 Passing Structure Members As Arguments	299
	10.13.2 Passing Structure Variable As Argument	300
	10.13.3 Passing Pointers To Structures As Arguments	301
	10.13.4 Returning A Structure Variable From Function	302
	10.13.5 Returning A Pointer To Structure From A Function	303
	10.13.6 Passing Array Of Structures As Argument	303
10.14	Self Referential Structures	309
10.15	Linked List	309
	10.15.1 Traversing a Linked List	311
	10.15.2 Searching in a Linked List	311
	10.15.3 Insertion into a Linked List	311
	10.15.4 Insertion in the Beginning	312
	10.15.5 Insertion in Between or at the end	312
	10.15.6 Deletion From A Linked List	313
	10.15.7 Deletion of First Node	313
	10.15.8 Deletion of a Node in Between or at the End	314
	10.15.9 Creation Of List	314
	10.15.10 Reversing A Linked List	318
10.16	union	321
10.17	typedef	326
	Exercise	329
	Programming Exercise	332
	Answers	332

11. Files	334-376
11.1 Text And Binary Modes	334
11.2 Concept Of Buffer	335
11.3 Opening a File	335
11.3.1 Errors in Opening Files	337
11.4 Closing a File	337
11.5 End of File	338
11.6 Structure of a General File Program	338
11.7 Predefined File Pointers	339
11.8 Character I/O	339
11.8.1 fputc()	339
11.8.2 fgetc()	340
11.8.3 getc() and putc()	340
11.9 Integer I/O	341
11.9.1 putw()	341
11.9.2 getw()	341
11.10 String I/O	342
11.10.1 fputs()	342
11.10.2 fgets()	342
11.11 Formatted I/O	343
11.11.1 fprintf()	343
11.11.2 fscanf()	344
11.12 Block Read / Write	345
11.12.1 fwrite()	345
11.12.2 fread()	347
11.13 Random Access To File	348
11.13.1 fseek()	349
11.13.2 ftell()	350
11.13.3 rewind()	351
11.14 Other File Functions	362
11.14.1 feof()	362
11.14.2 perror()	363
11.14.3 clearerr()	364
11.14.4 perror()	364
11.14.5 rename()	364
11.14.6 unlink()	365
11.14.7 remove()	365
11.14.8 fflush()	366
11.14.9 tmpfile()	366
11.14.10 tmpnam()	366
11.14.11 freopen()	367
11.15 Command Line Arguments	367
11.16 Some Additional Problems	368
Exercise	374
Programming Exercise	375
Answers	376

12. The C Preprocessor	377-406
12.1 #define	378
12.2 Macros with Arguments	379
12.3 Nesting in Macros	381
12.4 Problems with Macros	382
12.5 Macros Vs Functions	385
12.6 Generic Functions	386
12.7 #undef	387
12.8 Stringizing Operator (#)	387
12.9 Token Pasting Operator(##)	388
12.10 Including Files	389
12.11 Conditional Compilation	389
12.11.1 #if And #endif	390
12.11.2 #else and #elif	390
12.11.3 defined Operator	393
12.11.4 #ifdef and #ifndef	393
12.11.5 Writing Portable Code	395
12.11.6 Debugging	396
12.11.7 Commenting A Part Of Code	397
12.11.8 Other Uses of conditional compilation	397
12.12 Predefined Macro Names	398
12.13 #line	399
12.14 #error	399
12.15 Null Directive	400
12.16 #pragma	400
12.17 How to see the code expanded by the Preprocessor	401
Exercise	401
Answers	405
13. Operations on Bits	407-432
13.1 Bitwise AND (&)	408
13.2 Bitwise OR ()	408
13.3 Bitwise XOR (^)	409
13.4 One's Complement (~)	410
13.5 Bitwise Left Shift (<<)	411
13.6 Bitwise Right Shift (>>)	411
13.7 Multiplication and Division by 2 using shift operators	412
13.8 Masking	413
13.8.1 Masking Using Bitwise AND	413
13.8.2 Masking Using Bitwise OR	415
13.8.3 Masking Using Bitwise XOR	415
13.8.4 Switching off Bits Using Bitwise AND and Complement Operator	416
13.9 Some additional Problems	418
13.10 Bit Fields	426
Exercise	429
Answers	431

14. Miscellaneous Features In C	433-463
14.1 Enumeration	433
14.2 Storage Classes	437
14.2.1 Automatic	437
14.2.2 External	439
14.2.3 Static	442
14.2.3.1 Local Static Variables	442
14.2.3.2 Global Static Variables	443
14.2.4 Register	444
14.3 Storage Classes in Functions	445
14.4 Linkage	445
14.5 Memory During Program Execution	445
14.6 const	447
14.7 volatile	449
14.8 Functions With Variable Number Of Arguments	450
14.8.1 Passing Variable Number of Arguments To Another Function	454
14.9 lvalue and rvalue	456
14.10 Compilation And Execution of C Programs	457
14.10.1 Preprocessor	457
14.10.2 Compiler	457
14.10.3 Assembler	457
14.10.4 Linker	457
Exercise	458
Answers	463

15. Building project and creation of library	464-486
15.1 Requirement Analysis	464
15.2 Top Level Design	465
15.3 Detail Design	465
15.4 Coding	468
15.4.1 Dtmanip.h	468
15.4.2 Datefmt.c	469
15.4.3 Valid.c	469
15.4.4 Leap.c	470
15.4.5 Julian.c	470
15.4.6 Weekday.c	471
15.4.7 Cmpdate.c	472
15.4.8 Diffymd.c	472
15.4.9 Diffdays.c	473
15.4.10 Addyear.c	474
15.4.11 Subyear.c	474
15.4.12 Addmonth.c	475
15.4.13 Submonth.c	475
15.4.14 Adddays.c	476
15.4.15 Subdays.c	477
15.4.16 Main.c	477

15.5	Building Project in Turbo C	481
15.6	Testing	481
15.7	Creation Of Library And Using it in your Program in Turbo C	482
	15.7.1 Deletion of a Module From Library	482
	15.7.2 Getting Modules From Library	483
	15.7.3 Changing Version Of Module In Library	483
15.8	Building Project On Unix	483
	15.8.1 Writing Makefile	484
	15.8.2 Building Project With Make	486
15.9	Creation Of Library And Using In Your Program in Unix	486
16.	Code Optimization in C	487-494
16.1	Optimization is a Technique	487
16.2	Optimization With Tool	487
16.3	Optimization With Loop	487
	16.3.1 Loop Unrolling	487
	16.3.2 Avoiding Calculations In Loops	488
16.4	Fast Mathematics	488
	16.4.1 Avoid Unnecessary Integer Division	488
	16.4.2 Multiplication And Division by Power Of 2	488
16.5	Simplifying Expressions	489
16.6	Declare prototypes for Functions	489
16.7	Better Way Of Calling Function	489
16.8	Prefer int to char or short	489
16.9	Use of Register Variables	490
16.10	Optimization With Switch Statement	490
16.11	Avoid Pointer Dereference	492
16.12	Prefer Pre Increment/Decrement to Post Increment/Decrement	492
16.13	Prefer Array to Memory Allocation	492
16.14	Use Array Style Code Rather Than Pointer	492
16.15	Expression Order Understanding	492
16.16	Declaration Of Local Function	493
16.17	Breaking Loop With Parallel Coding	493
16.18	Trick to use Common Expression	493
16.19	Declaring Local Variables Based On Size	494
16.20	Prefer Integer Comparison	494
16.21	Avoid String Comparison	494
17.	C and Assembly Interaction	495-504
17.1	Inline Assembly Language	495
17.2	Linking Of Two Assembly Files	498
	17.2.1 Memory Models	500
	17.2.2 C And Segments in Library	500
17.3	Linking Assembly Procedure in C Program	502

18. Library Functions

18.1 Mathematical Functions

18.1.1	abs()	50:
18.1.2	acos()	50:
18.1.3	asin()	50:
18.1.4	atan()	50:
18.1.5	atan2()	50:
18.1.6	cabs()	50:
18.1.7	ceil()	50:
18.1.8	cos()	50:
18.1.9	cosh()	50:
18.1.10	exp()	50:
18.1.11	fabs()	50:
18.1.12	floor()	50:
18.1.13	fmod()	50:
18.1.14	frexp()	50:
18.1.15	ldexp()	50:
18.1.16	log()	50:
18.1.17	log10()	50:
18.1.18	modf()	50:
18.1.19	pow()	50:
18.1.20	sin()	50:
18.1.21	sinh()	50:
18.1.22	sqrt()	50:
18.1.23	tan()	50:
18.1.24	tanh()	50:

18.2 Character Type Functions

18.2.1	isalnum()	508
18.2.2	isalpha()	508
18.2.3	iscntrl()	508
18.2.4	isdigit()	508
18.2.5	isgraph()	508
18.2.6	islower()	508
18.2.7	isprint()	508
18.2.8	ispunct()	508
18.2.9	isspace()	508
18.2.10	isupper()	509
18.2.11	isxdigit()	509
18.2.12	tolower()	509
18.2.13	toupper()	509

18.3 String Manipulation Functions

18.3.1	strcat()	509
18.3.2	strchr()	509
18.3.3	strcmp()	510
18.3.4	strcpy()	510
18.3.5	strcspn()	510

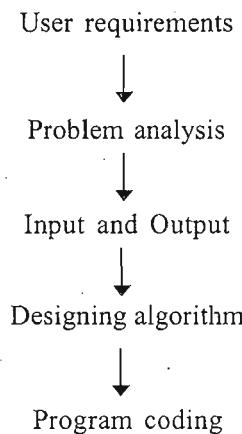
521	18.3.6	strlen()	510
505	18.3.7	strncat()	510
505	18.3.8	strcmp()	511
505	18.3.9	strcpy()	511
505	18.3.10	strpbrk()	512
505	18.3.11	strrchr()	512
505	18.3.12	strspn()	512
505	18.3.13	strstr()	512
506	18.4	Input/Output Functions	513
506	18.4.1	access()	513
506	18.4.2	chmod()	513
506	18.4.3	clearerr()	514
506	18.4.4	close()	514
506	18.4.5	creat()	514
506	18.4.6	fclose()	514
506	18.4.7	feof()	514
507	18.4.8	ferror()	514
507	18.4.9	fflush()	515
507	18.4.10	fgetc()	515
507	18.4.11	fgets()	515
507	18.4.12	fileno()	515
507	18.4.13	fopen()	515
507	18.4.14	fprintf()	515
507	18.4.15	fputc()	516
507	18.4.16	fputs()	516
507	18.4.17	fread()	516
508	18.4.18	fputchar()	516
508	18.4.19	fscanf()	516
508	18.4.20	fseek()	516
508	18.4.21	fstat()	517
508	18.4.22	ftell()	517
508	18.4.23	isatty()	517
508	18.4.24	open()	517
508	18.4.25	read()	518
508	18.4.26	remove()	518
508	18.4.27	rename()	518
509	18.4.28	setbuf()	518
509	18.4.29	sopen()	519
509	18.4.30	stat()	520
509	18.4.31	sprintf()	520
509	18.4.32	sscanf()	520
509	18.4.33	tell()	520
509	18.4.34	tmpfile()	520
510	18.4.35	tmpnam()	520
510	18.4.36	unlink()	521

Chapter 1

Introduction to C

Software is a collection of programs and a program is a collection of instructions given to the computer. Development of software is a stepwise process. Before developing a software, number of processes are done. The first step is to understand the user requirements. Problem analysis arises during the requirement phase of software development. Problem analysis is done for obtaining the user requirements and to determine the input and output of the program.

For solving the problem, an “algorithm” is implemented. Algorithm is a sequence of steps that gives method of solving a problem. This “algorithm” creates the logic of program. On the basis of this “algorithm”, program code is written. The steps before writing program code are as-



Process of program development

1.1 Design Methods

Designing is the first step for obtaining solution of a given problem. The purpose of designing is to represent the solution for the system. It is really difficult to design a large system because the complexity of system cannot be represented easily. So various methods have been evolved for designing.

1.1.1 Top-Down Design

Every system has several hierarchies of components. The top-level component represents the whole system. Top-Down design method starts from top-level component to lowest level (bottom) component. In this design method, the system is divided into some major components.

Then each major component is divided into lower level components. Similarly other components are divided till the lowest level component.

1.1.2 Bottom-Up Design

Bottom-Up design method is the reverse of Top-Down approach. It starts from the lowest level component to the highest-level component. It first designs the basic components and from these basic components the higher-level components are designed.

1.1.3 Modular Approach

It is better to divide a large system into modules. In terms of programming, module is logically a well-defined part of program. Each module is a separate part of the program. It is easy to modify a program written with modular approach because changes in one module don't affect other modules of program. It is also easy to check bugs in the program in module level programming.

1.2 Programming Languages

Before learning any language, it is important to know about the various types of languages and their features. It is interesting to know what were the basic requirements of the programmers and what difficulties they faced with the existing languages. The programming languages can be classified into two types-

1. Low level languages
2. High level languages

1.2.1 Low Level Languages

The languages in this category are the Machine level language and Assembly language.

1.2.1.1 Machine Level Language

Computers can understand only digital signals, which are in binary digits i.e. 0 and 1. So the instructions given to the computer can be only in binary codes. The machine language consists of instructions that are in binary 0 or 1. Computers can understand only machine level language.

Writing a program in machine level language is a difficult task because it is not easy for programmers to write instructions in binary code. A machine level language program is error-prone and its maintenance is very difficult. Furthermore machine language programs are not portable. Every computer has its own machine instructions, so the programs written for one computer are not valid for other computers.

1.2.1.2 Assembly Language

The difficulties faced in machine level language were reduced to some extent by using a modified form of machine level language called assembly language. In assembly language instructions are given in English like words, such as MOV, ADD, SUB etc. So it is easier to write and understand assembly programs. Since a computer can understand only machine level language, hence assembly language program must be translated into machine language. The translator that is used for translating is called "assembler".

Although writing programs in assembly language is a bit easier, but still the programmer has to know all the low level details related with the hardware of a computer. In assembly language, data is stored in computer registers and each computer has different set of registers. Hence the assembly language program is also not portable. Since the low level languages are related with the hardware, hence the execution of a low-level program is faster.

1.2.2 High-Level Languages

High-level languages are designed keeping in mind the features of portability i.e. these languages are machine independent. These are English like languages, so it is easy to write and understand the programs of high-level language. While programming in a high level language, the programmer is not concerned with the low level details, and so the whole attention can be paid to the logic of the problem being solved. For translating a high-level language program into machine language, compiler or interpreter is used. Every language has its own compiler or interpreter. Some languages in this category are- FORTRAN, COBOL, BASIC, Pascal etc.

1.3 Translators

We know that computers can understand only machine level language, which is in binary 1 or 0. It is difficult to write and maintain programs in machine level language. So the need arises for converting the code of high-level and low-level languages into machine level language and translators are used for this purpose. These translators are just computer programs, which accept a program written in high level or low-level language and produce an equivalent machine language program as output. The three types of translators used are-

- Assembler
- Compiler
- Interpreter

Assembler is used for converting the code of low-level language (assembly language) into machine level language.

Compilers and interpreters are used to convert the code of high-level language into machine language. The high level program is known as source program and the corresponding machine language program is known as object program. Although both compilers and interpreters perform the same task but there is a difference in their working.

A compiler searches all the errors of program and lists them. If the program is error free then it converts the code of program into machine code and then the program can be executed by separate commands. An interpreter checks the errors of program statement by statement. After checking one statement, it converts that statement into machine code and then executes that statement. This process continues until the last statement of program or an erroneous statement occurs.

1.4 History Of C

In earlier days, every language was designed for some specific purpose. For example FORTRAN (Formula Translator) was used for scientific and mathematical applications, COBOL (Common Business Oriented Language) was used for business applications. So need of such a language was felt which could withstand most of the purposes. "Necessity is the mother of invention". From here the first step towards C was put forward by Dennis Ritchie.

The C language was developed in 1970's at Bell laboratories by Dennis Ritchie. Initially it was designed for programming in the operating system called UNIX. After the advent of C, the whole UNIX operating system was rewritten using it. Now almost the entire UNIX operating system and the tools supplied with it including the C compiler itself are written in C.

The C language is derived from the B language, which was written by Ken Thompson at AT&T Bell laboratories. The B language was adopted from a language called BCPL (Basic Combined Programming Language), which was developed by Martin Richards at Cambridge University.

In 1982 a committee was formed by ANSI (American National Standards Institute) to standardize the C language. Finally in 1989, the standard for C language was introduced known as ANSI C. Generally most of the modern compilers conform to this standard.

1.5 Characteristics of C

It is a middle level language. It has the simplicity of a high level language as well as the power of low level language. This aspect of C makes it suitable for writing both application programs and system programs. Hence it is an excellent, efficient and general-purpose language for most of the application such as mathematical, scientific, business and system software applications.

C is small language, consisting of only 32 English words known as keywords (if, else, for, break etc.). The power of C is augmented by the library functions provided with it. Moreover, the language is extendible since it allows the users to add their own library functions to the library.

C contains control constructs needed to write a structured program hence it is considered a structure programming language. It includes structures for selection (if...else, switch), repetition (while, for do...while) and for loop exit (break).

The programs written in C are portable i.e. programs written for one type of computer or operating system can be run on another type of computer or operating system.

1.6 Structure of a C Program

C program is a collection of one or more functions. Every function is a collection of statements and performs some specific task. The general structure of C program is-

```

Comments
Preprocessor directives
Global variables
main( ) function
{
    local variables
    statements
    .....
    .....
}
func1( )
{
    local variables
    statements
    .....
    .....
}
func2( )
{
    local variables
    statements
    .....
    .....
}

```

Comments can be placed anywhere in a program and are enclosed between the delimiters /* and */. Comments are generally used for documentation purposes.

Preprocessor directives are processed through preprocessor before the C source code passes through compiler. The commonly used preprocessor directives are #include and #define. #include is used for including header files. #define is used to define symbolic constants and macros.

Every C program has one or more functions. If a program has only one function then it must be main(). Execution of every C program starts with main() function. It has two parts, declaration of local variables and statements. The scope of the local variable is local to that function only. Statements in the main() function are executed one by one. Other functions are the user-defined functions, which also have local variables and C statements. They can be defined before or after main(). It may be possible that some variables have to be used in many functions, so it is necessary to declare them globally. These variables are called global variables.

1.7 Environment For C

The steps for the execution of C program are as-

1. Program creation
2. Program compilation
3. Program execution

The C programs are written in mostly two environments, UNIX and MS-DOS.

1.7.1 Unix Environment

Generally a command line C compiler is provided with the UNIX operating system. This compiler is named cc or gcc.

(a) Program creation

In unix environment, file can be created with vi editor as-

```
$ vi filename.c
```

Here \$ is the unix prompt. The file can be saved by pressing ESC and SHIFT+zz.

(b) Program compilation

After creation of C program, it can be compiled as-

```
$ cc filename.c
```

If the program has mathematical function then it is compiled as-

```
$ cc filename.c -lm
```

After compilation, the executable code is stored in the file a.out .

(c) Program execution

After the compilation of program, it can be executed as-

```
$ a.out
```

1.7.2 MS-DOS Environment

In MS-DOS environment creation, compilation and execution can be done using command line or IDE (Integrated Development Environment).

1.7.2.1 Command Line

In Borland C, the command line compiler is bcc.exe and in Turbo C the command line compiler is tcc.exe.

(a) Program creation

The program file can be created using any editor and should be saved with .c extension

(b) Program compilation

After saving the file, C program can be compiled at DOS prompt by writing-

C:\>tcc filename (in Turbo C)

C:\>bcc filename (in Borland C)

(c) Program execution

After compilation of C program, the executable file filename.exe is created. It is executed at DOS prompt by writing-

C:\>filename

1.7.2.2 Integrated Development Environment

All these steps can be performed in an IDE using menu options or shortcut keys. In Borland C the program bc.exe is the IDE and in Turbo C the program tc.exe is the IDE. So we can open the IDE by typing bc or tc at the command prompt.

(a) Program creation

A new file can be created from menu option New. The file can be saved by menu option Save. If the file is unnamed then it is saved by menu option Save as. An existing file can be opened from the menu option Open.

(b) Program compilation

The file compiled by the menu option Compile. (Alt+F9)

(c) Program execution

The file can be executed by the menu option Run. (Ctrl+F9). The output appears in the output window that can be seen using the keys Alt+F5.

We have given you just a preliminary knowledge of how to execute your programs. There are several other options that you can explore while working and it is best to check the manual of your compiler to know about these options.

Chapter 2

Elements of C

Every language has some basic elements and grammatical rules. Before understanding programming, it is must to know the basic elements of C language. These basic elements are character set, variables, datatypes, constants, keywords (reserved words), variable declaration, expressions, statements etc. All of these are used to construct a C program.

2.1 C Character Set

The characters that are used in C programs are given below-

2.1.1 Alphabets

A, B, CZ

a, b, cz

2.1.2 Digits

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

2.1.3 Special characters

Character	Meaning	Character	Meaning
+	plus sign	-	minus sign(hyphen)
*	asterisk	%	percent sign
\	Backward slash	/	forward slash
<	less than sign	=	equal to sign
>	greater than sign	_	underscore
(left parenthesis)	right parenthesis
{	left braces	}	right braces
[left bracket	right bracket	
,	comma	.	period
'	single quotes	"	double quotes
:	colon	;	Semicolon
?	Question mark	!	Exclamation sign
&	ampersand		vertical bar
@	at the rate	^	caret sign
\$	dollar sign	#	hash sign
~	tilde sign	'	back quotation mark

2.2 Execution Characters/Escape Sequences

Characters are printed on the screen through the keyboard but some characters such as newline, tab, backspace cannot be printed like other normal characters. C supports the combination of backslash (\) and some characters from the C character set to print these characters.

These character combinations are known as escape sequences and are represented by two characters. The first character is “\” and second character is from the C character set. Some escape sequences are given below-

Escape Sequence	Meaning	ASCII Value	Purpose
\b	backspace	008	Moves the cursor to the previous position of the current line
\a	bell(alert)	007	Produces a beep sound for alert
\r	carriage return	013	Moves the cursor to beginning of the current line.
\n	newline	010	Moves the cursor to the beginning of the next line
\f	form feed	012	Moves the cursor to the initial position of the next logical page.
\0	null	000	Null
\v	vertical tab	011	Moves the cursor to next vertical tab position
\t	Horizontal tab	009	Moves the cursor to the next horizontal tab position.
\\\	backslash	092	Presents a character with backslash (\)

Blank, horizontal tab, vertical tab, newline, carriage return, form feed are known as whitespace in C language.

2.3 Trigraph Characters

There is a possibility that the keyboard doesn't print some characters. C supports the facility of “trigraph sequence” to print these characters. These trigraph sequences have three characters. First two are ‘??’ and third character is any character from C character set. Some trigraph sequences are as given below-

Trigraph Sequence	Symbol
??<	{ left brace
??>	} right brace
??([left bracket
??)] right bracket
??!	vertical bar
??/	\ backslash
??=	# hash sign
??-	~ tilde
??'	^ caret

2.4 Delimiters

Delimiters are used for syntactic meaning in C. These are as given below-

:	colon	used for label
;	semicolon	end of statement
()	parentheses	used in expression
[]	square brackets	used for array
{ }	curly braces	used for block of statements
#	hash	preprocessor directive
,	comma	variable delimiter

2.5 Reserved Words / Keywords

There are certain words that are reserved for doing specific tasks. These words are known as keywords and they have standard, predefined meaning in C. They are always written in lowercase. There are only 32 keywords available in C which are given below-

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

2.6 Identifiers

All the words that we'll use in our C programs will be either keywords or identifiers. Keywords are predefined and can't be changed by the user, while identifiers are user defined words and are used to give names to entities like variables, arrays, functions, structures etc. Rules for naming identifiers are given below-

- (1) The name should consist of only alphabets (both upper and lower case), digits and underscore sign(_).
- (2) First character should be an alphabet or underscore.
- (3) The name should not be a keyword.
- (4) Since C is case sensitive, the uppercase and lowercase letters are considered different. For example code, Code and CODE are three different identifiers.
- (5) An identifier name may be arbitrarily long. Some implementations of C recognize only the first eight characters, though most implementations recognize 31 characters. ANSI standard compilers recognize 31 characters.

The identifiers are generally given meaningful names. Some examples of valid identifier names-

Value a net_pay recl _data MARKS

Some examples of invalid identifier names are-

5bc	First character should be an alphabet or underscore
int	int is a keyword
rec#	# is a special character
avg no	blank space is not permitted

2.7 Data Types

C supports different types of data. Storage representation of these data types is different in memory. There are four fundamental datatypes in C, which are int, char, float and double.

'char' is used to store any single character, 'int' is used to store integer value, 'float' is used for storing single precision floating point number and 'double' is used for storing double precision floating point number. We can use type qualifiers with these basic types to get some more types.

There are two types of type qualifiers-

1. Size qualifiers - short, long
2. Sign qualifiers - signed, unsigned

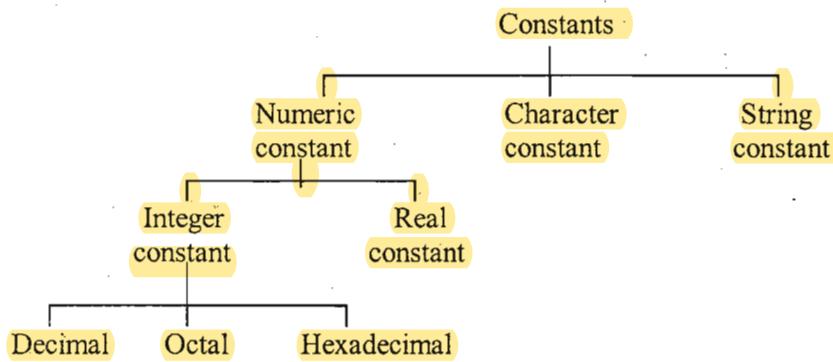
When the qualifier unsigned is used the number is always positive, and when signed is used number may be positive or negative. If the sign qualifier is not mentioned, then by default signed qualifier is assumed. The range of values for signed data types is less than that of unsigned type. This is because in signed type, the leftmost bit is used to represent the sign, while in unsigned type this bit is also used to represent the value.

The size and range of different data types on a 16-bit machine is given in the following table. The size and range may vary on machines with different word sizes.

Basic data types	Data types with type qualifiers	Size(bytes)	Range
char	char or signed char	1	-128 to 127
	unsigned char	1	0 to 255
	int or signed int	2	-32768 to 32767
		2	0 to 65535
		1	-128 to 127
		1	0 to 255
	long int or signed long int	4	-2147483648 to 2147483647
	unsigned long int	4	0 to 4294967295
	float	4	3.4E-38 to 3.4E+38
	double	8	1.7E-308 to 1.7E+308
	long double	10	3.4E-4932 to 1.1E+4932

2.8 Constants

Constant is a value that cannot be changed during execution of the program. There are three types of constants-



2.8.1 Numeric Constants

Numeric constants consist of numeric digits, they may or may not have decimal point(.). These are the rules for defining numeric constants-

1. Numeric constant should have at least one digit.
2. No comma or space is allowed within the numeric constant.
3. Numeric constants can either be positive or negative but default sign is always positive.

There are two types of numeric constants-

2.8.1.1 Integer constant

Integer constants are whole numbers which have no decimal point (.). There are three types of integer constants based on different number systems. The permissible characters that can be used in these constants are-

Decimal constants - 0,1,2,3,4,5,6,7,8,9 (base 10)

Octal constants - 0,1,2,3,4,5,6,7 (base 8)

Hex decimal constants - 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F,a,b,c,d,e,f (base 16)

Some valid decimal integer constants are-

0

123

3705

23759

Some invalid decimal integer constants are-

Invalid	Remark
2.5	illegal character (.)
3#5	illegal character (#)
98 5	No blank space allowed
0925	First digit can not be zero
8,354	Comma is not allowed

In octal integer constants, first digit must be 0. For example-

0

05

077

0324

In hexadecimal integer constants, first two characters should be 0x or 0X. Some examples are as-

0x

0X23

0x515

0XA15B

0xFFFF

0xac

By default the type of an integer constant is int. But if the value of integer constant exceeds the range of values represented by int type, the type is taken to be unsigned int or long int. We can also explicitly mention the type of the constant by suffixing it with l or L (for long), u or U (for unsigned), ul or UL (for unsigned long). For example-

6453

Integer constant of type int

45238722UL

or

45238722ul

Integer constant of type unsigned long int

6655U

or

6655u

Integer constant of type unsigned int

2.8.1.2 Real (floating point) Constants

Floating point constants are numeric constants that contain decimal point. Some valid floating point constants are-

0.5

5.3

4000.0

0.0073

5597.

39.0807

For expressing very large or very small real constants, exponential (scientific) form is used. Here the number is written in the mantissa and exponent form, which are separated by 'e' or 'E'. The mantissa can be an integer or a real number, while the exponent can be only an integer (positive or negative). For example the number 1800000 can be written as 1.8e6, here 1.8 is mantissa and 6 is the exponent. Some more examples are as-

Number			Exponential form
2500000000	→	$2.5 * 10^9$	2.5e9
0.0000076	→	$7.6 * 10^{-6}$	7.6e-6
-670000	→	$-6.7 * 10^5$	-6.7E5

By default the type of a floating point constant is double. We can explicitly mention the type of constant by suffixing it with a f or F (for float type), l or L (for long double). For example-

2.3e5

floating point constant of type double

2.4e-91 or 2.4e-9L

floating point constant of type long double

3.52f or 3.52F

floating point constant of type float

2.8.2 Character Constants

A character constant is a single character that is enclosed within single quotes. Some valid character constants are-

'9' 'D' '\$' ' ' '#'

Some invalid character constants are-

Invalid	Remark
'four'	There should be only one character within quotes
"d"	Double quotes are not allowed
" "	No character between single quotes
y	Single quotes missing

Every character constant has a unique integer value associated with it. This integer is the numeric value of the character in the machine's character code. If the machine is using ASCII (American Standard Code for Information Interchange), then the character 'G' represents integer value 71 and the character '5' represents value 53. Some ASCII values are-

- A - Z ASCII value (65 - 90)
- a - z ASCII value (97 - 122)
- 0 - 9 ASCII value (48 - 57)
- ; ASCII value (59)

ASCII values for all characters are given in Appendix A.

2.8.3 String Constants

A string constant has zero, one or more than one character. A string constant is enclosed within double quotes (""). At the end of string, \0 is automatically placed by the compiler.

Some examples of string constants are-

"Kumar"
"593"
"8"
" "
"A"

Note that "A" and 'A' are different, the first one is a string constant which consists of character A and \0 while the second one is a character constant which represents integer value 65.

2.8.4 Symbolic Constants

If we want to use a constant several times then we can provide it a name. For example if we have to use the constant 3.14159265 at many places in our program, then we can give it a name PI and use this name instead of writing the constant value everywhere. These types of constants are called symbolic constants or named constants.

A symbolic constant is a name that substitutes for a sequence of characters. The characters may represent a numeric constant, a character constant or a string constant.

These constants are generally defined at the beginning of the program as-

```
#define name value
```

Here 'name' is the symbolic name for the constant, and is generally written in uppercase letters. 'value' can be numeric, character or string constant.

Some examples of symbolic constants are as-

```
#define MAX 100
#define PI 3.14159625
#define CH 'y'
#define NAME "Suresh"
```

In the program, these names will be replaced by the corresponding values. These symbolic constants improve the readability and modifiability of the program.

2.9 Variables

Variable is a name that can be used to store values. Variables can take different values but one at a time. These values can be changed during execution of the program. A data type is associated with each variable. The data type of the variable decides what values it can take. The rules for naming variables are same as that for naming identifiers.

2.9.1 Declaration of Variables

It is must to declare a variable before it is used in the program. Declaration of a variable specifies its name and datatype. The type and range of values that a variable can store depends upon its datatype. The syntax of declaration of a variable is-

```
datatype variablename;
```

Here datatype may be int, float, char, double etc. Some examples of declaration of variables are-

```
int x;
float salary;
char grade;
```

Here x is a variable of type int, salary is a variable of type float, and grade is a variable of type char. We can also declare more than one variable in a single declaration. For example-

```
int x, y, z, total;
```

Here x, y, z, total are all variables of type int.

2.9.2 Initialisation of Variables

When a variable is declared it contains undefined value commonly known as garbage value. If we want we can assign some initial value to the variable during the declaration itself, this is called **initialisation** of the variable. For example-

```
int a = 5;
float x = 8.9, y = 10.5;
char ch = 'y';
double num = 0.15197e-7;
int l, m, n, total = 0;
```

In the last declaration only variable total has been initialised.

2.10 Expressions

An expression is a combination of operators, constants, variables and function calls. The expression can be arithmetic, logical or relational.

Some examples are as-

$x+y$	- arithmetic operation
$a = b+c$	- uses two operators ($=$) and ($+$)
$a > b$	- relational expression
$a == b$	- logical expression
$\text{func}(a, b)$	- function call

We'll study about these operators and expressions in the next chapter.

2.11 Statements

In a C program, instructions are written in the form of statements. A statement is an executable part of the program and causes the computer to carry out some action. Statements can be categorized as-

- (i) Expression statements
- (ii) Compound statements
- (iii) Selection statements (if, if...else, switch)
- (iv) Iterative statements (for, while, do...while)
- (v) Jump statements (goto, continue, break, return)
- (vi) Label statements (case, default, label statement used in goto)

2.11.1 Expression Statement

Expression statement consists of an expression followed by a semicolon. For example-

```
x = 5;
x = y - z;
func( a , b );
```

A statement that has only a semicolon is also known as null statement. For example-

```
; /* null statement */
```

2.11.2 Compound Statement

A compound statement consists of several statements enclosed within a pair of curly braces { }.

Compound statement is also known as block of statements. Note that there is no semicolon after the closing brace. For example-

```
int l=4,b=2,h=3;
int area,volume;
area=2*(l*b+b*h+h*l);
volume=l*b*h;
```

If any variable is to be declared inside the block then it can be declared only at the beginning of the block. The variables that are declared inside a block can be used only inside that block. All other categories of statements are discussed in further chapters.

2.12 Comments

Comments are used for increasing readability of the program. They explain the purpose of the program and are helpful in understanding the program. Comments are written inside /* and */. There can be single line or multiple line comments. We can write comments anywhere in a program except inside a string constant or a character constant.

Some examples of comments are-

/*Variable b represents basic salary*/	(single line comment)
/*This is a C program to calculate simple interest */	(multiple line comment)

Comments can't be nested i.e. we can't write a comment inside another comment.

Chapter 3

Input-Output In C

There are three main functions of any program- it takes data as input, processes this data and gives the output. The input operation involves movement of data from an input device (generally keyboard) to computer memory, while in output operation the data moves from computer memory to the output device (generally screen). C language does not provide any facility for input-output operations. The input output is performed through a set of library functions that are supplied with every C compiler. These functions are formally not a part of the language but they are considered standard for all input-output operations in C. The set of library functions that performs input-output operations is known as standard I/O library.

There are several header files that provide necessary information in support of the various library functions. These header files are entered in the program using the #include directive at the beginning of the program. For example if a program uses any function from the standard I/O library, then it should include the header file stdio.h as-

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

Similarly there are other header files like math.h, string.h, alloc.h that should be included when certain library functions are used.

In this chapter we'll discuss about the input functions scanf() and getchar() and the output functions printf() and putchar(). There are several other input-output functions that will be discussed in further chapters.

A simple method for taking the data as input is to give the value to the variables by assignment statement. For example-

```
int basic = 2000;  
char ch = 'y';
```

But in this way we can give only particular data to the variables.

The second method is to use the input function scanf(), which takes the input data from the keyboard. In this method we can give any value to the variables at run time. For output, we use the function printf().

3.1 Conversion Specifications

The functions scanf() and printf() make use of conversion specifications to specify the type and size of data. Each conversion specification must begin with a percent sign (%). Some conversion specification are as given below-

%c

a single character

%d	-	a decimal integer
%f	-	a floating point number
%e	-	a floating point number
%g	-	a floating point number
%lf	-	long range of floating point number (for double data type)
%h	-	a short integer
%o	-	an octal integer
%x	-	a hexadecimal integer
%i	-	a decimal, octal or hexadecimal integer
%s	-	a string
%u	-	an unsigned decimal integer

The modifier h can be used before conversion specifications d, i, o, u, x to specify short integer and the modifier l can be used before them to specify a long integer. The modifier l can be used before conversion specifications f, e, g to specify double while modifier L can be used before them to specify a long double. For example %ld, %hd, %Lf, %hx are valid conversion specifications.

3.2 Reading Input Data

Input data can be entered into the memory from a standard input device (keyboard). C provides the scanf() library function for entering input data. This function can take all types of values (numeric, character, string) as input. The scanf() function can be written as-

```
scanf("control string", address1, address2, ...);
```

This function should have at least two parameters. First parameter is a control string, which contains conversion specification characters. It should be within double quotes. The conversion specification characters may be one or more; it depends on the number of variables we want to input. The other parameters are addresses of variables. In the scanf() function at least one address should be present. The address of a variable is found by preceding the variable name by an ampersand (&) sign. This sign is called the address operator and it gives the starting address of the variable name in memory. A string variable is not preceded by & sign to get the address.

Some examples of scanf() function are as-

```
#include<stdio.h>
main()
{
    int marks;
    .....
    scanf("%d", &marks);
    .....
}
```

In this example, the control string contains only one conversion specification character %d, which implies that one integer value should be entered as input. This entered value will be stored in the variable marks.

```
#include<stdio.h>
main()
{
    char ch;
    .....
```

```
scanf("%c", &ch);
.....
```

Here the control string contains conversion specification character %c, which means that a single character should be entered as input. This entered value will be stored in the variable ch.

```
#include<sc
main( )
{
    float height;
    .....
    scanf("%f", &height);
    .....
```

Here the control string contains the conversion specification %f, which means that a floating point number should be entered as input. This entered value

will be stored in the variable height.

```
#include<stdio.h>
main( )
{
    char str[30] ;
    .....
    scanf("%s", str);
    .....
```

In this example control string has conversion specification character %s implying that a string should be taken as input. Note that the variable str is not preceded by ampersand(&) sign. The entered string will be stored in the variable str.

More than one value can also be entered by single scanf() function. For example-

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

    int basic, da;
    .....
    scanf("%d%d", &basic, &da);
    .....
```

Here the control string has two conversion specification characters implying that two integer values should be entered. These values are stored in the variables basic and da. The data can be entered with space as the delimiter as-

1500 1200

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

    int basic;
    float hra;
```

```

char grade;
.....
scanf("%d %f %c",&basic,&hra,&grade);
.....
}

```

Here the control string has three conversion specifications characters %d, %f and %c, means that one integer value, one floating point value and one single character can be entered as input. These values are stored in the variables basic, hra and grade. The input data can be entered as-

1500 200.50 A

When more than one values are input by scanf(), these values can be separated by whitespace characters like space, tab or newline (default). A specific character can also be placed between two conversion specification characters as a delimiter.

```

#include<stdio.h>
main( )
{
    int basic;
    float hra;
    .....
    scanf("%d;%f",&basic,&hra);
    .....
}

```

Here the delimiter is colon (:). The input data can be entered as-

1500:200.50

The value 1500 is stored in variable basic and 200.50 is stored in hra.

```

#include<stdio.h>
main( )
{
    int basic;
    float hra;
    .....
    scanf("%d,%f",&basic,&hra);
    .....
}

```

Here the delimiter is comma (,). The input data can be entered as-

1500, 200.40

```

#include<stdio.h>
main( )
{
    int day,month,year;
    int basic;
    .....
    scanf("%d-%d-%d",&day,&month,&year);
    scanf("$%d",&basic);
    .....
}

```

Here if the data is entered as-

24-5-1973

\$3000

Then 24 is stored in variable day, 5 is stored in variable month and 1973 is stored in variable year and 3000 is stored in variable basic.

If we include any spaces between the conversion specifications inside the control string, then they are just ignored.

```
#include<stdio.h>
main( )
{
    int x,y,z;
    .....
    scanf("%d %d %d", &x, &y, &z);
    .....
}
```

If the data is entered as-

12 34 56

Then 12 is stored in x, 34 is stored in y and 56 is stored in z.

3.3 Writing Output Data

Output data can be written from computer memory to the standard output device (monitor) using printf() library function. With this function all type of values (numeric, character or string) can be written as output. The printf() function can be written as-

`printf("control string", variable 1, variable 2,.....);`

In this function the control string contains conversion specification characters and text. It should be enclosed within double quotes. The name of variables should not be preceded by an ampersand(&) sign. If the control string does not contain any conversion specification, then the variable names are not specified. Some example of printf() function are as-

```
#include<stdio.h>
main( )
{
    printf("C is excellent\n");
}
```

Output:

C is excellent

Here control string has only text and no conversion specification character, hence the output is only text.

```
#include<stdio.h>
main( )
{
    int age;
    printf("Enter your age : ");
    scanf("%d", &age);
}
```

Here also printf does not contain any conversion specification character and is used to display a message that tells the user to enter his age.

```
#include<stdio.h>
main( )
{
    int basic=2000;
    .....
    printf("%d",basic);
    .....
}
```

In this example control string contains a conversion specification character %d, which implies that an integer value will be displayed. The variable basic has that integer value which will be displayed as output.

```
#include<stdio.h>
main( )
{
    float height=5.6;
    .....
    printf("%f",height);
    .....
}
```

Here control string has conversion specification character %f, which means that floating point number will be displayed. The variable height has that floating point value which will be displayed as output.

```
#include<stdio.h>
main( )
{
    char ch='$';
    .....
    printf("%c",ch);
    .....
}
```

In the above example, the control string has conversion specification character %c, means that a single character will be displayed and variable ch has that character value.

```
#include<stdio.h>
main( )
{
    char str[30];
    .....
    printf("%s",str);
    .....
}
```

Here control string has conversion specification character %s, implying that a string will be displayed and variable name str is a character array, holding the string which will be displayed.

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

```
{
    int basic=2000;
    printf("Basic Salary = %d", basic);
}
```

Output:

Basic Salary = 2000

Here the control string contains text with conversion specification character %d. The text will be displayed as it is, and the value of variable basic will be displayed in place of %d.

```
#include<stdio.h>
main()
{
    int b=1500;
    float h=200.50;
    char g='A';
    printf("Basic = %d , HRA = %f , Grade = %c",b,h,g);
```

Output:

Basic = 1500 , HRA = 200.500000 , Grade = A

Here control string contains text with three conversion specification characters %d, %f and %c. %d is for integer value, %f is for floating point number and %c is for a single character.

```
#include<stdio.h>
main()
{
    int num=10;
    printf("Octal equivalent of decimal %d = %o",num,num);
```

Output:

Octal equivalent of decimal 10 = 12

Here the second conversion specification character is %o hence the octal equivalent of the decimal number stored in the variable num is displayed.

```
#include<stdio.h>
main()
{
    int num=10;
    printf("Hex equivalent of decimal %d = %x",num,num);
```

Output:

Hex equivalent of decimal 10 = A

Here the second conversion specification character is % x hence the hexadecimal equivalent of the decimal number stored in the variable num is displayed.

In chapter 2 we had studied about escape sequences. Here we'll see how we can use them in the printf statement. The most commonly used escape sequences used are '\n' and '\t'.

```
#include<stdio.h>
main( )
{
    int b=1500 ;
    float h=200.50 ;
    char g='A';
    printf("Basic = %d\nHRA = %f\nGrade = %c\n",b,h,g);
}
```

Output:

Basic = 1500
 HRA = 200.500000
 Grade = A

'\n' moves the cursor to the beginning of next line. Here we have placed a '\n' at the end of control string also, it ensures that the output of the next program starts at a new line.

```
#include<stdio.h>
main( )
{
    int b=1500;
    float h=200.50;
    char g='A';
    printf("Basic = %d\tHRA = %f\tGrade = %c\n",b,h,g);
}
```

Output:

Basic = 1500 HRA = 200.500000 Grade = A

'\t' moves the cursor to the next tab stop. Similarly we can use other escape sequences also. For example '\b' moves the cursor one position back, '\r' moves the cursor to the beginning of the current line and '\a' alerts the user by a beep sound. '\v' moves the cursor to the next vertical tab position(first column of the next line), and '\f' move the cursor to the next page. '\v' and '\f' are effective only when output is printed through a printer.

If we want to print characters like single quotes ('), double quotes (") or the backslash character (\), then we have to precede them by a backslash character in the format string.

For example-

printf("9 \\ 11 \\ 1978");	will print	9 \ 11 \ 1978
printf("She said, \"I have to go\".");	will print	She said, "I have to go".

3.4 Formatted Input And Output

Formatted input and output means that data is entered and displayed in a particular format. Through format specifications, better presentation of result can be obtained. Formats for different specifications are as-

3.4.1 Format For Integer Input

%wd

Here 'd' is the conversion specification character for integer value and 'w' is an integer number specifying

the maximum field width of input data. If the length of input is more than this maximum field width then the values are not stored correctly. For example-

```
scanf ("%2d%3d", &a, &b );
```

- (i) When input data length is less than the given field width, then the input values are unaltered and stored in given variables.

Input-

6 39

Result-

6 is stored in a and 39 is stored in b.

- (ii) When input data length is equal to the given field width, then the input values are unaltered and stored in given variables.

Input-

26 394

Result-

26 is stored in a and 394 is stored in b.

- (iii) When input data length is more than the given field width, then the input values are altered and stored in the variable as -

Input-

269 3845

Result-

26 is stored in a and 9 is stored in b and the rest of input is ignored.

3.4.2 Format For Integer Output

%wd

Here w is the integer number specifying the minimum field width of the output data. If the length of the variable is less than the specified field width, then the variable is right justified with leading blanks. For example -

```
printf("a=%3d, b=%4d", a, b );
```

- (i) When the length of variable is less than the width specifier.

Value of variables-

78 9

Output:

a =	7	8	,	b =				9
-----	---	---	---	-----	--	--	--	---

The width specifier of first data is 3 while there are only 2 digits in it, so there is one leading blank. The width specifier of second data is 4 while there is only 1 digit, so there are 3 leading blanks.

- (ii) When the length of the variable is equal to the width specifier

Value of variables-

263 1941

Output:

a	=	2	6	3	,	b	=	1	9	4	1
---	---	---	---	---	---	---	---	---	---	---	---

- (iii) When length of variable is more than the width specifier, then also the output is printed correctly.

Value of variables-

2691 19412

Output:

a	=	2	6	9	1	,	b	=	1	9	4	1	2
---	---	---	---	---	---	---	---	---	---	---	---	---	---

```
#include<stdio.h>
main( )
{
    int a=4000,b=200,c=15;
    printf("a = %d \nb = %d \nc = %d\n",a,b,c);
    printf("%a = %4d \n%b = %4d \n%c = %4d\n",a,b,c);
}
```

The output of the first printf would be-

a = 4000
b = 200
c = 15

while the output of second printf would be-

a = 4000
b = 200
c = 15

3.4.3 Format For Floating Point Numeric Input

% wf

Here 'w' is the integer number specifying the total width of the input data (including the digits before and after decimal and the decimal itself). For example-

scanf("%3f %4f ", &x, &y);

- (i) When input data length is less than the given width, values are unaltered and stored in the variables.

Input

5 5.9

Result

5.0 is stored in x and 5.90 is stored in y.

- (ii) When input data length is equal to the given width, then the given values are unaltered and stored in the given variables.

Input

5.3 5.92

Result

5.3 is stored in x and 5.92 is stored in y

- (iii) When input data length is more than the given width then the given values are altered and stored in the given variables as-

Input

5.93	65.87
------	-------

Result

5.9 is stored in x and 3.00 is stored in y.

3.4.4 Format For Floating Point Numeric Output**%w.nf**

Here w is the integer number specifying the total width of the input data and n is the number of digits to be printed after decimal point. By default 6 digits are printed after the decimal. For example-

```
printf("x = %4.1f, y = %7.2f ", x, y);
```

If the total length of the variable is less than the specified width 'w', then the value is right justified with leading blanks. If the number of digits after decimal is more than 'n' then the digits are rounded off.

Value of variables-

8 5.9

Output:

x =	8	.	0	,	y =	5	.	9	0
-----	---	---	---	---	-----	---	---	---	---

Value of variables-

25.3 1635.92

Output:

x =	2	5	.	3	,	y =	1	6	3	5	.	9	2
-----	---	---	---	---	---	-----	---	---	---	---	---	---	---

Value of variables

15.231 65.875948

Output:

x =	1	5	.	2	,	y =	6	5	.	8	8
-----	---	---	---	---	---	-----	---	---	---	---	---

3.4.5 Format For String Input**% ws**

Here w specifies the total number of characters that will be stored in the string.

```
char str [8] ;
```

```
scanf ("%3s", str );
```

If the input is-

Srivastava

only first three characters of this input will be stored in the string, so the characters in the string will be-

‘S’, ‘r’, ‘i’, ‘\0’

The null character(‘\0’) is automatically stored at the end.

3.4.6 Format For String Output

%w.ns

Here w is the specified field width. Decimal point and ‘n’ are optional. If present then ‘n’ specifies that only first n characters of the string will be displayed and (w - n) leading blanks are displayed before string.

(i) printf("%3s", "sureshkumar");

s	u	r	e	s	h	k	u	m	a	r
---	---	---	---	---	---	---	---	---	---	---

(ii) printf("%10s", "reeta");

					r	e	e	t	a
--	--	--	--	--	---	---	---	---	---

(iii) printf("%.3s", "sureshkumar ");

s	u	r
---	---	---

(iv) printf("% 8.3s", "sureshkumar ");

				s	u	r
--	--	--	--	---	---	---

(8 - 3 = 5 leading blanks)

3.5 Suppression Character in scanf()

If we want to skip any input field then we specify * between the % sign and the conversion specification. The input field is read but its value is not assigned to any address. This character * is called the suppression character. For example-

scanf ("%d %*d %d", &a, &b, &c);

Input:

25 30 35

Here 25 is stored in ‘a’ , 30 is skipped and 35 is stored in the ‘b’. Since no data is available for ‘c’ so it has garbage value.

scanf("%d %*c %d %*c %d", &d, &m, &y);

Input:

3/1/2003

Here 3 will be stored in d, then / will be skipped, 11 will be stored in m, again / will be skipped and finally 2003 will be stored in y.

```
#include<stdio.h>
main( )
{
    int a,b,c;
    printf("Enter three numbers :");
```

```
scanf("%d %*d %d", &a, &b, &c);
printf("%d %d %d", a, b, c);
```

Output:

Enter three numbers : 25 30 35

25 35 25381

The variable c has garbage value.

3.6 Character I/O**3.6.1 getchar() and putchar()**

These macros getchar() and putchar() can be used for character I/O. getchar() reads a single character from the standard input. putchar() outputs one character at a time to the standard output.

```
#include<stdio.h>
main()
{
    char ch;
    printf("Enter a character : ");
    ch=getchar();
    printf("The entered character is : ");
    putchar(ch);
}
```

Output:

Enter a character : B

The entered character is : B

Exercise

Assume stdio.h is included in all programs.

```
(1) #define MSSG "Hello World\n"
main()
{
    printf(MSSG);
}

(2) main()
{
    printf("Indian\b is great\n");
    printf("New\rDelhi\n");
}

(3) main()
{
    int a=11;
    printf("a = %d\t",a);
    printf("a = %o\t",a);
    printf("a = %x\n",a);
}
```

```
(4) main()
{
    int a=50000;
    unsigned int b=50000;
    printf("a = %d, b = %u\n",a,b);
}

(5) main()
{
    char ch;
    printf("Enter a character:");
    scanf("%c",&ch);
    printf("%d\n",ch);
}

(6) main()
{
    float b=123.1265;
    printf("%f\t",b);
    printf("%.2f\t",b);
    printf("%.3f\n",b);
}

(7) main()
{
    int a=625,b=2394,c=12345;
    printf("%5d, %5d, %5d\n",a,b,c);
    printf("%3d, %4d, %5d\n",a,b,c);
}

(8) main()
{
    int a=98;
    char ch='c';
    printf("%c, %d\n",a,ch);
}

(9) main()
{
    float a1,b1,a2,b2,a3,b3;
    a1=2;
    b1=6.8;
    a2=4.2;
    b2=3.57;
    a3=9.82;
    b3=85.673;
    printf("%3.1f,%4.2f\n",a1,b1);
    printf("%5.1f,%6.2f\n",a2,b2);
    printf("%7.1f,%8.2f\n",a3,b3);
}
```

```
(10) main( )
{
    printf("%10s\n", "India");
    printf("%4s\n", "India");
    printf(".2s\n", "India");
    printf("%5.2s\n", "India");
}
```

Answers

- (1) Hello World
- (2) India is great
Delhi
\b takes the cursor to the previous position of current line, \r takes the cursor to the beginning of current line, \n takes the cursor to the beginning of next line.
- (3) a = 11 a = 13 a = b
- (4) a = -15536, b=50000
The value 50000 is outside the range of int data type.
- (5) This program enters a character and prints its ASCII value.
- (6) 123.126503 123.13 123.127
- (7) 625, 2394, 12345
625, 2394, 12345
- (8) b, 99
- (9) 2.0,6.80
4.2, 3.57
9.8, 85.67
- (10) India
India
In
In

Chapter 4

Operators And Expressions

An operator specifies an operation to be performed that yields a value. The variables, constants can be joined by various operators to form an expression. An operand is a data item on which an operator acts. Some operators require two operands, while others act upon only one operand. C includes a large number of operators that fall under several different categories, which are as-

1. Arithmetic operators
2. Assignment operators
3. Increment and Decrement operators
4. Relational operators
5. Logical operators
6. Conditional operator
7. Comma operator
8. sizeof operator
9. Bitwise operators
10. Other operators

4.1 Arithmetic Operators

Arithmetic operators are used for numeric calculations. They are of two types -

1. Unary arithmetic operators
2. Binary arithmetic operators

4.1.1 Unary Arithmetic Operators

Unary operators require only one operand. For example-

+x -y

Here '-' changes the sign of the operand y.

4.1.2. Binary Arithmetic Operators

Binary operators require two operands. There are five binary arithmetic operators-

Operator	Purpose
+	addition
-	subtraction
*	multiplication
/	division
%	gives the remainder in integer division

`%` (modulus operator) cannot be applied with floating point operands. There is no exponent operator in C. However there is a library function `pow()` to carry out exponentiation operation.

Note that unary plus and unary minus operators are different from the addition and subtraction operators.

4.2 Integer Arithmetic

When both operands are integers then the arithmetic operation with these operands is called integer arithmetic and the resulting value is always an integer. Let us take two variables `a` and `b`. The value of `a = 17` and `b = 4`, the results of the following operations are-

Expression	Result
<code>a+b</code>	21
<code>a-b</code>	13
<code>a*b</code>	68
<code>a/b</code>	4 (decimal part truncates)
<code>a%b</code>	1 (Remainder after integer division)

After division operation the decimal part will be truncated and result is only integer part of quotient. After modulus operation the result will be remainder part of integer division. The second operand must be nonzero for division and modulus operations.

```
/*P4.1 Program to understand the integer arithmetic operation*/
#include<stdio.h>
main()
{
    int a=17,b=4;
    printf("Sum = %d\n",a+b);
    printf("Difference = %d\n",a-b);
    printf("Product = %d\n",a*b);
    printf("Quotient = %d\n",a/b);
    printf("Remainder = %d\n",a%b);
}
```

Output:

```
Sum = 21
Difference = 13
Product = 68
Quotient = 4
Remainder = 1
```

4.3 Floating-Point Arithmetic

When both operands are of float type then the arithmetic operation with these operands is called floating point arithmetic. Let us take two variables a and b. The value of a = 12.4 and b = 3.1, the results of the following operations are as-

Expression	Result
a+b	15.5
a-b	9.3
a*b	38.44
a / b	4.0

The modulus operator % cannot be used with floating point numbers.

```
/*P4.2 Program to understand the floating point arithmetic operation */
#include<stdio.h>
main ( )
{
    float a=12.4,b=3.8;
    printf("Sum = %.2f\n",a+b);
    printf("Difference = %.2f\n",a-b);
    printf("Product = %.2f\n",a*b);
    printf("a/b = %.2f\n",a/b);
}
```

Output

Sum = 16.20
 Difference = 8.60
 Product = 47.12
 a/b = 3.26

4.4 Mixed Mode Arithmetic

When one operand is of integer type and the other is of floating type then the arithmetic operation with these operands is known as mixed mode arithmetic and the resulting value is float type.

If a = 12, b = 2.5

Expression	Result
a+b	14.5
a-b	9.5
a*b	30.0
a / b	4.8

Sometimes mixed mode arithmetic can help in getting exact results. For example the result of expression 5/2 will be 2, since integer arithmetic is applied. If we want exact result we can make one of the operands float type. For example 5.0/2 or 5/2.0, both will give result 2.5.

4.5 Assignment Operators

A value can be stored in a variable with the use of assignment operator. This assignment operator “= ” is used in assignment expressions and assignment statements.

The operand on the left hand side should be a variable, while the operand on the right hand side can be any variable, constant or expression. The value of right hand operand is assigned to the left hand operand. Here are some examples of assignment expressions-

```
x = 8          /* 8 is assigned to x*/
y = 5          /* 5 is assigned to y*/
s = x+y-2     /* Value of expression x+y-2 is assigned to s*/
y = x          /* Value of x is assigned to y*/
x = y          /* Value of y is assigned to x*/
```

The value that is being assigned is considered as value of the assignment expression. For example $= 8$ is an assignment expression whose value is 8.

We can have multiple assignment expressions also, for example-

```
x = y = z = 20
```

Here all the three variables x, y, z will be assigned value 20, and the value of the whole expression will be 20.

If we put a semicolon after the assignment expression then it becomes an assignment statement.

For example these are assignment statements-

```
x = 8;
y = 5;
s = x+y-2;
x = y = z = 20;
```

When the variable on the left hand side of assignment operator also occurs on right hand side then we can avoid writing the variable twice by using compound assignment operators. For example-

```
x = x + 5
```

can also be written as-

```
x += 5
```

Here $+=$ is a compound assignment operator.

Similarly we have other compound assignment operators-

$x -= 5$	is equivalent to	$x = x - 5$
$y *= 5$	is equivalent to	$y = y * 5$
$sum /= 5$	is equivalent to	$sum = sum / 5$
$k \% = 5$	is equivalent to	$k = k \% 5$

4.6 Increment And Decrement Operators

C has two useful operators increment ($++$) and decrement ($--$). These are unary operators because they operate on a single operand. The increment operator ($++$) increments the value of the variable by 1 and decrement operator ($--$) decrements the value of the variable by 1.

$++x$	is equivalent to	$x = x + 1$
$--x$	is equivalent to	$x = x - 1$

These operators should be used only with variables; they can't be used with constants or expressions. For example the expressions $++5$ or $++(x+y+z)$ are invalid.

These operators are of two types-

1. Prefix increment / decrement – operator is written before the operand (e.g. $++x$ or $--x$)
2. Postfix increment / decrement – operator is written after the operand (e.g. $x++$ or $x--$)

4.6.1 Prefix Increment / Decrement

Here first the value of variable is incremented / decremented then the new value is used in the operation
Let us take a variable x whose value is 3.

The statement $y = ++x$; means first increment the value of x by 1, then assign the value of x to y
This single statement is equivalent to these two statements-

```
x = x+1;
y = x;
```

Hence now value of x is 4 and value of y is 4.

The statement $y = --x$; means first decrement the value of x by 1 then assign the value of x to y
This statement is equivalent to these two statements.

```
x = x-1;
y = x;
```

Hence now value of x is 3 and value of y is 3.

```
/*P4.3 Program to understand the use of prefix increment / decrement *
#include<stdio.h>
main()
{
    int x=8;
    printf("x = %d\t",x);
    printf("x = %d\t",++x); /*prefix increment*/
    printf("x = %d\t",x);
    printf("x = %d\t",--x); /*prefix decrement*/
    printf("x = %d\n",x);
}
```

Output:

```
x = 8      x = 9  x = 9  x = 8  x = 8
```

In the second printf statement, first the value of x is incremented and then printed; similarly in the four printf statement first the value of x is decremented and then printed.

4.6.2 Postfix Increment / Decrement

Here first the value of variable is used in the operation and then increment/decrement is performed
Let us take a variable whose value is 3.

The statement $y = x++$; means first the value of x is assigned to y and then x is incremented. The statement is equivalent to these two statements-

```
y = x;
x = x+1;
```

Hence now value of x is 4 and value of y is 3.

The statement $y = x -$; means first the value of x is assigned to y and then x is decremented.

This statement is equivalent to these two statements-

```
y = x;
x = x-1;
```

Hence now value of x is 3 and value of y is 4.

```
/*P4.4 Program to understand the use of postfix increment/decrement*/
#include<stdio.h>
main()
{
    int x=8;
    printf("x = %d\n",x);
    printf("x = %d\t",x++); /*postfix increment*/
    printf("x = %d\t",x);
    printf("x = %d\t",x--); /*postfix decrement*/
    printf("x = %d\n",x);
}
```

Output:

```
x = 8      x = 8          x = 9          x = 9          x = 8
```

In the second printf statement, first the value of x is printed and then incremented; similarly in the fourth printf statement first the value of x is printed and then decremented.

4.7 Relational Operators

Relational operators are used to compare values of two expressions depending on their relations. An expression that contains relational operators is called relational expression. If the relation is true then the value of relational expression is 1 and if the relation is false then the value of expression is 0. The relational operators are-

Operator	Meaning
<	less than
\leq	less than or equal to
$= =$	equal to
\neq	Not equal to
>	Greater than
\geq	Greater than or equal to

Let us take two variables $a = 9$ and $b = 5$, and form simple relational expressions with them-

Expression	Relation	Value of Expression
a < b	False	0
a <= b	False	0
a == b	False	0
a != b	True	1
a > b	True	1
a >= b	True	1
a == 0	False	0
b!=0	True	1
a>8	True	1
2 > 4	False	0

The relational operators are generally used in if...else construct and loops. In our next program we'll use the **if** statement to illustrate the use of relational operators. The if control statement evaluates an expression, and if this expression is true(non zero) then the next statement is executed, otherwise the next statement is skipped. The details of if statement are discussed in chapter 5. Here we have used it to give you an idea of how the relational operators are used.

```
/*P4.5 Program to understand the use of relational operators*/
#include<stdio.h>
main( )
{
    int a,b;
    printf("Enter values for a and b : ");
    scanf("%d%d",&a,&b);
    if(a<b)
        printf("%d is less than %d\n",a,b);
    if(a<=b)
        printf("%d is less than or equal to %d\n",a,b);
    if(a==b)
        printf("%d is equal to %d\n",a,b);
    if(a!=b)
        printf("%d is not equal to %d\n",a,b);
    if(a>b)
        printf("%d is greater than %d\n",a,b);
    if(a>=b)
        printf("%d is greater than or equal to %d\n",a,b);
}
```

Output:

Enter values for a and b : 12 7

12 is not equal to 7

12 is greater than 7

12 is greater than or equal to 7

It is important to note that the assignment operator(=) and equality operator(==) are entirely different. Assignment operator is used for assigning values while equality operator is used to compare t

expressions. Beginners generally confuse between the two and use one in the place of another, and this leads to an error difficult to find out. For example if in the above program we use '=' instead of '==', then we'll get wrong output.

```
if(a = b)
    printf("%d is equal to %d\n",a,b);
```

Here $a = b$ is treated as an assignment expression, so the value of b is assigned to variable a , and the value of the whole expression becomes 7 which is non-zero, and so the next statement is executed.

4.8 Logical Or Boolean Operators

An expression that combines two or more expressions is termed as a logical expression. For combining these expressions we use logical operators. These operators return 0 for false and 1 for true. The operands may be constants, variables or expressions. C has three logical operators.

Operator	Meaning
&&	AND
	OR
!	NOT

Here logical NOT is a unary operator while the other two are binary operators. Before studying these operators let us understand the concept of true and false. In C any non-zero value is regarded as true and zero is regarded as false.

4.8.1 AND (&&) Operator

This operator gives the net result true if both the conditions are true, otherwise the result is false.

Boolean Table

Condition1	Condition2	Result
False	False	False
False	True	False
True	False	False
True	True	True

Let us take three variables $a = 10$, $b = 5$, $c = 0$

Suppose we have a logical expression-

$(a == 10) \&\& (b < a)$

Here both the conditions $a == 10$ and $b < a$ are true, and hence this whole expression is true. Since the logical operators return 1 for true hence the value of this expression is 1.

Expression	Result	Value of expression
$(a == 10) \&\& (b > a)$	false	0
$(b \geq a) \&\& (b == 3)$	false	0
$a \&\& b$	true	1
$a \&\& c$	false	0

In the last two expressions we have taken only variables. Since nonzero values are regarded as true and zero value is regarded as false, so variables a and b are considered true and variable c is considered false.

4.8.2 OR (||) Operator

This operator gives the net result false, if both the conditions have the value false, otherwise the result is true.

Boolean Table

Condition1	Condition2	Result
False	False	False
False	True	True
True	False	True
True	True	True

Let us take three variables $a = 10, b = 5, c = 0$

Consider the logical expression-

$$(a \geq b) \mid\mid (b > 15)$$

This gives result true because one condition is true.

Expression	Result	Value of expression
$a \mid\mid b$	true $\mid\mid$ true	1
$a \mid\mid c$	true $\mid\mid$ false	1
$(a < 9) \mid\mid (b > 10)$	false $\mid\mid$ false	0
$(b \neq 7) \mid\mid c$	true $\mid\mid$ false	1

4.8.3 Not (!) Operator

This is a unary operator and it negates the value of the condition. If the value of the condition is false then it gives the result true. If the value of the condition is true then it gives the result false.

Boolean Table

Condition	Result
False	True
True	False

Let us take three variables $a = 10, b = 5, c = 0$

Suppose we have this logical expression-

$$! (a == 10)$$

The value of the condition $(a == 10)$ is true. NOT operator negates the value of the condition. Hence the result is false.

Expression		Result	Value of expression
!a	!true	false	0
!c	!false	true	1
!(b>c)	!true	false	0
!(a && c)	!false	true	1

4.9 Conditional Operator

Conditional operator is a ternary operator (? and :) which requires three expressions as operands. This is written as-

TestExpression ? expression1 : expression2

Firstly the TestExpression is evaluated.

- (i) If TestExpression is true(nonzero), then expression1 is evaluated and it becomes the value of the overall conditional expression.
- (ii) If TestExpression is false(zero), then expression2 is evaluated and it becomes the value of overall conditional expression.

For example consider this conditional expression-

a > b ? a : b

Here first the expression a > b is evaluated, if the value is true then the value of variable a becomes the value of conditional expression otherwise the value of b becomes the value of conditional expression.

Suppose a = 5 and b = 8, and we use the above conditional expression in a statement as-

max = a > b ? a : b;

First the expression a > b is evaluated, since it is false so the value of b becomes the value of conditional expression and it is assigned to variable max.

In our next example we have written a conditional statement by putting a semicolon after the conditional expression.

a < b ? printf("a is smaller") : printf("b is smaller");

Since the expression a < b is true, so the first printf function is executed.

```
/*P4.6 Program to print the larger of two numbers using conditional operator
*/
#include<stdio.h>
main()
{
    int a,b,max;
    printf("Enter values for a and b : ");
    scanf("%d %d",&a,&b);
    max = a>b ? a:b;           /*ternary operator*/
    printf("Larger of %d and %d is %d\n",a,b,max);
```

Output:

Enter values for a and b : 12 7

Larger of 12 and 7 is 12

4.10 Comma Operator

The comma operator (,) is used to permit different expressions to appear in situations where only one expression would be used. The expressions are separated by the comma operator. The separated expressions are evaluated from left to right and the type and value of the rightmost expression is the type and value of the compound expression.

For example consider this expression-

```
a = 8, b = 7, c = 9, a+b+c
```

Here we have combined 4 expressions. Initially 8 is assigned to the variable a, then 7 is assigned to the variable b, 9 is assigned to variable c and after this a+b+c is evaluated which becomes the value of whole expression. So the value of the above expression is 24. Now consider this statement-

```
sum = ( a = 8, b = 7, c = 9, a+b+c );
```

Here the value of the whole expression on right side will be assigned to variable sum i.e. sum will be assigned value 24. Since precedence of comma operator is lower than that of assignment operator hence the parentheses are necessary here. The comma operator helps make the code more compact, for example without the use of comma operator the above task would have been done in 4 statements.

```
a = 8;  
b = 7;  
c = 9;  
sum = a+b+c ;
```

```
/*P4.7 Program to understand the use of comma operator */  
#include<stdio.h>  
main( )  
{  
    int a,b,c,sum;  
    sum = (a=8,b=7,c=9,a+b+c);  
    printf("Sum = %d\n",sum);  
}
```

Output:

Sum = 24

```
/* P4.8 Program to interchange the value of two variables using comm  
operator */  
#include<stdio.h>  
main( )  
{  
    int a=8,b=7,temp;  
    printf("a = %d, b = %d\n",a,b);  
    temp=a, a=b, b=temp;  
    printf("a = %d, b = %d\n",a,b);  
}
```

Output:

```
a = 8 , b = 7  
a = 7 , b = 8
```

The comma operator is also used in loops, we'll study about that in the next chapter.

4.11 sizeof Operator

sizeof is an unary operator. This operator gives the size of its operand in terms of bytes. The operand can be a variable, constant or any datatype(int, float, char etc). For example sizeof(int) gives the bytes occupied by the int datatype i.e. 2.

```
/*P4.9 Program to understand the sizeof operator*/
#include<stdio.h>
main()
{
    int var;
    printf ("Size of int = %d", sizeof(int));
    printf("Size of float = %d", sizeof(float));
    printf("Size of var = %d", sizeof(var));
    printf("Size of an integer constant = %d", sizeof(45));
}
```

Output:

Size of int = 2
 Size of float = 4
 Size of var = 2
 Size of an integer constant = 2

Generally sizeof operator is used to make portable programs i.e. programs that can be run on different machines. For example if we write our program assuming int to be of 2 bytes, then it won't run correctly on a machine on which int is of 4 bytes. So to make general code that can run on all machines we can use sizeof operator.

Bitwise Operators

C has the ability to support the manipulation of data at the bit level. Bitwise operators are used for operations on individual bits. Bitwise operators operate on integers only. The bitwise operators are as-

Bitwise operator	Meaning
&	bitwise AND
	bitwise OR
~	one's complement
<<	left shift
>>	right shift
^	bitwise XOR

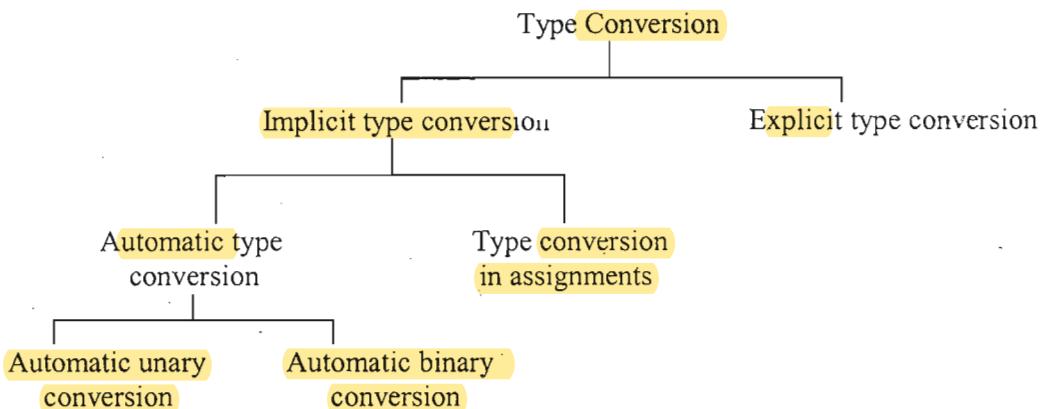
These bitwise operators are discussed in detail in chapter 13.

Other operators

There are many other operators that we'll study in further chapters. The last operator that we'll take up in this chapter is the type cast operator. Before studying that we'll discuss about type conversions in C.

4.12 Type Conversion

C provides the facility of mixing different types of variables and constants in an expression. In these types of operations data type of one operand is converted into data type of another operand. This is known as type conversion. The different types of type conversion are-



Implicit type conversions are done by the compiler while the explicit type conversions are user defined conversions.

4.12.1 Implicit Type Conversions

These conversions are done by the C compiler according to some predefined rules of C language. The two types of implicit type conversions are automatic type conversions and type conversion in assignments.

4.12.2 Automatic Conversions

Automatic unary conversions

All operands of type char and short will be converted to int before any operation. Some compilers convert all float operands to double before any operation.

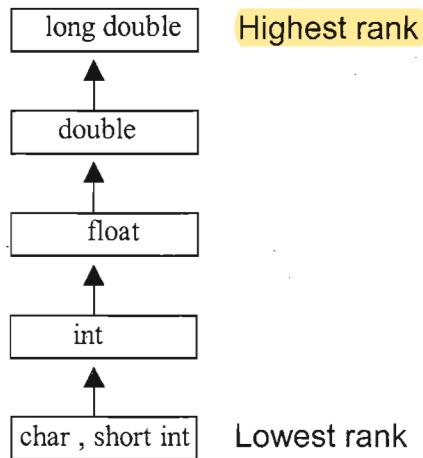
Automatic binary conversions

The rules for automatic binary conversions are as-

- (i) If one operand is long double, then the other will be converted to long double, and the result will be long double,
- (ii) Otherwise if one operand is double, then the other will be converted to double and the result will be double,
- (iii) Otherwise if one operand is float, the other will be converted to float and the result will be float,
- (iv) Otherwise if one operand is unsigned long int, then other will be converted to unsigned long int and the result will be unsigned long int.
- (v) Otherwise if one operand is long int and other is unsigned int
 - (a) If long int can represent all the values of an unsigned int, then unsigned int will be converted to long int and the result will be long int,
 - (b) Else both the operands will be converted to unsigned long int and the result will be unsigned long int,
- (vi) Otherwise if one operand is long int, then the other will be converted to long int and the result will be long int.

- (vii) Otherwise if one operand is unsigned int, then the other will be converted to unsigned int and the result will be unsigned int.
- (viii) Otherwise both operands will be int and the result will be int.

If we leave aside unsigned variables, then these rules are rather simple and can be summarized by assigning a rank to each data type. Whenever there are two operands of different data types the operand with a lower rank will be converted to the type of higher rank operand. This is called promotion of data type.



4.12.3 Type Conversion In Assignment

If the types of the two operands in an assignment expression are different, then the type of the right hand side operand is converted to the type of left hand operand. Here if the right hand operand is of lower rank then it will be promoted to the rank of left hand operand, and if it is of higher rank then it will be demoted to the rank of left hand operand.

Some consequences of these promotions and demotions are-

1. Some high order bits may be dropped when long is converted to int, or int is converted to short int or char.
2. Fractional part may be truncated during conversion of float type to int type.
3. When double type is converted to float type, digits are rounded off.
4. When a signed type is changed to unsigned type, the sign may be dropped.
5. When an int is converted to float, or float to double there will be no increase in accuracy or precision.

```

/*P4.10 Program to understand the type conversion in assignment*/
#include<stdio.h>
main( )
{
    char c1,c2;
    int i1,i2;
    float f1,f2;
    c1='H';
    i1=80.56; /*Demotion:float converted to int, only 80 assigned to i1*/
    f1=12.6;
}
  
```

```

c2=i1;      /*Demotion : int converted to char*/
i2=f1;      /*Demotion : float converted to int*/
/*Now c2 has character with ASCII value 80,i2 is assigned value
12*/
printf("c2 = %c, i2 = %d\n",c2,i2);
f2=i1;      /*Promotion : int converted to float*/
i2=c1;      /*Promotion : char converted to int*/
/*Now i2 contains ASCII value of character 'H' which is 72*/
printf("f2 = %.2f, i2 = %d\n",f2,i2);
}

```

Output:

c2 = P, i2 = 12
f2 = 80.00, i2 = 72

4.12.4 Explicit Type Conversion Or Type Casting

There may be certain situations where implicit conversions may not solve our purpose. For example-

```

float z;
int x = 20, y = 3;
z = x/y;

```

The value of z will be 6.0 instead of 6.66.

In these types of cases we can specify our own conversions known as type casting or coercion. This is done with the help of cast operator. The cast operator is a unary operator that is used for converting an expression to a particular data type temporarily. The expression can be any constant or variable.

The syntax of cast operator is-

(datatype) expression

Here the datatype along with the parentheses is called the **cast operator**.

So if we write the above statement as-

$z = (\text{float})x/y;$

Now the value of z will come out be 6.66. This happens because the cast operator (float) temporarily converted the int variable x into float type and so floating point arithmetic took place and fraction part was not lost.

Note that the cast operator changes the data type of variable x only temporarily for the evaluation of this expression, everywhere else in the program it will be an int variable only.

```

/*P4.11 Program to illustrate the use of cast operator*/
#include<stdio.h>
main()
{
    int x=5,y=2;
    float p,q;
    p=x/y;
    printf("p = %f\n",p);
    q=(float)x/y;
    printf("q = %f\n",q);
}

```

Output:

p = 2.000000

q = 2.500000

Initially the expression x/y is evaluated, both x and y are integers so according to integer arithmetic after division, decimal value is truncated and result is integer value 2. This value will be assigned to p, but p is a float variable so according to implicit type conversion in assignment the integer value 2 will be converted to float and then assigned to p. So finally the value of p is 2.0

When cast operator is used, floating point arithmetic is performed hence the value of q is 2.5

Here are some other examples of usage of cast operator-

(int)20.3	constant 20.3 converted to integer type and fractional part is lost(Result 20)
(float)20/3	constant 20 converted to float type, and then divided by 3 (Result 6.66)
(float)(20/3)	First 20 divided by 3 and then result of whole expression converted to float type(Result 6.00)
(double)(x +y -z)	Result of expression x+y-z is converted to double
(double)x+y-z	First x is converted to double and then used in expression

4.13 Precedence And Associativity Of Operators

For evaluation of expressions having more than one operator, there are certain precedence and associativity rules defined in C. Let us see what these rules are and why are they required.

Consider the following expression-

2 + 3 * 5

Here we have two operators- addition and multiplication operators. If addition is performed before multiplication then result will be 25 and if multiplication is performed before addition then the result will be 17.

In C language, operators are grouped together and each group is given a precedence level. The precedence of all the operators is given in the following table. The upper rows in the table have higher precedence and it decreases as we move down the table. Hence the operators with precedence level 1 have highest precedence and with precedence level 15 have lowest precedence. So whenever an expression contains more than one operator, the operator with a higher precedence is evaluated first. For example in the above expression, multiplication will be performed before addition since multiplication operator has higher precedence than the addition operator.

Operator	Description	Precedence level	Associativity
() [] → .	Function call Array subscript Arrow operator Dot operator	1	Left to Right
+ - ++ -- ! ~ * & (datatype) sizeof	Unary plus Unary minus Increment Decrement Logical NOT One's complement Indirection Address Type cast Size in bytes	2	Right to Left
*	Multiplication	3	Left to Right
/	Division		
%	Modulus		
+	Addition	4	Left to Right
-	Subtraction		
<<	Left shift	5	Left to Right
>>	Right shift		
<	Less than		
<=	Less than or equal to	6	Left to Right
>	Greater than		
>=	Greater than or equal to		
= = !=	Equal to Not equal to	7	Left to Right
&	Bitwise AND	8	Left to Right
^	Bitwise XOR	9	Left to Right
	Bitwise OR	10	Left to Right
&&	Logical AND	11	Left to Right
	Logical OR	12	Left to Right
? :	Conditional operator	13	Right to Left
*= /= %= += -= &= ^= = <=>=	Assignment operators	14	Right to Left
,	Comma operator	15	Left to Right

Let us take some expressions and see how they will be evaluated-

- (i) $x = a+b < c$

Here + operator has higher precedence than $<$ and $=$, and $<$ has more precedence than $=$, so first $a+b$ will be evaluated, then $<$ operator will be evaluated, and at last the whole value will be assigned to x . If initial values are $a = 2$, $b = 6$, $c = 9$, then final value of x will be 1.

- (ii) $x *= a + b$

Here + operator has higher precedence than $*$, so $a+b$ will be evaluated before compound assignment. This is interpreted as $x = x*(a+b)$ and not as $x = x*a+b$.

If initial values are $x = 5$, $a = 2$, $b = 6$, then final value of x will be 13.

- (iii) $x = a <= b \parallel b = = c$

Here order of evaluation of operators will be $<=$, $= =$, \parallel , $=$. If initial values are $a = 2$, $b = 3$, $c = 4$, then final value of x will be 1.

In the above examples we have considered expressions that contain operators having different precedence levels. Now consider a situation when two operators with the same precedence occur in an expression. For example-

$$5 + 16 / 2 * 4$$

Here / and * have higher precedence than + operator, so they will be evaluated before addition. But / and * have same precedence, so which one of them will be evaluated first still remains a problem. If / is evaluated before *, then the result is 37 otherwise the result is 7. Similarly consider this expression-

$$20 - 7 - 5 - 2 - 1$$

Here we have four subtraction operators, which of course have the same precedence level. If we decide to evaluate from left to right then answer will be 5, and if we evaluate from right to left the answer will be 17.

To solve these types of problems, an associativity property is assigned to each operator. Associativity of the operators within same group is same. All the operators either associate from left to right or from right to left. The associativity of all operators is also given in the precedence table. Now again consider the above two expressions-

$$5 + 16 / 2 * 4$$

Since / and * operators associate from left to right so / will be evaluated before * and the correct result is 37.

$$20 - 7 - 5 - 2 - 1$$

The subtraction operator associates from left to right so the value of this expression is 5.

The assignment operator associates from right to left. Suppose we have a multiple assignment expression like this-

$$x = y = z = 5$$

Here initially the integer value 5 is assigned to variable z and then value of expression $z = 5$ is assigned to variable y . The value of expression $z = 5$ is 5, so 5 is assigned to variable y and now the value of expression $y = z = 5$ becomes 5. Now value of this expression is assigned to x and the value of whole expression $x = y = z = 5$ becomes 5.

The conditional operator also associates from right to left. Consider this expression-

$$x ? y : a ? b : c$$

Since the conditional operator is right associative so this expression is considered as:

$x ? y : (a ? b : c)$

If x is non zero then the value of this expression will be equal to y . If x is zero then value of this expression will be equal to the value of expression $(a ? b : c)$ i.e. equal to b if a is non zero and equal to c if a is zero.

4.14 Role Of Parentheses In Evaluating Expressions

If we want to change the order of precedence of any operation, we can use parentheses. According to the parentheses rule, all the operations that are enclosed within parentheses are performed first.

For example in expression $24/2+4$, division will take place before addition according to precedence rule but if we enclose $2+4$ inside parentheses then addition will be performed first. So the value of expression $24/(2+4)$ is 4.

For evaluation of expression inside parentheses same precedence and associativity rules apply. For example-

$(22 - 4) / (2+4*2-1)$

Here inside parentheses multiplication will be performed before addition.

There can be nesting of parentheses in expressions i.e. a pair of parentheses can be enclosed with another pair of parentheses. For example-

$(4 * (3+2)) / 10$

In these cases, expressions within innermost parentheses are always evaluated first, and then next innermost parentheses and so on, till outermost parentheses. After evaluation of all expressions within parentheses, the remaining expression is evaluated as usual. In the above expression $3+2$ is evaluated first and then $4*5$ and then $20/10$.

Sometimes in complex expressions, parentheses are used just to increase readability. For example compare the following two expressions-

$x = a != b \ \&\& \ c * d >= m \% n$

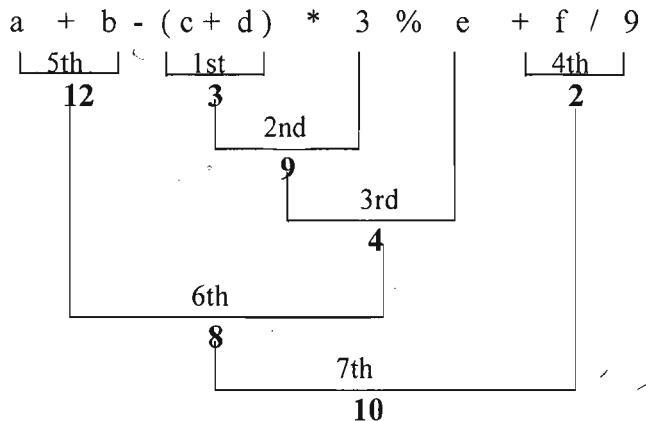
$x = (a != b) \ \&\& \ ((c * d) >= (m \% n))$

These two expressions are evaluated in the same way but the operations performed in second one are clearer.

Let us take some expressions and evaluate them according the precedence, associativity and parentheses rules.

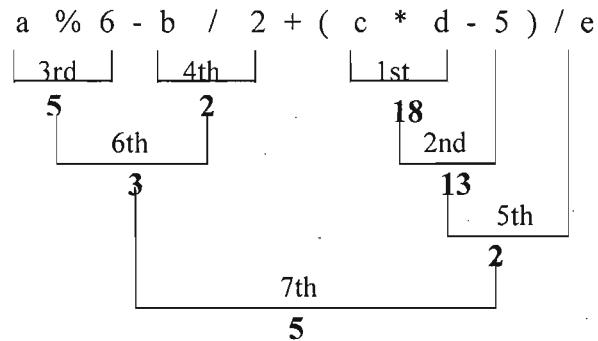
$a = 8, \ b = 4, \ c = 2, \ d = 1, \ e = 5, \ f = 20$

$a + b - (c + d) * 3 \% e + f / 9$



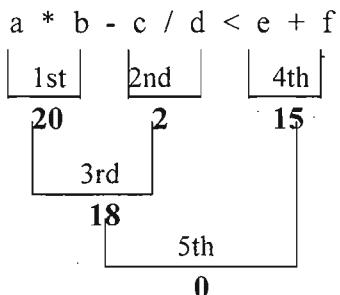
$$a = 17, b = 5, c = 6, d = 3, e = 5$$

$$a \% 6 - b / 2 + (c * d - 5) / e$$



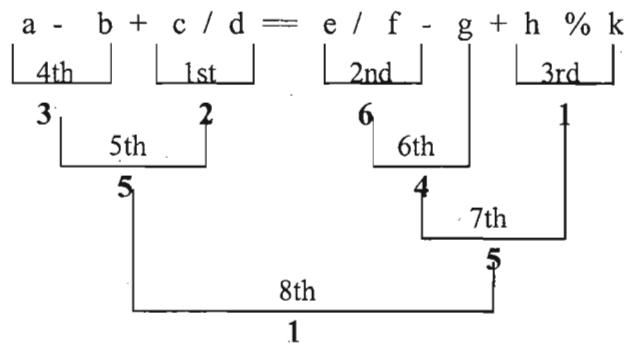
$$a = 4, b = 5, c = 6, d = 3, e = 5, f = 10$$

$$a * b - c / d < e + f$$



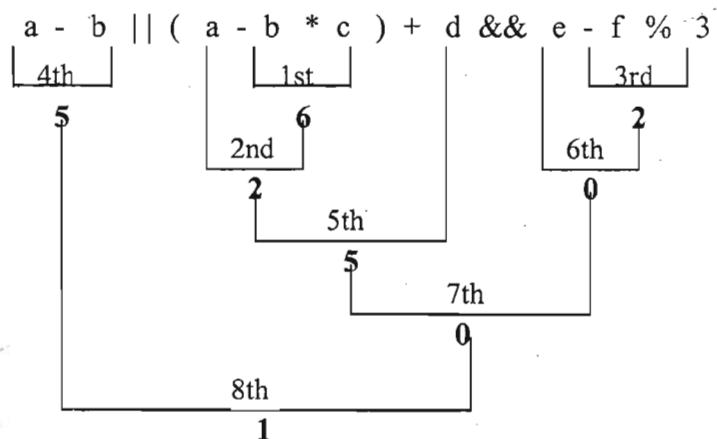
$$a = 8, b = 5, c = 8, d = 3, e = 65, f = 10, g = 2, h = 5, k = 2$$

$$a - b + c / d == e / f - g + h \% k$$



$$a = 8, b = 3, c = 2, d = 3, e = 2, f = 11$$

`a - b || (a - b * c) + d && e - f % 3`



The following program prints the results of above expressions-

```

/*P4,12 Program to evaluate some expressions*/
main()
{
    int a,b,c,d,e,f,g,h,k;
    a=8,b=4,c=2,d=1,e=5,f=20;
    printf("%d\t",a+b-(c+d)*3%e+f/9);
    a=17,b=5,c=6,d=3,e=5;
    printf("%d\t",a%6-b/2+(c*d-5)/e);
    a=4,b=5,c=6,d=3,e=5,f=10;
    printf("%d\t",a*b-c/d<e+f);
    a=8,b=5,c=8,d=3,e=65,f=10,g=2,h=5,k=2;
    printf("%d\t",a-b+c/d==e/f-g+h%k);
    a=8,b=3,c=2,d=3,e=2,f=11;
    printf("%d\n",a-b|| (a-b*c)+d&&e-f%3);
}
  
```

Output:

10 5 0 1 1

4.15 Order Of Evaluation Of Operands

C does not specify the order in which the operands of an operator are evaluated. For example consider the expression-

```
y = (++x) + (- -x);           /*Assume value of x is 5*/
```

Here the two operands of the addition operator are $(++x)$ and $(- -x)$. These two operands have to be added and then the result has to be assigned to variable y . Before addition, these two operands will be evaluated. Now if the first operand $(++x)$ is evaluated first then y will be assigned value 11, while if second operand $(- -x)$ is evaluated first then y will be assigned the value 9. Since C is silent on such controversies, the answers may vary on different compilers so it is better to avoid such type of expressions.

There are four exceptional operators where C clearly specifies the order of evaluation of operands. These operators are logical AND(`&&`), logical OR(`||`), conditional `(? :)` and comma operator(`,`). In all these cases the operand on the left side is evaluated first.

In the case of logical AND and logical OR operators, sometimes there is no need to evaluate second operand. In `&&` operator, if the first operand evaluates to false(zero), then second operand is not evaluated and in the case of `||` operator if the first operand evaluates to true(non zero) then second operand is not evaluated.

Exercise

Assume stdio.h is included in all programs.

```
(1) main( )
{
    int a=-3;
    a=-a-a+!a;
    printf("%d\n",a);
}

(2) main( )
{
    int a=2,b=1,c,d;
    c=a<b;
    d=(a>b)&&(c<b);
    printf("c = %d, d = %d\n",c,d);
}

(3) main( )
{
    int a=9,b=15,c=16,d=12,e,f;
    e=! (a<b || b<c);
    f=(a>b) ? a-b:b-a;
    printf("e = %d, f = %d\n",e,f );
}

(4) main( )
{
    int a=5;
    a=6;
```

```

    a=a+5*a;
    printf("a = %d\n", a);
}

(5) main()
{
    int a=5,b=5;
    printf("%d, %d\t", ++a, b--);
    printf("%d, %d\t", a, b);
    printf("%d, %d\t", ++a, b++);
    printf("%d, %d\n", a, b);
}

(6) main()
{
    int x,y,z;
    x=8++;
    y=++x++;
    z=(x+y)--;
    printf("x = %d, y = %d, z = %d\n", x, y);
}

(7) main()
{
    int a=4,b=8,c=3,d=9,z;
    z=a++ + ++b * c-- - --d;
    printf("a = %d, b = %d, c = %d, d = %d, z = %d\n", a, b, c, d, z);
}

(8) main()
{
    int a=14,b,c;
    a=a%5;
    b=a/3;
    c=a/5%3;
    printf("a = %d, b = %d, c = %d\n", a, b, c);
}

(9) main()
{
    int a=15,b=13,c=16,x,y;
    x=a-3%2+c*2/4%2+b/4;
    y=a=b+5-b+9/3;
    printf("x = %d, y = %d\n", x, y);
}

(10) main()
{
    int x,y,z,k=10;
    k+=(x=5,y=x+2,z=x+y);
    printf("x = %d, y = %d, z = %d, k = %d\n", x, y, z, k);
}

```

```
(11)main( )
{
    int a;
    float b;
    b=15/2;
    printf("%f\t",b);
    b=(float)15/2+(15/2);
    printf("%f\n",b);
}

(12)main( )
{
    int a=9;
    char ch='A';
    a=a+ch+24;
    printf("%d,%c\t%d,%c\n",ch,ch,a,a);
}

(13)main( )
{
    int a,b,c,d;
    a=b=c=d=4;
    a*=b+1;
    c+=d*=3;
    printf("a = %d,c = %d\n",a, c);
}

(14)main( )
{
    int a=5,b=10,temp;
    temp=a,a=b,b=temp;
    printf("a = %d,b = %d\n",a,b);
}

(15)main( )
{
    int a=10,b=3,max;
    a>b?max=a:max=b;
    printf("%d",max);
}

(16)#include<stdio.h>
main()
{
    int a=5,b=6;
    printf("%d\t",a=b);
    printf("%d\t",a==b);
    printf("%d %d\n",a,b);
}

(17)main()
```

```

{
    int a=3,b=4,c=3,d=4,x,y;
    x=(a=5)&&(b=7);
    y=(c=5) || (d=8);
    printf("a=%d,b=%d,c=%d,d=%d,x=%d,y=%d\n",a,b,c,d,x,y);
    x=(a==6)&&(b=9);
    y=(c==6) || (d=10);
    printf("a=%d,b=%d,c=%d,d=%d,x=%d,y=%d\n",a,b,c,d,x,y);
}

(18)main()
{
    int a=10;
    a=a++*a--;
    printf("%d\n",a);
}

(19)main()
{
    int a=2,b=2,x,y;
    x=4*(++a*2+3);
    y=4*(b++*2+3);
    printf("a=%d,b=%d,x=%d,y=%d\n",a,b,x,y);
}

```

Programming Exercise

1. Enter the temperature in Celsius and convert that into Fahrenheit.
2. Accept the radius of the circle and calculate the area and perimeter of the circle.
3. Write a program to accept the number in decimal and print the number in octal and hexadecimal.
4. Accept any five digit number and print the value of remainder after dividing by 3.
5. Accept any two numbers, if the first number is greater than second then print the sum of these two numbers, otherwise print their difference. Write this program using ternary operator.
6. Write a program to accept the principal, rate, and number of years and find out the simple interest.
7. Accepts marks in five subject and calculate the total percentage marks.

Answers

- (1) 6
- (2) c = 0, d = 1
- (3) e = 0, f = 6
- (4) 36
- (5) 6, 5 6, 4 7, 4 7, 5
- (6) Error, Expressions like 8++, ++x++, (x+y)-- are not valid.
- (7) a = 5, b = 9, c = 2, d = 8, z = 23
- (8) a = 4, b = 1, c = 0
- (9) x = 17, y = 8
- (10) x = 5, y = 7, z = 12, k = 22

(11) 7.000000 14.500000

(12) 65, A 98, b

(13) a = 20, c = 16

Precedence of + operator is more than that of +=. Associativity of compound assignment operators is from right to left

(14) a = 10, b = 5

(15) Since the precedence of assignment operator is less than that of conditional operator so the conditional statement is interpreted as- (a>b?max = a : max) = b;

Now this statement reduces to statement 10 = b, which results in an error. So the solution is to put parentheses around the third operand as- a>b? max = a : (max = b);

(16) 6 1 6 6

(17) a = 5, b = 7, c = 5, d = 4, x = 1, y = 1

a = 5, b = 7, c = 5, d = 10, x = 0, y = 1

(18) Result of these types of expressions is undefined.

(19) a = 3, b = 3, x = 36, y = 28

Chapter 5

Control Statements

In C programs, statements are executed sequentially in the order in which they appear in the program. But sometimes we may want to use a condition for executing only a part of program. Also many situations arise where we may want to execute some statements several times. Control statements enable us to specify the order in which the various instructions in the program are to be executed. This determines the flow of control. Control statements define how the control is transferred to other parts of the program. C language supports four types of control statements, which are as-

1. if...else
2. goto
3. switch
4. loop
 - while
 - do...while
 - for

5.1 Compound Statement or Block

A compound statement or a block is a group of statements enclosed within a pair of curly braces {}. The statements inside the block are executed sequentially. The general form is-

```
{  
    statement1;  
    statement2;  
    .....  
    .....  
}
```

For example-

```
{  
    l=4;  
    b=2;  
    area=l*b;  
    printf("%d",area);  
}
```

A compound statement is syntactically equivalent to a single statement and can appear anywhere in the program where a single statement is allowed.

5.2 if...else

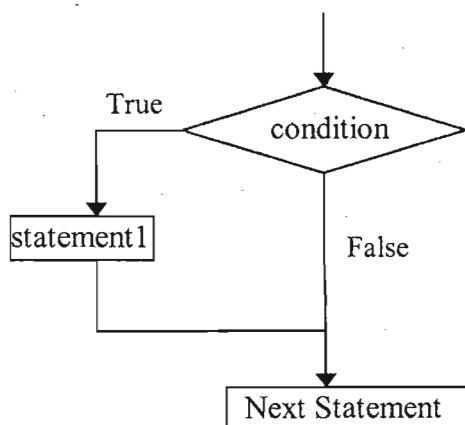
This is a bi-directional conditional control statement. This statement is used to test a condition and take one of the two possible actions. If the condition is true then a single statement or a block of statements is executed (one part of the program), otherwise another single statement or a block of statements is executed (other part of the program). Recall that in C, any nonzero value is regarded as true while zero is regarded as false.

Syntax 1:

```
if(condition)
    statement1;
```

```
if(condition)
{
    statement;
    .....
}
```

There can be a single statement or a block of statements after the if part.



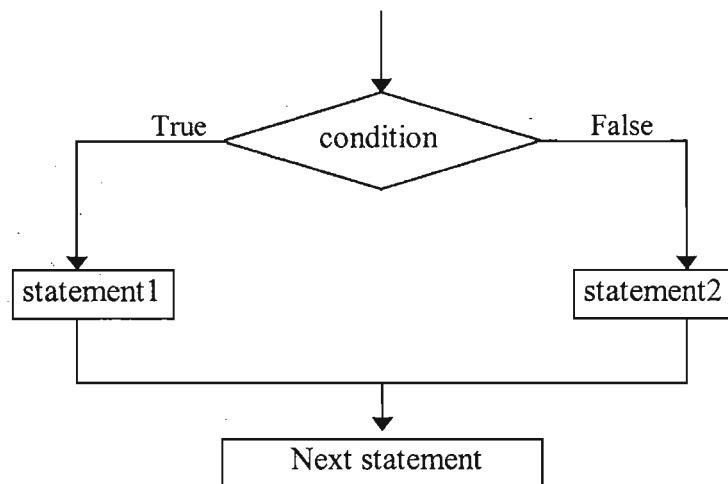
Flow chart of if control statement

Here if the condition is true(nonzero) then statement1 is executed, and if it is false(zero), then the next statement which is immediately after the if control statement is executed.

Syntax 2:

```
if(condition)
    statement1;
else
    statement2;
```

```
if(condition)
{
    statement;
    .....
}
else
{
    statement;
    .....
}
```



Flow chart of if...else control statement

Here if the condition is true then statement1 is executed and if it is false then statement2 is executed. After this the control transfers to the next statement which is immediately after the if...else control statement.

```
/*P5.1 Program to print a message if negative number is entered*/
#include<stdio.h>
main()
{
    int num;
    printf("Enter a number : ");
    scanf("%d",&num);
    if(num<0)
        printf("Number entered is negative\n");
    printf("Value of num is : %d\n", num);
}
```

1st run:

Enter a number : -6

Number entered is negative

Value of num is -6

2nd run:

Enter a number : 8

Value of num is 8

```
/*P5.2 Program to print the larger and smaller of the two numbers*/
#include<stdio.h>
main()
{
    int a,b;
    printf("Enter the first number : ");
    scanf("%d",&a);
```

```

printf("Enter the second number : ");
scanf("%d",&b);
if(a>b)
    printf("larger number = %d and smaller number = %d\n",a,b);
else
    printf("larger number = %d and smaller number = %d\n",b,a);
}

```

Output:

Enter the first number : 9

Enter the second number : 11

larger number = 11 and smaller number = 9

```

/*P5.3 Program to print whether the number is even or odd */
#include<stdio.h>
main()
{
    int num;
    printf("Enter a number : ");
    scanf("%d",&num);
    if(num%2==0)           /*test for even */
        printf("Number is even\n");
    else
        printf("Number is odd\n");
}

```

Output:

Enter the number : 15

Number is odd

5.2.1 Nesting of if...else

We can have another if...else statement in the if block or the else block. This is called nesting of if...else statements. Here is an example of nesting where we have if...else inside both if block and else block.

```

if(condition 1)
{
    if(condition 2)
        statementA1;
    else
        statementA2;
}
else
{
    if(condition 3)
        statementB1;
    else
        statementB2 ;
}

```

While nesting if...else statements, sometimes confusion may arise in associating else part with appropriate if part. Let us take an example-

```

if(grade=='A')
{
    if(marks>95)
        printf("Excellent");
}
else
    printf("Work hard for getting A grade");

```

If we write the above code without braces as-

```

if(grade=='A')
    if(marks>95)
        printf("Excellent");
else
    printf("Work hard for getting A grade");

```

Here the else part is matched with the second if, but we wanted to match it with the first if. The compiler does not associate if and else parts according to the indentations, it matches the else part with closest unmatched if part. So whenever there is doubt regarding matching of if and else parts we can use braces to enclose each if and else blocks.

```

/*P5.4 Program to find largest number from three given numbers*/
#include<stdio.h>
main()
{
    int a,b,c,large;
    printf("Enter three numbers : ");
    scanf("%d%d%d",&a,&b,&c);
    if(a>b)
    {
        if(a>c)
            large=a;
        else
            large=c;
    }
    else
    {
        if(b>c)
            large=b;
        else
            large=c;
    }
    printf("Largest number is %d\n",large);
}/*End of main()*/

```

The next program finds whether a given year is leap or not. A centennial(divisible by 100) year is leap if it is divisible by 400, and a non centennial year is leap if it is divisible by 4.

```

/*P5.5 Program to find whether a year is leap or not*/
#include<stdio.h>
main()
{

```

```

int year;
printf("Enter year : ");
scanf("%d",&year);
if(year%100==0)
{
    if(year%400==0)
        printf("Leap year\n");
    else
        printf("Not leap\n");
}
else
{
    if(year%4==0)
        printf("Leap year\n");
    else
        printf("Not leap\n");
}

```

Note that we can write this program using a single if condition and && and || operators.

```

if(year%4==0 && year%100!=0 || year%400==0)
    printf("Leap year\n");
else
    printf("Not leap\n");

```

5.2.2 else if Ladder

This is a type of nesting in which there is an if...else statement in every else part except the last else part. This type of nesting is frequently used in programs and is also known as else if ladder.

```

if( condition1)
    statementA;
else
    if( condition2)
        statementB;
    else
        if( condition 3)
            statementC;
        else
            statementD;

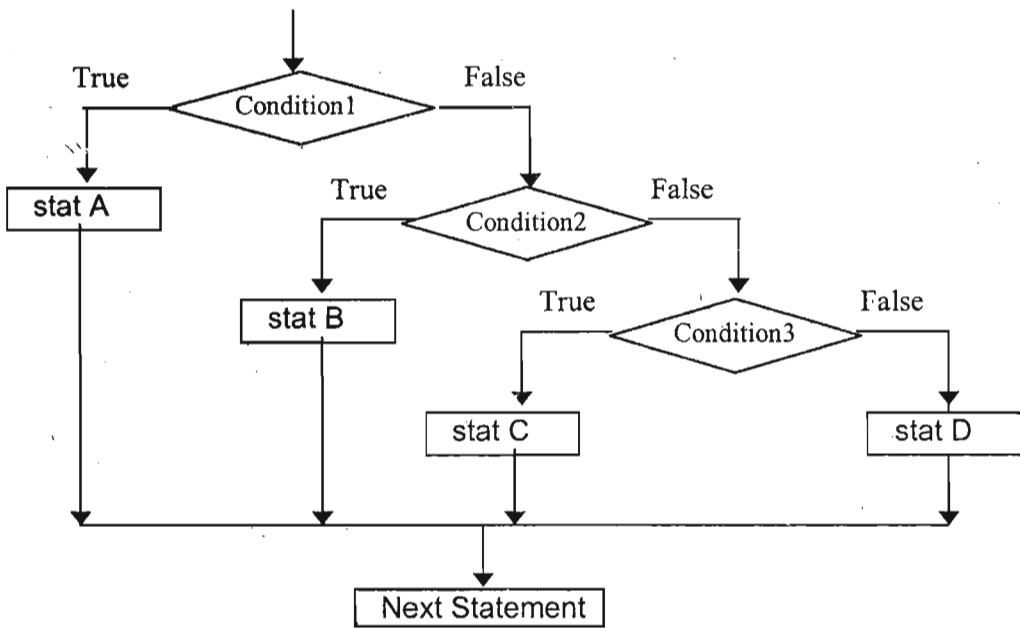
```

```

if( condition1)
    statementA;
else if( condition2)
    statementB;
else if( condition 3)
    statementC;
else
    statementD;

```

This nested structure is generally written in compact form as in second figure. The flow chart for this structure is-



Flow chart of else...if ladder

Here each condition is checked, and when a condition is found to be true, the statements corresponding to that are executed, and the control comes out of the nested structure without checking remaining conditions. If none of the conditions is true then the last else part is executed.

```

/*P5.6 Program to find out the grade of a student when the marks of
4 subjects are given. The method of assigning grade is as-
per>= 85           grade=A
per<85 and per>=70   grade=B
per<70 and per>=55   grade=C
per<55 and per>=40   grade=D
per<40              grade=E
Here per is percentage.
*/
#include<stdio.h>
main()
{
    float m1,m2,m3,m4,total,per;
    char grade;
    printf("Enter marks of 4 subjects : ");
    scanf("%f%f%f%f",&m1,&m2,&m3,&m4);
    total=m1+m2+m3+m4;
    per=total/4;
    if(per>=85)
        grade='A';
    else if(per>=70)
        grade='B';
    else if(per>=55)
        grade='C';
  
```

```

else if(per>=40)
    grade='D';
else
    grade='E';
printf("Percentage is %f\nGrade is %c\n",per,grade);
}

```

If we don't use the else if ladder, the equivalent code for this problem would be-

```

if(per>=85)
    grade='A';
if(per<85&&per>=70)
    grade='B';
if(per<70&&per>=55)
    grade='C';
if(per<55&&per>=40)
    grade='D';
if(per<40)
    grade='E';

```

In if...else ladder whenever a condition is found true other conditions are not checked, while in this case all the conditions will always be checked wasting a lot of time, and moreover the conditions here are more lengthy.

5.3 Loops

Loops are used when we want to execute a part of the program or a block of statements several times. For example, suppose we want to print "C is the best" 10 times. One way to get the desired output is - we write 10 printf statements, which is not preferable. Other way out is - use loop. Using loop we can write one loop statement and only one printf statement, and this approach is definitely better than the first one. With the help of loop we can execute a part of the program repeatedly till some condition is true. There are three loop statements in C-

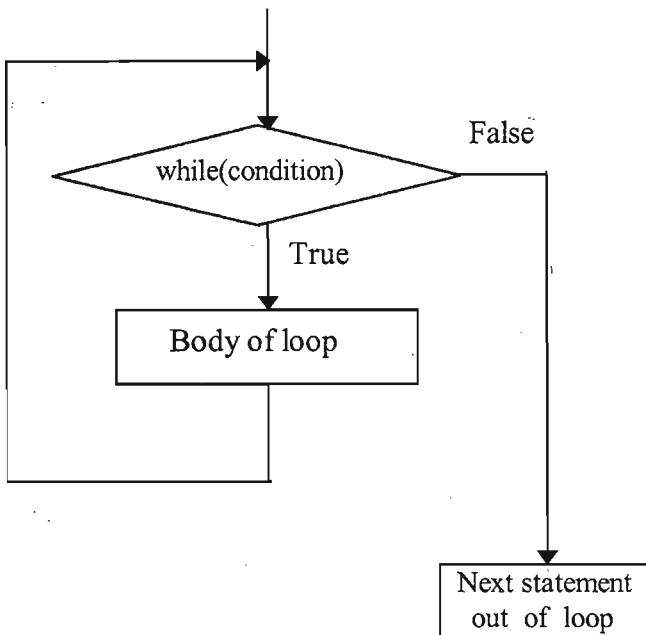
- (i) while
- (ii) do while
- (iii) for

5.3.1 while loop

The while statement can be written as:

```
while(condition)
    statement;
```

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



Flow chart of while loop

Like if-else statement here also we can have either a single statement or a block of statements, a here it is known as the body of loop. Now let's see how this loop works.

First the condition is evaluated; if it is true then the statements in the body of loop are executed. Af the execution, again the condition is checked and if it is found to be true then again the stateme in the body of loop are executed. This means that these statements are executed continuously till t condition is true and when it becomes false, the loop terminates and the control comes out of the lo Each execution of the loop body is known as iteration.

```

/*
P5.7 Program to print the numbers from 1 to 10 using while loop
#include<stdio.h>
main()
{
    int i=1;
    while(i<=10)
    {
        printf("%d\t",i);
        i=i+1; /*Statement that changes the value of condition */
    }
    printf("\n");
}
  
```

Output:

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

Here initially the condition ($i \leq 10$) is true. After each iteration of the loop, value of i increases : when the value of i equals 11 the condition becomes false and the loop terminates.

Note that inside the body of the loop there should be a statement that alters the value of the condition, so that the condition becomes false ultimately at some point.

```
/*P5.8 Program to print numbers in reverse order with a difference of 2*/
main( )
{
    int k=10;
    while(k>=2)
    {
        printf("%d\t",k);
        k=k-2;
    }
    printf("\n");
}
```

Output:

10 8 6 4 2

```
/*P5.9 Program to print the sum of digits of any number*/
#include<stdio.h>
main( )
{
    int n,sum=0,rem;
    printf("Enter the number : ");
    scanf("%d", &n);
    while(n>0)
    {
        rem=n%10; /* taking last digit of number */
        sum+=rem;
        n/=10; /* skipping last digit */
    }
    printf("Sum of digits = %d\n",sum);
}
```

Output:

Enter the number : 1452

Sum of digits = 12

Here we are extracting the digits of the number from right to left and then these digits are added one by one to the variable sum. Note that the variable sum is initialized to 0. This is because we are adding some numbers to it, and if not initialized then these numbers will be added to garbage value present in it. Let's see how the loop works when the value of n is 1452.

Before loop starts	rem = garbage value,	sum = 0,	n = 1452
After 1st iteration	rem = 1452%10 = 2,	sum = 0+2 = 2,	n = 145
After 2nd iteration	rem = 145%10 = 5,	sum = 2+5 = 7,	n = 14
After 3rd iteration	rem = 14%10 = 4,	sum = 7+4 = 11,	n = 1
After 4th iteration	rem = 1%10 = 1,	sum = 11+1 = 12,	n = 0

Now since the value of n is equal to zero, hence the condition (n > 0) becomes false and the loop stops.

```
/*P5.10 Program to find the product of digits of any number*/
#include<stdio.h>
main()
{
    int n,prod=1,rem;
    printf("Enter the number : ");
    scanf("%d",&n);
    while(n>0)
    {
        rem=n%10;           /*taking last digit*/
        prod*=rem;
        n/=10;              /*skipping last digit of number*/
    }
    printf("Product of digits = %d\n",prod);
}
```

Output :

Enter the number : 234

Product of digits = 24

The logic of extracting digits is similar to that in previous program, but here we are multiplying digits instead of adding, so here the variable prod is initialized by value 1.

```
/*P5.11 Program to find the factorial of any number*/
#include<stdio.h>
main()
{
    int n,num;
    long fact=1;
    printf("Enter the number : ");
    scanf("%d",&n);
    num=n;
    if(n<0)
        printf("No factorial of negative number\n");
    else
    {
        while(n>1)
        {
            fact*=n;
            n--;
        }
        printf("Factorial of %d = %ld\n", num, fact);
    }
}
```

Output:

Enter the number : 4

Factorial of 4 = 24

The factorial of a number n is the product of numbers from 1 to n.

$$n! = n * (n-1) * (n-2) \dots * 2 * 1$$

$$4! = 4 * 3 * 2 * 1 = 24$$

$$8! = 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1 = 40320$$

We have taken the variable fact of type long instead of type int. This is so because the factorials grow at a very fast rate, and they may exceed the range of an int (32767). Even the factorial of 8 is more than this limit.

```
/*P5.12 Program to convert a binary number to a decimal number*/
#include<stdio.h>
main()
{
    int n, nsave, rem, d, j=1, dec=0;
    printf("Enter the number in binary : ");
    scanf("%d", &n);
    nsave=n;
    while(n>0)
    {
        rem=n%10;           /*taking last digit*/
        d=rem*j;
        dec+=d;
        j*=2;
        n/=10;             /*skipping last digit*/
    }
    printf("Binary number = %d, Decimal number = %d\n", nsave, dec);
}
```

Ouput :

Enter the number in binary : 1101

Binary number = 1101, Decimal number = 13

To convert a binary number to decimal, we extract binary digits from right and add them after multiplying by powers of 2. This is somewhat similar to the program P5.9, only we have to multiply the digits by powers of 2 before adding them. This is how the loop works for binary number 1101.

Before loop starts:	rem=garbage,	d= garbage,	dec=0,	j=1,	n=nsave=1101
After 1st iteration:	rem=1,	d=1*1,	dec=1,	j=2,	n=110
After 2nd iteration:	rem=0,	d=0*2,	dec=1+0,	j=4,	n=11
After 3rd iteration:	rem=1,	d=1*4,	dec=1+0+4,	j=8,	n=1
After 4th iteration:	rem=1,	d=1*8,	dec=1+0+4+8,	j=16,	n=0

Now the value of n becomes zero so the loop terminates. We have taken a variable nsave to save the value of binary number, because the value of n gets changed after the loop execution.

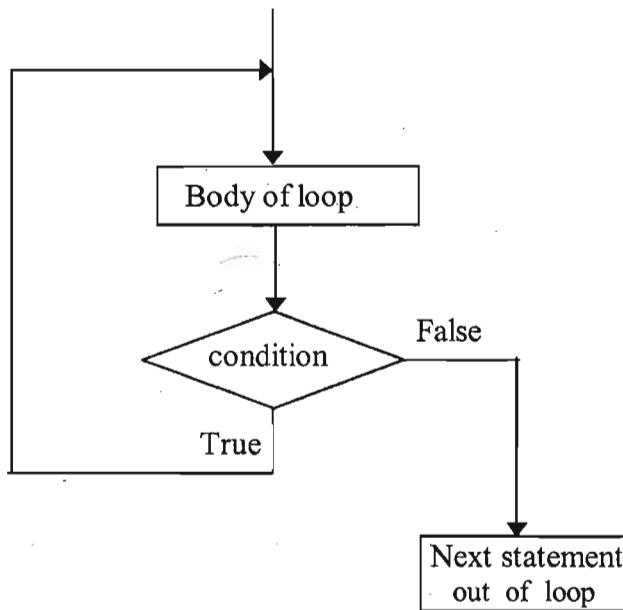
(The above program will be valid for binary numbers upto 11111 only, for larger numbers take long int).

5.3.2 do...while loop

The 'do...while' statement is also used for looping. The body of this loop may contain a single statement or a block of statements. The syntax for writing this loop is:

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

	}while(condition);



Flow chart of do ... while loop

Here firstly the statements inside loop body are executed and then the condition is evaluated. If the condition is true, then again the loop body is executed and this process continues until the condition becomes false. Note that unlike while loop, here a semicolon is placed after the condition.

In a 'while' loop, first the condition is evaluated and then the statements are executed whereas in a do while loop, first the statements are executed and then the condition is evaluated. So if initially the condition is false the while loop will not execute at all, whereas the do while loop will always execute at least once.

```

/*P5.13 Program to print the numbers from 1 to 10 using do...while loop*/
#include<stdio.h>
main()
{
    int i=1;
    do
    {
        printf("%d\t",i);
        i=i+1;
    }while(i<=10);
    printf("\n");
}
  
```

Output:

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

Generally while loop is used more frequently than the do while loop but some situations may arise where it is better to check the condition at the bottom of loop. For example in the next two programs it is better to use a do while loop.

```
/*P5.14 Program to count the digits in any number */
#include<stdio.h>
main()
{
    int n, count=0, rem;
    printf("Enter the number : ");
    scanf("%d", &n);
    do
    {
        n/=10;
        count++;
    }while(n>0);
    printf("Number of digits = %d\n", count);
}
```

If we write the same program using a while loop then we won't get correct answer when input number is 0. The count of digits in number 0 is 1, but using while loop the answer will come out to be zero.

```
/* P5.15 Program to find the sum of numbers entered*/
#include<stdio.h>
main( )
{
    int num, sum=0;
    do
    {
        printf("Enter a number (0 to stop) : ");
        scanf("%d", &num);
        sum+=num;
    }while(num!=0);
    printf("Sum is %d\n", sum);
}
```

In this program if we use while loop then the condition will be checked at the top, so either we'll have to give some initial value to the variable num or we'll have to write duplicate printf and scanf() statements before the loop.

The do...while loop is generally used for checking validity of entered data. Suppose we want the user to enter an employee ID, and the valid IDs are in the range 100 to 500 only. If the user enters an invalid ID, we want to ignore that ID and again ask him to enter another one, and we want this process to continue till he enters a valid ID. In this type of situation we can use do...while loop.

```
do
{
    printf("Enter employee ID : ");
    scanf("%d", &emp_id);
}while(emp_id<100 || emp_id>500);
```

This loop will terminate only when the entered number is in the valid range 100 to 500.

5.3.3 for loop

The 'for' statement is very useful while programming in C. It has three expressions and semicolons are used for separating these expressions. The 'for' statement can be written as-

```

for(expression1;expression2;expression3)
    statement;
for(expression1;expression2;expression3)
{
    statement;
    statement;
    .....
}

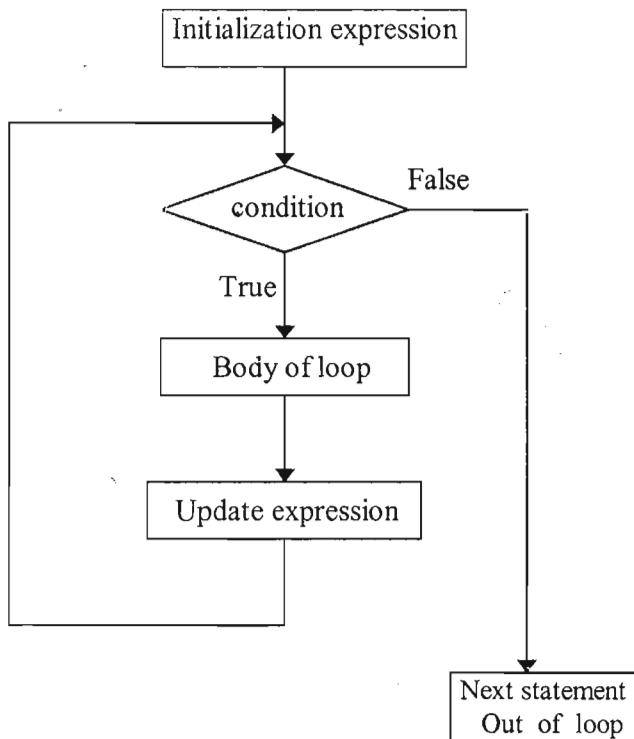
```

The loop body can be a single statement or block of statements.

expression1 is an initialization expression, expression2 is a test expression or condition and expression3 is an update expression. expression1 is executed only once when the loop starts and is used to initialize the loop variables. This expression is generally an assignment expression. expression2 is a condition and is tested before each iteration of the loop. This condition generally uses relational and logical operators. expression3 is an update expression and is executed each time after the body of the loop is executed.

Now let us see how this loop works. Firstly the initialization expression is executed and the loop variables are initialized, and then the condition is checked, if the condition is true then the body of loop is executed. After executing the loop body, control transfers to expression3(update expression) and it modifies the loop variables and then again the condition is checked , and if it is true, the body of loop is executed. This process continues till the condition is true and when the condition becomes false the loop is terminated and control is transferred to the statement following the loop.

The work done by the for loop can be performed by writing a while loop as-



Flow chart of for loop

```

expression 1;
while(expression 2)
{
    statement;
    .....
    .....
    expression 3;
}

```

Although the task of while and for loops is same, the for loop is generally used when the number of iterations are known in advance and while loop is used where number of iterations are not known.

```

/*P5.16 Program to print the numbers from 1 to 10 using for loop*/
#include<stdio.h>
main()
{
    int i;
    for(i=1;i<=10;i++)
        printf("%d\t",i);
    printf("\n");
}

```

Output:

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

```

/*P5.17 Program to print numbers in reverse order with a difference of
2*/
main( )
{
    int k;
    for(k=10;k>=2;k-=2)
        printf("%d\t",k);
    printf("\n");
}

```

Output:

10	8	6	4	2
----	---	---	---	---

 P5.18 Multiply two positive numbers without using * operator*

```

#include<stdio.h>
main( )
{
    int a,b,i;
    int result=0;
    printf("Enter two numbers to be multiplied : ");
    scanf("%d%d",&a,&b);
    for(i=1;i<=b;i++)
        result=result+a;
    printf("%d * %d = %d\n",a,b,result);
}

```

*P5.19 Find the sum of this series upto n terms

```

1+2+4+7+11+16+..... */
#include<stdio.h>
main()
{
    int i,n,sum=0,term=1;
    printf("Enter number of terms : ");
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    {
        sum+=term;
        term=term+i;
    }
    printf("The sum of series upto %d terms is %d\n",n,sum);
}

/*P5.20 Program to generate fibonacci series
1,1,2,3,5,8,13,34,55,89.....
In this series each number is a sum of the previous two numbers*/
#include<stdio.h>
main()
{
    long x,y,z;
    int i,n;
    x=0;
    y=1;
    printf("Enter the number of terms : ");
    scanf("%d",&n);
    printf("%ld ",y);
    for(i=1;i<n;i++)
    {
        z=x+y;
        printf("%ld ",z);
        x=y;
        y=z;
    }
    printf("\n");
}

```

All the three expressions of the for loop are optional. We can omit any one or all the three expressions in the for loop but in any case the two separating semicolons should always be present.

expression1 is omitted when the initialization work is done before entering the loop.

expression2 is a condition and if omitted, it is always assumed to be true and so this type of loop will never stop executing. This type of loop is infinite and to avoid it there should be a statement inside the loop that takes the control out of the loop.

expression3 is an update expression and is omitted when it is present inside the body of the loop.

```

/*P5.21 Program to print the sum of digits of any number using for loop
*/
#include<stdio.h>

```

```

main()
{
    int n,sum=0,rem;
    printf("Enter the number : ");
    scanf("%d",&n);
    for( ;n>0;n/=10)
    {
        rem=n%10; /*taking last digit of number*/
        sum+=rem;
    }
    printf("Sum of digits = %d\n",sum);
}

```

We can also have any number of expressions separated by commas. For example, we may want to initialize more than one variable or take more than one variable as loop variable.

```

/*P5.22 Program to print numbers using for loop*/
#include<stdio.h>
main()
{
    int i,j;
    for(i=0,j=10;i<=j;i++,j--)
        printf("i = %d j = %d\n",i,j);
}

```

Output:

```

i = 0 j = 10
i = 1 j = 9
i = 2 j = 8
i = 3 j = 7
i = 4 j = 6
i = 5 j = 5

```

5.3.4 Nesting of Loops

When a loop is written inside the body of another loop, then it is known as nesting of loops. Any type of loop can be nested inside any other type of loop. For example a for loop may be nested inside another for loop or inside a while or do while loop. Similarly while and do while loops can be nested.

```

/*P5.23 Program to understand nesting in for loop*/
#include<stdio.h>
main()
{
    int i,j;
    for(i=1;i<=3;i++)           /*outer loop*/
    {
        printf("i = %d\n",i);
        for(j=1;j<=4;j++)       /*inner loop*/
            printf("j = %d\t",j);
        printf("\n");
    }
}

```

Output:

```
i = 1
j = 1 j = 2 j = 3 j = 4
i = 2
j = 1 j = 2 j = 3 j = 4
i = 3
j = 1 j = 2 j = 3 j = 4
```

Here for each iteration of the outer loop, the inner loop is executed 4 times.

The next program prints armstrong numbers. Armstrong number is a three digit number in which the sum of cube of all digits is equal to the number, for example 371 is an armstrong number since $3^3+7^3+1^3=27+343+1=371$

```
/*P5.24 Program to print armstrong numbers from 100 to 999*/
#include<stdio.h>
main()
{
    int num,n,cube,d,sum;
    printf("Armstrong numbers are :\n");

    for(num=100;num<=999;num++)      /*outer loop*/
    {
        n=num;
        sum=0;
        while(n>0)          /*inner loop*/
        {
            d=n%10;
            n/=10;
            cube=d*d*d;
            sum=sum+cube;
        }/*End of while loop*/
        if(num==sum)
            printf("%d\n",num);
    }/*End of for loop*/
}
```

Here the outer for loop is used to generate numbers from 100 to 999, and the inner while loop is used to extract digits and then find the sum of cube of those digits.

```
/*P5.25 Program to find the sum of digits of a number until the sum
is reduced to 1 digit.
For example: 538769->38->11->2*/
#include<stdio.h>
main()
{
    long num;
    int dig,sum;
    printf("Enter a number : ");
    scanf("%ld",&num);
    printf("%ld->",num);
```

```

do
{
    for(sum=0;num!=0;num/=10)
    {
        dig=num%10;
        sum+=dig;
    }
    printf("%d\t",sum);
    num=sum;
}while(num/10!=0);
printf("\n");
}

```

Output:

Enter a number: 789988

789988->49 13 4

Here the inner for loop is used to find the digits of the number.

5.3.5 Infinite Loops

The loops that go on executing infinitely and never terminate are called infinite loops. Sometimes we write these loops by mistake while sometimes we deliberately make use of these loops in our programs. Let us take some examples and see what type of mistakes lead to infinite loops.

- (A) `for(i=0;i<=5;i--)`

```
    printf("%d", i);
```

This loop will execute till the value of i is less than or equal to 5 i.e. the loop will terminate only when i becomes greater than 5. The initial value of i is 0 and after each iteration its value is decreasing, hence it will never become greater than 5. So the loop condition will never become false and the loop will go on executing infinitely. For the loop to work correctly we should write `i++` instead of `i--`.

- (B) There should be a statement inside the loop body that changes the value of loop variable after each iteration. In for loop this work is done by the update expression but in while and do while we may forget to change the loop variable and this can lead to infinite loop.

```
int k=1;
do
{
    printf("%d", k);
    sum=sum+k;
}while(k<5);
```

Here we are not changing the value of k inside the loop body and hence the loop becomes infinite.

- (C) A common mistake made by beginners is to use the assignment operator(`=`) where equality operator(`==`) should have been used. If this mistake is made in the loop condition then it may cause the loop to execute infinitely. For example consider this loop:

```
while(n=2)
{
    .....
}
```

Here we wanted the loop to execute till the value of n is equal to 2. So we should have written

`n= =2` but mistakenly we have written `n = 2`. Now `n = 2` is an assignment expression and the value of this expression is 2, which is a nonzero(true) value and hence the loop condition is always true.

(D) `int i;
for(i=32000;i<=32767;i++)
printf("%d ", i);`

Everything seems to be correct with this loop but even then it executes infinitely. This is because `i` is an int variable and the range of an int variable is from -32768 to 32767. As the value of `i` exceeds 32767 it goes on the negative side and this process goes on, leading to an infinite loop. The output is-

32000 32001..... 32767 -32768 -32767 -1 0 1 2 32767 -32768

(E) `float k=2.0;
while(k!=3.0)
{
 printf("%f\n", k);
 k=k+0.2;
}`

This loop is infinite because the computer represents a floating point value as an approximation of the real value. The computer may represent the value 3.0 as 2.999999 or may be as 3.000001. So our condition (`k!=3.0`) never becomes false. The solution for this problem is to write the condition as (`k<=3.0`).

(F) `int i=1;
while(i<=5);
printf("%d", i++);`

This loop will produce no output and will go on executing infinitely. The mistake here is that we have put a semicolon after the condition. So this loop is treated as:

`while(i<=5)`

This null statement is treated as the body of loop and it keeps on executing.

These were some of the examples where infinite loops occur due to mistakes, but sometimes infinite loops are intentionally used in programs. To come out of these loops break or goto statements are used

```
while(1)      for( ; ; )      do
{           {           {
.....       .....       .....
.....       .....       .....
}           }           }while(1);
```

These types of loops are generally used in menu driven programs.

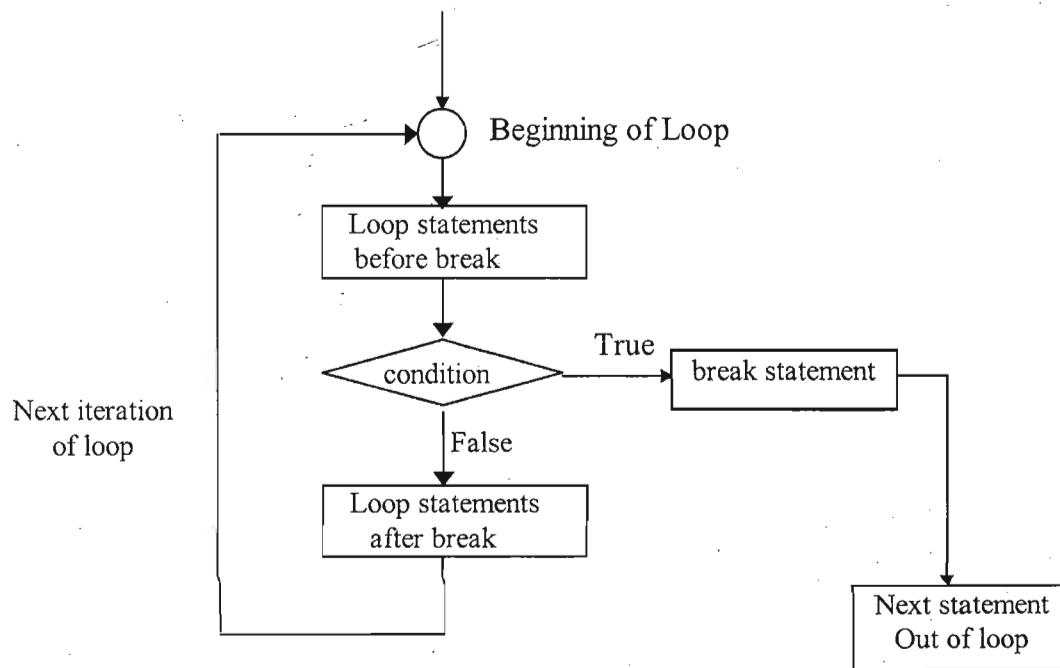
5.4 break statement

break statement is used inside loops and switch statements. Sometimes it becomes necessary to come out of the loop even before the loop condition becomes false. In such a situation, break statement is used to terminate the loop. This statement causes an immediate exit from that loop in which thi

statement appears. It can be written as-

break;

When break statement is encountered, loop is terminated and the control is transferred to the statement immediately after the loop. The break statement is generally written along with a condition. If break is written inside a nested loop structure then it causes exit from the innermost loop.



Break (control statement)

```

/*P5.26 Program to understand the use of break*/
#include<stdio.h>
main()
{
    int n;
    for(n=1;n<=5;n++)
    {
        if(n==3)
        {
            printf("I understand the use of break\n");
            break;
        }
        printf("Number = %d\n",n);
    }
    printf("Out of for loop\n");
}

```

Output:

Number = 1
 Number = 2
 I understand the use of break
 Out of for loop

This is a simple program, not of much use but illustrates the use of break statement. Had there been no break statement, this loop would have executed 5 times. But here as the value of n becomes equal to 3, break is encountered and loop is terminated.

Now we take a program which checks whether a number is prime or not. A prime number is a number that is divisible only by 1 and itself. A number n will be prime if remainders of $n/2$, $n/3$, $n/4$ \sqrt{n} are all non-zero, or in other words n is not divisible by any number from 2 to \sqrt{n} .

```
/*P5.27 Program to find whether a number is prime or not*/
#include<stdio.h>
#include<math.h>
main()
{
    int i,num,flag=1;
    printf("Enter a number : ");
    scanf("%d",&num);

    for(i=2;i<=sqrt(num);i++)
    {
        if (num%i==0)
        {
            printf("%d is not prime\n",num);
            flag=0;
            break;
        }
    }
    if(flag==1)
        printf("%d is prime\n",num);
}
```

`sqrt()` is a library function that returns the square root of a number, we have to include header file `math.h` when we use this function. The range of for loop is from 2 to \sqrt{n} , if the number is divisible by any number from this range means the number is not prime and there is no need to check divisibility by other numbers, hence the control is transferred out of loop by break statement. Control can come out of loop in two situations, after the full execution of loop when loop condition becomes false, due to break statement. If control comes out after full execution of loop, then the value of flag will be 1 and number will be prime.

5.5 continue statement

The `continue` statement is used when we want to go to the next iteration of the loop after skipping some statements of the loop. This `continue` statement can be written simply as-

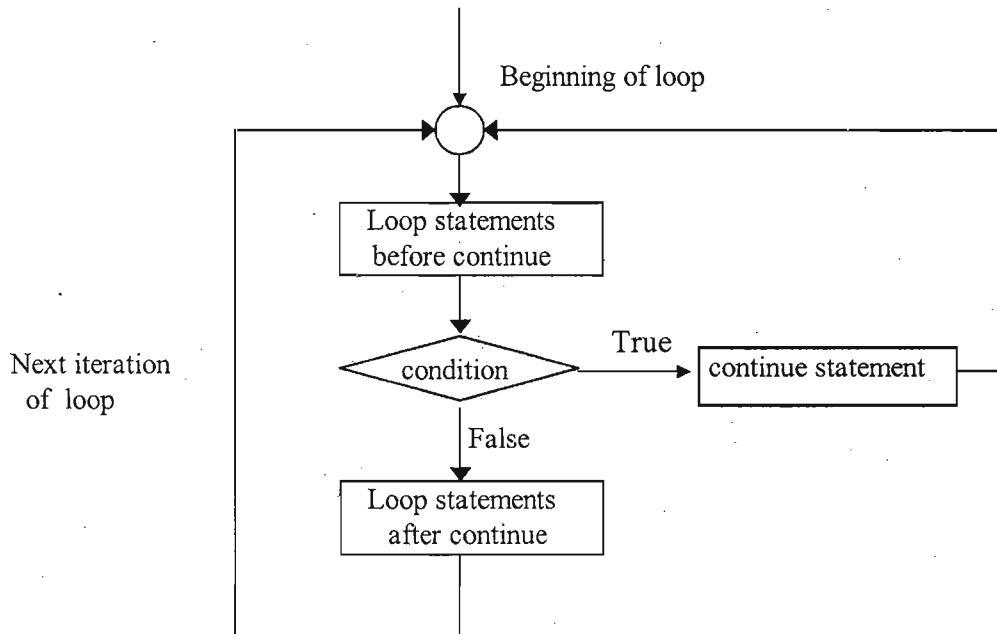
`continue;`

It is generally used with a condition. When `continue` statement is encountered all the remaining statements in the loop body are skipped and the control goes to the next iteration.

(statements after continue) in the current iteration are not executed and the loop continues with the next iteration.

The difference between break and continue is that when break is encountered the loop terminates and the control is transferred to the next statement following the loop, but when a continue statement is encountered the loop is not terminated and the control is transferred to the beginning of the loop.

In while and do-while loops, after continue statement the control is transferred to the test condition and then the loop continues, whereas in for loop after continue statement the control is transferred to update expression and then the condition is tested.



continue (control statement)

```

/*P5.28 Program to understand the use of continue statement*/
#include<stdio.h>
main()
{
    int n;
    for(n=1;n<=5;n++)
    {
        if(n==3)
        {
            printf("I understand the use of continue\n");
            continue;
        }
        printf("Number = %d\n",n);
    }
    printf("Out of for loop\n");
}

```

Output

```
Number = 1
Number = 2
I understand the use of continue
Number = 4
Number = 5
Out of for loop
```

```
/*P5.29 Program to find the sum and average of 10 positive integers*/
#include<stdio.h>
main()
{
    int i=1,n,sum=0;
    float avg;
    printf("Enter 10 positive numbers : \n");
    while(i<=10)
    {
        printf("Enter number %d : ",i);
        scanf("%d", &n);
        if(n<0)
        {
            printf("Enter only positive numbers\n");
            continue;
        }
        sum+=n;
        i++;
    }
    avg=sum/10.0;
    printf("Sum = %d Avg = %f\n",sum,avg);
}
```

In this program if any negative number is entered then a message is displayed and the control is transferred to the beginning of loop.

5.6 goto

This is an unconditional control statement that transfers the flow of control to another part of the program.

The goto statement can be used as-

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

Here label is any valid C identifier and it is followed by a colon.

Whenever the statement goto label; is encountered, the control is transferred to the statement that follows label.

immediately after the label.

```
/*P5.30 Program to print whether the number is even or odd*/
#include<stdio.h>
main()
{
    int n;
    printf("Enter the number : ");
    scanf(" %d", &n);
    if(n%2==0)
        goto even;
    else
        goto odd;
even :
    printf("Number is even");
    goto end;
odd :
    printf("Number is odd");
    goto end;
end:
    printf("\n");
}
```

Output:

Enter the number : 14

Number is even

The label can be placed anywhere. If the label is after the goto then the control is transferred forward and it is known as forward jump or forward goto, and if the label is before the goto then the control is transferred backwards and it is known as backward jump or backward goto. In forward goto, the statements between goto and label will not be executed and in backward goto statements between goto and label will be executed repeatedly. More than one goto can be associated with the same label but we cannot have same label at more than one place.

The control can be transferred only within a function using goto statement. (Concept of functions will be introduced in further chapters)

There should always be a statement after any label. If label is at the end of program, and no statements are to be written after it, we can write the null statement (single semicolon) after the label because a program can't end with a label.

The use of 'goto' should be avoided, as it is difficult to understand where the control is being transferred. Sometimes it leads to "spaghetti" code, which is not understandable and is very difficult to debug and maintain. We can always perform all our jobs without using goto, and the use of goto is not favoured in structured programming.

Although the use of goto is not preferred but there is a situation where goto can actually make the code simpler and more readable. This situation arises when we have to exit from deeply nested loops. To exit from a single loop we can use the break statement, but in nested loops break will take the control only out of the innermost loop.

```

for( .........)
{
    while(.....)
    {
        for( .........)
        {
            if( .........)
                goto stop;
        }
    }
}
stop:

```

As mentioned earlier, we can always write any code without using goto, so here also we have another way of exiting out of the deeply nested loop.

```

flag=0;
for( .........)
{
    while(.....)
    {
        for( .........)
        {
            if( .........)
            {
                flag=1;
                break; /*out of innermost for loop*/
            }
        }
        if( flag==1)
            break; /*out of while loop*/
    }
    if( flag==1)
        break; /* out of outer for loop*/
}

```

We can see that in the case of nested loops, the code using goto is more readable and if it is not used then many tests have to be performed.

5.7 switch

This is a multi-directional conditional control statement. Sometimes there is a need in program to make choice among number of alternatives. For making this choice, we use the switch statement. This can be written as-

```

switch(expression)
{
    case constant1:
        statement
        .....
    case constant2:
        statement
        .....
        .....
    case constantN:
        statement
        .....
    default :
        statement
        .....
}

```

Here **switch**, **case** and **default** are keywords. The "expression" following the **switch** keyword can be any C expression that yields an integer value. It can be value of any integer or character variable, or a function call returning an integer, or an arithmetic, logical, relational, bitwise expression yielding an integer. It can be any integer or character constant also. Since characters are converted to their ASCII values, so we can also use characters in this expression. Data types **long int** and **short int** are also allowed.

The constants following the **case** keywords should be of integer or character type. They can be either constants or constant expressions. These constants must be different from one another.

We can't use floating point or string constants. Multiple constants in a single case are not allowed; each case should be followed by only one constant.

Each case can be followed by any number of statements. It is also possible that a case has no statement under it. If a case is followed by multiple statements, then it is not necessary to enclose them within pair of curly braces, but it is not an error if we do so. The statements under case can be any valid C statements like if else, while, for or even another switch statement. Writing a switch statement inside another is called nesting of switches. Now we'll see some valid and invalid ways of writing switch expressions and case constants.

```
int a, b, c;      char d, e;      float f;
```

Valid

```
switch(a)  switch(a>b)  switch(d +e-3)  switch(a>b && b>c)  switch(func(a, b))
```

Invalid

```
switch(f)  switch(a+4.5)
```

Valid

```
case 4:      case 'a':      case 2+4:      case 'a'>'b':
```

Invalid

```
case "second":  case 2.3:  case a:  case a>b:  case a+2:  case 2, 4, 5:  
case 2 : 4 : 5 :
```

Now let us see how the switch statement works. Firstly the **switch expression** is evaluated, then the

value of this expression is compared one by one with every case constant. If the value of expression matches with any case constant, then all statements under that particular case are executed. If none of the case constant matches with the value of the expression then the block of statements under default is executed. 'default' is optional, if it is not present and no case matches then no action takes place. These cases and default can occur in any order.

```
/*P5.31 Program to understand the switch control statement*/
#include<stdio.h>
main()
{
    int choice;
    printf("Enter your choice : ");
    scanf("%d",&choice);
    switch(choice)
    {
        case 1:
            printf("First\n");
        case 2:
            printf("Second\n");
        case 3:
            printf("Third\n");
        default:
            printf("Wrong choice\n");
    }
}
```

Output:

```
Enter your choice : 2
Second
Third
Wrong choice
```

Here value of choice matches with second case so all the statements after case 2 are executed sequentially. The statements of case 3 and default are also executed in addition to the statements of case 2. This is known as falling through cases.

Suppose we don't want the control to fall through the statements of all the cases under the matching case, then we can use break statement. If a break statement is encountered inside a switch, then all the statements following break are not executed and the control jumps out of the switch. Let's rewrite the above program using break.

```
/*P5.32 Program to understand the switch with break statement*/
#include<stdio.h>
main()
{
    int choice;
    printf("Enter your choice : ");
    scanf("%d",&choice);

    switch(choice)
    {
        case 1:
```

```

printf("First\n");
break; //break statement in switch*
case 2:
printf("Second\n");
break;
case 3:
printf("Third\n");
break;
default:
printf("Wrong choice\n");
}
}/*End of main()*/

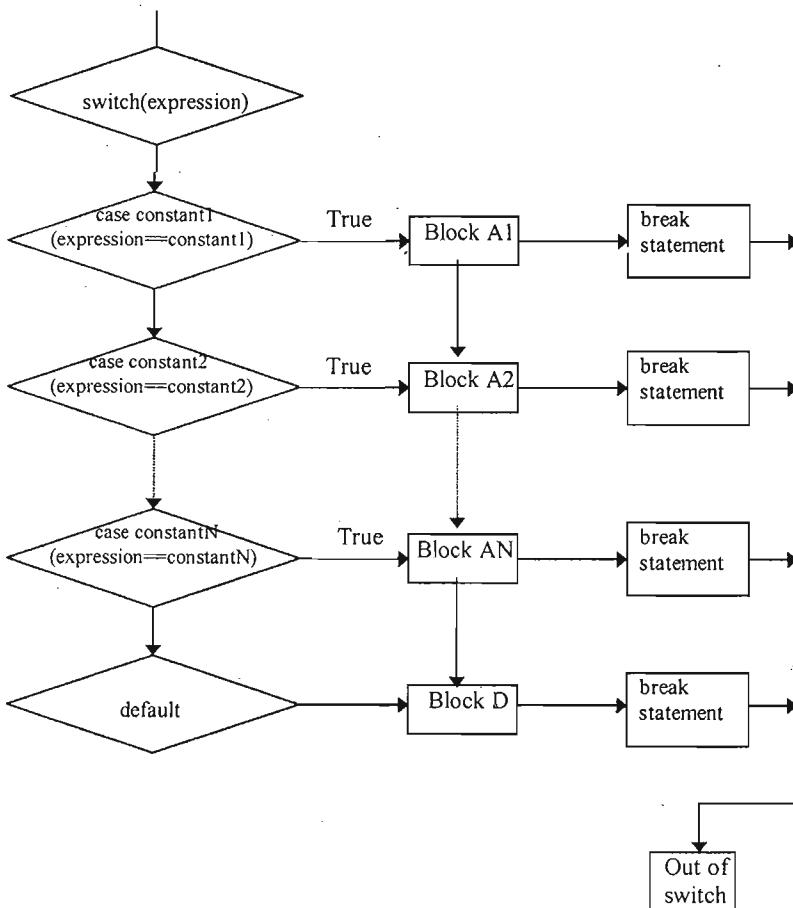
```

Output:

Enter your choice : 2 :

Second :

It is not necessary that if break is present inside switch, then it should be present after all case statements. Some cases may have break while others may not have. So the use of break inside switch is optional, it may or may not be used depending on the requirement and logic of the program. The use of continue statement is not valid inside switch.



The break statement is not logically needed after the last case(or default if it is at the last) but it is a good programming practice to do so because in future we may need to add new cases at the last.

```
/*P5.33 Program to perform arithmetic calculations on integers*/
#include<stdio.h>
main()
{
    char op;
    int a,b;
    printf("Enter number operator and another number : ");
    scanf("%d%c%d",&a,&op,&b);
    switch(op)
    {
        case '+':
            printf("Result = %d\n",a+b);
            break;
        case '-':
            printf("Result = %d\n",a-b);
            break;
        case '*':
            printf("Result = %d\n",a*b);
            break;
        case '/':
            printf("Result = %d\n",a/b);
            break;
        case '%':
            printf("Result = %d\n",a%b);
            break;
        default:
            printf("Enter valid operator\n");
    }/*End of switch*/
}/*End of main()*/

```

Output:

Enter number operator and another number : 2+5

Result = 7

In this program, the valid operator for multiplication is only '*', if we want to make 'x', 'X' also valid operators for multiplication, then we can modify above switch statement like this.

```
switch(op)
{
    .....
    case '-':
        result = a-b;
        break;
    case 'x':
    case 'X':
    case '*':
        result = a*b;
```

```

        break;
    case '/':
    .....
    .....
}/*End of switch*/

```

Here we have added two cases which have no statements. In both these cases, statements of case '*' will be executed since there is no break after case 'x' and case 'X'.

```

/*P5.34 Program to find whether the alphabet is a vowel or consonant*/
#include<stdio.h>
main( )
{
    char ch;
    printf("Enter an alphabet : ");
    scanf("%c", &ch);
    switch(ch)
    {
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':
            printf("Alphabet is a vowel\n");
            break;
        default:
            printf("Alphabet is a consonant\n");
    }
}

```

In this program, if someone enters a vowel in uppercase then it will be regarded as a consonant. One solution to this problem is to add 5 more cases for uppercase vowels. But we can solve this problem in another way without adding any cases. We have studied earlier that switch expression can be a function call returning an int or char. So change the switch expression to tolower(ch) and now our problem will be solved. tolower() is a library function which converts uppercase to lowercase, header file ctype.h has to be included to use this function.

Switch statement is very important in menu driven programs. Here we'll make a menu driven(dummy) program which is identical to a database application. Although this program performs nothing useful but it shows us the structure of menu driven programs.

```

/*P5.35 A menu driven program using infinite loop and switch*/
#include<stdio.h>
main( )
{
    int choice;
    while(1)
    {
        printf("1.Create database\n");
        printf("2.Insert new record\n");
        printf("3.Modify a record\n");
        printf("4.Delete a record\n");
    }
}

```

```

printf("5.Display all records\n");
printf("6.Exit\n");
printf("Enter your choice : ");
scanf("%d",&choice);
switch(choice)
{
    case 1:
        printf("Database created.....\n\n");
        break;
    case 2:
        printf("Record inserted.....\n\n");
        break;
    case 3:
        printf("Record modified.....\n\n");
        break;
    case 4:
        printf("Record deleted.....\n\n");
        break;
    case 5:
        printf("Records displayed.....\n\n");
        break;
    case 6:
        exit(1);
    default:
        printf("Wrong choice\n");
}/*End of switch*/
}/*End of while*/
}/*End of main()*/

```

Here we have taken switch inside an infinite loop. The function exit() is used to terminate the program. Each time this loop executes, all options will be displayed and we'll be asked to enter our choice. Appropriate action will be taken depending on our choice. If we enter choice 6, then we'll come out of the program. In real programs we'll write full code inside the cases instead of simple printf statements.

5.8 Some Additional Problems

Problem 1

Write a program to find whether a date entered is valid or not. Assume that dates between years 1850 and 2050 are valid.

```

/*P.36 Program to check whether a date is valid or not.*/
#include<stdio.h>
main()
{
    int d,m,y;
    int flag=1,isleap=0;
    printf("Enter date (dd/mm/yyyy) : ");
    scanf("%d/%d/%d",&d,&m,&y);
    if(y%4==0 && y%100!=0 || y%400==0)
        isleap=1;
    if(y<=1850 || y>=2050)

```

```

    flag=0;
else if(m<1 || m>12)
    flag=0;
else if(d<1)
    flag=0;
else if(m==2)           /*check for number of days in February*/
{
    if(d>28)
        flag=0;
    if(d==29&&isleap)
        flag=1;
}
else if(m==4 || m==6 || m==9 || m==11)/*Check days in april, june,
sept, nov*/
{
    if(d>30)
        flag=0;
}
else if(d>31)
    flag=0;

if(flag==0)
    printf("Not a valid date\n");
else
    printf("Valid Date \n");
}/*End of main() */

```

Problem 2

Write a program to find difference of two dates in years, months and days. Assume that the dates are entered in valid range and that the first date falls before second date.

```

/*P5.37 Program to get difference of two dates in years, months, days*/
#include<stdio.h>
main()
{
    int d1,d2,d,m1,m,y1,y2,y;
    printf("Enter first date (dd/mm/yyyy) : ");
    scanf("%d/%d/%d",&d1,&m1,&y1);
    printf("Enter second date (dd/mm/yyyy) : ");
    scanf("%d/%d/%d",&d2,&m2,&y2);
    if(d2<d1)
    {
        if(m2==3)
        {
            if(y2%4==0 && y2%100!=0 || y2%400==0) /*check for leap*/
                d2=d2+29;
            else
                d2=d2+28;
        }
        else if(m2==5 || m2==7 || m2==10 || m2==12)
            d2=d2+30;
    }
}
```

```

    else
        d2=d2+31;
        m2=m2-1;
    }
    if(m2<m1)
    {
        y2=y2-1;
        m2=m2+12;
    }
    y=y2-y1;
    m=m2-m1;
    d=d2-d1;
    printf("Difference of the two dates is : ");
    printf("%d years %d months %d days\n",y,m,d);
}/*End of main()*/

```

Here we are subtracting first date from second date, so we can get the difference in years, months and days by writing:

$$y = y_2 - y_1; \quad m = m_2 - m_1; \quad d = d_2 - d_1;$$

We have assumed that the second date falls after first date, so y_2 will always be greater or equal to y_1 and y will always come out positive.

It is possible that m_2 is less than m_1 , in this case m will come out negative, and if d_2 is less than d_1 then d will come out to be negative. So before calculating y , m and d we should make sure that m_2 is greater than or equal to m_1 and d_2 is greater than or equal to d_1 .

If d_2 is less than d_1 , then we borrow a month from m_2 and add the days of that month to d_2 . Now since all the months have different number of days, the days added to d_2 will depend on the month borrowed. We'll always borrow a month that is before m_2 . For example if $m_2 = 5$ (May), then we'll borrow month April from m_2 , so we'll add 30 to d_2 .

$$d_2 = d_2 + 30; \quad m_2 = m_2 - 1;$$

If $m_2 = 3$ (March), then we'll borrow month February from m_2 , so we'll add 28 or 29 (if it is leap year) to d_2 .

$$d_2 = d_2 + 28; \quad m_2 = m_2 - 1;$$

In the program we've done this through nested if else statements.

If m_2 is less than m_1 , then we borrow 1 year(12 months) from y_2 and add it to m_2 .

$$m_2 = m_2 + 12; \quad y_2 = y_2 - 1;$$

Now m_2 will become greater than m_1 .

Problem 3

Write a program to multiply two numbers by Russian peasant method. Russian peasant method multiplies any two positive numbers using multiplication by 2, division by 2 and addition. Here the first number is divided by 2(integer division), and the second is multiplied by 2 repeatedly until the first number reduces to 1. Suppose we have to multiply 19 by 25, we write the result of division and multiplication by 2, in the two columns like this:

19	25	Add
9	50	Add
4	100	
2	200	
<u>1</u>	<u>400</u>	Add
	475	

Now to get the product we'll add those values of the right hand column, for which the corresponding left column values are odd. So 25, 50, 400 will be added to get 475, which is the product of 19 and 25. Now we will see how this method can be implemented in program.

```
/*P5.38 Program to multiply two numbers by russian peasant method*/
#include<stdio.h>
main()
{
    int a,b,x,y,s=0;
    printf("Enter two numbers to be multiplied : ");
    scanf("%d%d",&x,&y);
    a=x;
    b=y;
    while(a>=1)           /*Loop till first number reduces to 1*/
    {
        if(a%2!=0)         /*If first number is odd*/
            s=s+b;          /*Add second number to s*/
        a/=2;                /*Divide first number by 2*/
        b*=2;                /*Multiply Second number by two*/
    }
    printf("%d * %d = %d\n",x,y,s);
}
```

Problem 4

Write a currency program, that tells you how many number of 100, 50, 20, 10, 5, 2 and 1 Rs notes will be needed for a given amount of money. For example if the total amount is Rs 545 then five 100 Rs notes, two 20 Rs note and one 5 Rs note will be needed. This sort of program can be used in ATM machines.

```
/*P5.39 Program to find out the number of notes required for a given
amount of money*/
#include<stdio.h>
main( )
{
    int n,choice,notes;
    printf("Enter the total amount in Rs : ");
    scanf("%d",&n);
    printf("Enter the value of note from which u want to begin : ");
    scanf("%d",&choice);
    switch (choice)
    {
        default:
            printf("Enter only valid values");
    }
}
```

```

        break;
case 100:
    notes=n/100;
    printf("Number of 100 Rs notes = %d\n",notes);
    n=n%100;
case 50:
    notes=n/50;
    printf("Number of 50 Rs notes = %d\n",notes);
    n=n%50;
case 20:
    notes=n/20;
    printf("Number of 20 Rs notes = %d\n",notes);
    n=n%20;
case 10:
    notes=n/10;
    printf("Number of 10 Rs notes = %d\n",notes);
    n=n%10;
case 5:
    notes=n/5;
    printf("Number of 5 Rs notes = %d\n",notes);
    n=n%5;
case 2:
    notes=n/2;
    printf("Number of 2 Rs notes = %d\n",notes);
    n=n%2;
case 1:
    notes=n/1;
    printf("Number of 1 Rs notes = %d\n",notes);
}
printf("\n");
}

```

Output:

Enter the total amount in Rs : 748

Enter the value of note from which u want to begin : 50

Number of 50 Rs notes = 14

Number of 20 Rs notes = 2

Number of 10 Rs notes = 0

Number of 5 Rs notes = 1

Number of 2 Rs notes = 1

Number of 1 Rs notes = 1

The logic of the program is simple, break statements are not used. The default statement is in the beginning and there is a break after the default.

Problem 5

Write a program that finds out the day of week from a given date.

The formula for calculating the day is-

day = (y + j + f - h + fh) % 7;

j = julian day of the date

y = year of given date(in 4 digits)

f = int part of (y-1)/4

h = int part of (y-1)/100

fh = int part of (y-1)/400

The value of variable day tells us the day of week.

Value of variable day	Name of day of week
0	Saturday
1	Sunday
2	Monday
3	Tuesday
4	Wednesday
5	Thursday
6	Friday

In the above formula, julian day of a date represents the day of year on which the date falls. Julian day of 1st Jan is 1, of 2nd Feb is 33, of 31st Dec is 365(366 if leap year).

Now let's decide the structure of our program. Once we get the value of variable day, we can use switch to print the name of day of the week from the value of day.

To calculate the value of day we'll have to find out the value of julian day. Let's assume that the day, month and year of entered date are stored in variables d, m and y respectively. We take another variable j for the value of julian day.

Initially the value of j is taken equal to d, and then days of previous months are added to j to get the value of julian day. We can use switch statement for this and take m as the switch variable.

```

j=d;
switch(m)
{
    case 2:   j+=31;    break;
    case 3:   j+=31+28;  break;
    case 4:   j+=31+28+31; break;
    case 5:   j+=31+28+31+30; break;
    case 6:   j+=31+28+31+30+31; break;
    case 7:   j+=31+28+31+30+31+30; break;
    case 8:   j+=31+28+31+30+31+30+31; break;
    case 9:   j+=31+28+31+30+31+30+31+31; break;
    case 10:  j+=31+28+31+30+31+30+31+31+30; break;
    case 11:  j+=31+28+31+30+31+30+31+31+30+31; break;
    case 12:  j+=31+28+31+30+31+30+31+31+30+31+30; break;
}

```

Suppose the date is 2nd May 2002, then d = 2, m = 5, so the control will go in case 5, and julian day will be $2+31+28+31+30 = 122$

The above logic applied in switch statement was simple but as we can see the whole switch statement

is confusing to read and understand. If we omit the break statements, take the cases in descending order, and take the switch variable as m-1 then we can accomplish our job by writing a very concise switch statement.

```
j=d;
switch(m - 1)
{
    case 11: j+=30;
    case 10: j+=31;
    case 9: j+=30;
    case 8: j+=31;
    case 7: j+=31;
    case 6: j+=30;
    case 5: j+=31;
    case 4: j+=30;
    case 3: j+=31;
    case 2: j+=28;
    case 1: j+=31;
}
```

Here if d = 2, m = 5 then control will go in case 4(5-1), and since there is no break, all statements of cases 4, 3, 2, 1 will be executed. Hence $j = 2+30+31+28+31 = 122$. This shows that we can use the flexibility of switch statement according to the need and logic of the program.

We have found out the julian day, but what if the year is leap. In that case we'll have to add 1 to the julian day, if the month is other than January or February. This is how we'll do it.

```
if(y%4==0&&y%100!=0 || y%400==0) /*check for leap year*/
    if(m!=1&&m!=2)
        j=j+1;
```

Recall that a non centennial year is leap if it is divisible by 4, and a centennial year (divisible by 100) is leap if it is divisible by 400.

```
/*P5.40 Program to find day of week from a given date*/
#include<stdio.h>
main()
{
    int d,m,y,j,f,h,fh,day;
    printf("Enter date (dd/mm/yyyy) : ");
    scanf("%d/%d/%d",&d,&m,&y);
    j=d;
    switch(m - 1)
    {
        case 11: j+=30;
        case 10: j+=31;
        case 9: j+=30;
        case 8: j+=31;
        case 7: j+=31;
        case 6: j+=30;
        case 5: j+=31;
        case 4: j+=30;
        case 3: j+=31;
```

```

        case 2: j+=28;
        case 1: j+=31;
    }
    if(y%4==0&&y%100!=0||y%400==0)
        if(m!=1&&m!=2)
            j=j+1;
    f=(y-1)/4;
    h=(y-1)/100;
    fh=(y-1)/400;
    day=(y+j+f-h+fh)%7;
    switch(day)
    {
        case 0: printf("Saturday\n"); break;
        case 1: printf("Sunday\n"); break;
        case 2: printf("Monday\n"); break;
        case 3: printf("Tuesday\n"); break;
        case 4: printf("Wednesday\n"); break;
        case 5: printf("Thursday\n"); break;
        case 6: printf("Friday\n"); break;
    }
}/*End of main()*/

```

Problem 6

Write a program to print triad numbers. Any three numbers will be triad numbers if they satisfy the following conditions-

1. Each number is a three digit number.
 2. All the digits in the three numbers (total 9 digits) should be different.
 3. Second number should be twice the first number and third number should be thrice the first number.
- For example-

219	438	657
267	534	801

```

/*P5.41 Program to print triad numbers*/
#include<stdio.h>
main( )
{
    int m,n,p,num;
    int i,k,d1,d2,d3;
    for(num=100;num<=999/3;num++) /*Loop A*/
    {
        for(i=num;i<=3*num;i+=num) /*loop B */
        {
            k=i;
            d1=k%10; k/=10;
            d2=k%10; k/=10;
            d3=k%10; k/=10;
            if(d1==d2 || d2==d3 || d3==d1)
                goto nextnum;
        }/*End of loop B*/
    }
}
```

```

for(m=num;m>0;m/=10) /*Loop C*/
{
    d1=m%10;
    for(n=num*2;n>0;n/=10) /*Loop D*/
    {
        d2=n%10;
        for(p=num*3;p>0;p/=10) /*Loop E*/
        {
            d3=p%10;
            if(d1==d2||d2==d3||d1==d3)
                goto nextnum;
        }/*End of Loop E*/
    }/*End of Loop D*/
}/*End of loop C*/
printf("%d %d %d\t",num,num*2,num*3);
nextnum:
}/*End of loop A*/
}/*End of main()*/

```

Loop A is used to generate numbers. Since triad numbers are three digit numbers, hence this loop starts from 100. The upper limit of this loop is 333 (999/3) because for numbers more than 333, 3^*num will not be a three digit number.

Loop B is inside loop A, and it executes three times, for $i = num$, $i = 2*num$ and $i = 3*num$. It finds out whether the three digits in a number(i) are different or not. In this loop d1, d2, d3 are digits of a single number, and if any two digits are found to be same then we again go to the update expression of loop A and check for next number.

Loops C, D, E are nested loops which find out whether there are any common digits in the three numbers(num , $2*num$, $3*num$). Here d1, d2, d3 are digits of num , $2*num$, $3*num$ respectively. If two digits are same then we go to the update expression of loop A to check for next number.

Note that in loops C, D and E, the digits of different numbers are compared, and in loop B digits of a single number are compared.

Problem 7

Write a program to find out the Least Common Multiple and Highest Common Factor of two numbers.

```

/*P5.42 Program to find the LCM and HCF of two numbers*/
#include<stdio.h>
main()
{
    int x,y,a,b;
    printf("Enter two numbers : ");
    scanf("%d %d",&x,&y);
    a=x;b=y;
    while(a!=b)
    {
        if(a<b)
            a=a+x;
        else
            b=b+y;
    }
}
```

```

}
printf("LCM of %d and %d is %d\n",x,y,a);
a=x; b=y;
while(a!=b)
{
    if(a>b)
        a=a-b;
    else
        b=b-a;
}
printf("HCF of %d and %d is %d\n",x,y,a);
}

```

Output

Enter two numbers : 60 135

LCM of 60 and 135 is 540

HCF of 60 and 135 is 15

5.9 Pyramids

*	1	1	1	2	1	5	5
**	2 2	1 2	2 3	3 4	0 1	5 4	4 4
***	3 3 3	1 2 3	4 5 6	4 5 6	1 0 1	5 4 3	3 3 3
****	4 4 4 4	1 2 3 4	7 8 9 10	5 6 7 8	0 1 0 1	5 4 3 2	2 2 2 2
*****	5 5 5 5 5	1 2 3 4 5	11 12 13 14 15	6 7 8 9 10	1 0 1 0 1	5 4 3 2 1	1 1 1 1 1
(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)

The program for pyramid (a) is-

```

main()
{
    int i,j,n;
    printf("Enter n : ");
    scanf("%d",&n);

    for(i=1;i<=n;i++)
    {
        for(j=1;j<=i;j++)
            printf("* ");
        printf("\n"); /*for next line of pyramid*/
    }
}

```

Here the outer for loop is for number of lines and the inner loop is for number of stars in each line. We can see that the number of stars is equal to the line number, hence the inner loop will execute once for first line, twice for second line, thrice for third line and so on.

In the above program if we print the value of i, then we'll get the pyramid (b), and on printing the value of j we'll get the pyramid(c).

For the pyramid (d) we'll take a variable p = 1, and write the printf statement as-

```
printf("%3d", p++);
```

For pyramid (e) we'll print the value of $i+j$, for pyramid (f) we'll print 1 if $(i+j)$ is even and print 0 if $(i+j)$ is odd.

For pyramid (g) we have to print $(n+1-j)$ and for pyramid (h) we have to print $(n+1-i)$. These two pyramids can also be written by reversing the loops and then printing the values of i and j . For example code for pyramid (g) can be written as-

```
for(i=n;i>=1;i--)
{
    for(j=n;j>=i;j--)
        printf("%3d",j);
    printf("\n");
}
```

* * * * *	5 5 5 5 5	1 2 3 4 5	5 4 3 2 1	1 1 1 1 1	*	*
* * * *	4 4 4 4	1 2 3 4	5 4 3 2	2 2 2 2	**	**
* * *	3 3 3	1 2 3	5 4 3	3 3 3	***	***
* *	2 2	1 2	5 4	2 2	****	****
*	1	1	5	1	*****	*****
(i)	(j)	(k)	(l)	(m)	(n)	(o)

The code for pyramid (i) is-

```
for(i=n;i>=1;i--)
{
    for(j=1;j<=i;j++)
        printf("* ");
    printf("\n");
}
```

We can see that line 1 has 5 stars, line 2 has 4 stars and so on. The outer loop is for number of line and will execute n times, but here we have taken it as a decreasing loop. So for first iteration of outer loop, inner loop will execute n times, for second iteration of outer loop inner loop will execute $n-1$ time and so on.

For pyramids (j), (k), (l), (m) we'll print values of i , j , $(n+1-j)$, $(n+1-i)$ respectively. The pyramids (j) and (m) can also be printed by reversing both the loops and then printing i and j .

For pyramid (n), we have to print spaces before printing stars. The code for it is-

```
for(i=1;i<=n;i++)           /*loop for number of lines*/
{
    for(j=1;j<=n-i;j++)     /*loop for printing spaces*/
        printf(" ");
    for(j=1;j<=i;j++)       /*loop for printing stars*/
        printf("*");
    printf("\n");            /*for next line of pyramid*/
}
```

The code for pyramid (o) is same as this one, only a space is given after star in the printf statement

*	* *	1	1	5	*****
***	* * * *	123	232	545	*****
****	* * * * *	12345	34543	54345	****
*****	* * * * * *	1234567	4567654	5432345	***
*****	* * * * * * *	123456789	567898765	543212345	*

(p)

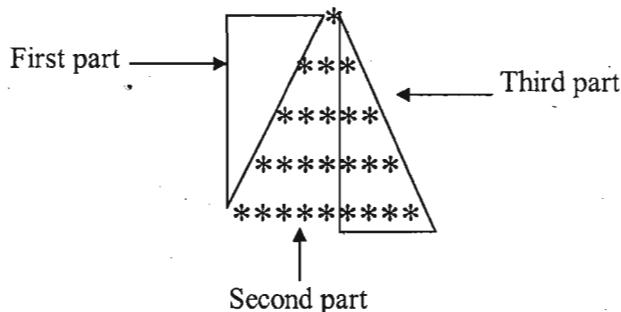
(q)

(r)

(s)

(t)

(u)



The code for pyramid (p) is-

```

for (i=1;i<=n;i++)
{
    for(j=1;j<=n-i;j++)
        printf(" ");
    for(j=1;j<=i;j++)
        printf("*");
    for(j=1;j<i;j++)
        printf("*");
    printf("\n");
}
  
```

/*loop for number of lines in pyramid*/
 /*loop for spaces (first part)*/
 /*loop for second part*/
 /*loop for third part*/
 /*for next line of pyramid*/

Here first part and second part are written same as in pyramid (n) and third part is written same as in pyramid (a).

The code for pyramid (q) will be the same, only the range of first for loop for spaces will be from 1 to $2*(n+1-i)$ and there will be a space after star in the printf statements.

The loops for pyramids (r), (s) and (t) will be same as that of pyramid (p), but here we'll take a variable p and print its value.

For pyramid (r) we'll initialize the value of p with 1 each time before second inner for loop, and then print the value of p++ in the last two for loops.

For pyramid (s) we'll initialize the value of p with i before second inner for loop, and then print the value of p++ in second for loop. After this the value of p is decreased by 1 and then the value of -p is printed in the third for loop.

For pyramid (t) we'll initialize the value of p with n before second inner for loop, and then print the value of p- - in second loop and p++ in third loop. The value of p has to be increased by 2 before third for loop. The code for pyramid (t) is-

```

for(i=1;i<=n;i++)
{
    for(j=1;j<=n-i;j++)
        /* loop for spaces (first part) */
        
```

```

        printf(" ");
p=n;
for(j=1;j<=i;j++)           /*loop for second part */
    printf("%d",p--);
p=p+2;
for(j=1;j<i;j++)           /*loop for third part */
    printf("%d",p++);
printf("\n");                /* for next line of pyramid */
}

```

The code for inverted pyramid (u) is-

```

for(i=1;i<=n;i++)
{
    for(j=1;j<=i;j++)
        printf(" ");
    for(j=1;j<=(n-i);j++)
        printf("*");
    for(j=1;j<(n-i);j++)
        printf("*");
    printf("\n");
}

```

*	1	1	5
***	123	232	545
*****	12345	34543	54345
*****	1234567	4567654	5432345
*****	123456789	567898765	543212345
*****	1234567	4567654	5432345
*****	12345	34543	54345
***	123	232	545
*	1	1	5
(v)	(w)	(x)	(y)

The diamond (v) can be obtained by joining pyramids (p) and (u) so the code is also same as that of (p) and (u). Note that if n = 5 then the upper pyramid has 5 lines while inverted pyramid has only 4 lines in this diamond. So the loop for the inverted pyramid here will range from 1 to n-1 instead of 1 to n. The code for diamond (v) is-

```

for(i=1;i<=n;i++)
{
    for(j=1;j<=n-i;j++)
        printf(" ");
    for(j=1;j<=i;j++)
        printf("*");
    for(j=1;j<i;j++)
        printf("*");
    printf("\n");
}
for(i=1;i<=n-1;i++)
{
    for(j=1;j<=i;j++)
        printf(" ");

```

```

for(j=1;j<=n-i;j++)
    printf("*");
for(j=1;j<n-i;j++)
    printf("*");
printf("\n");
}

```

The structure of loops for diamonds (w), (x) and (y) will be same as that of (v), and the logic is similar to that of pyramids (r), (s) and (t) respectively. Apply the same logic (of variable p) in the second outer for loop also.

The code for diamond (y) is-

```

for(i=1;i<=n;i++)
{
    for(j=1;j<=n-i;j++)
        printf("  ");
    p=n;
    for(j=1;j<=i;j++)
        printf("%d",p- );
    p=p+2;
    for(j=1;j<i;j++)
        printf("%d",p++ );
    printf("\n");
}
for(i=1;i<=n-1;i++)
{
    for(j=1;j<=i;j++)
        printf("  ");
    p=n;
    for(j=1;j<=n-i;j++)
        printf("%d",p- );
    p=p+2;
    for(j=1;j<n-i;j++)
        printf("%d",p++ );
    printf("\n");
}

```

In the diamond (x) initialize p with i in first for loop but with n-i in the second for loop.

Exercise

Assume stdio.h is included in all programs.

```

(1) main()
{
    int a=9;
    if(a==5)
        printf("It is important to be nice\n");
    else
        printf("It is nice to be important\n");
}

(2) main()

```

```

    {
        int a=20,b=3;
        if(a<10)
            a=a-5;
            b=b+5;
        printf("%d %d\n",a,b);
    }

(3) main()
{
    int a=9,b=0,c=0;
    if(!a<10&&!b||c)
        printf("Difficulties make us better\n");
    else
        printf("Difficulties make us bitter\n");
}

(4) main()
{
    int i=1,j=9;
    if(i>=5&&j<5);
        i=j+2;
    printf("%d\n",i);
}

(5) main()
{
    int a=0,b=0;
    if(!a)
    {
        b=!a;
        if(b)
            a=!b;
    }
    printf("%d %d \n",a,b);
}

(6) main()
{
    int a=5;
    begin:
    if(a)
    {
        printf("%d ",a);
        a--;
        goto begin;
    }
}

(7) main()
{

```

Control Statements

```

int a=5;
begin:
if(a)
    printf("%d\t",a);
a--;
goto begin;
}

(8) main()
{
    int a=2,x=10;
    if(a==2)
        if(x==8)
            printf("a is equal to 2 and x is equal to 8");
    else
        printf("a is not equal to 2");
}

(9) main()
{
    int a=6,b=4;
    while(a+b)
    {
        printf("a = %d, b = %d\n",a,b);
        a=a/2;
        b%=3;
    }
}

(10)main( )
{
    int i=10;
    do
    {
        printf("i=%d\n",i);
        i=i-3;
    }while(i);
}

(11)main( )
{
    int i,j=10;
    for(;i=j;j-=2)
        printf("%d ",j);
}

(12)main( )
{
    int i,j,x=0;
    for(i=0;i<5;i++)
        for(j=i;j>0;j--)

```

```
x=i+j+1;
printf("x = %d\n", x);
}

(13)main()
{
    int i, index=0;
    for(i=0; i<10; i++)
    {
        int i=0;
        while(i++<5)
            index++;
    }
    printf("index = %d\n", index);
}

(14)main()
{
    int i;
    for(i=1; i<10; i++)
    {
        if(i==3)
            continue;
        printf("%d ", i);
    }
}

(15)main()
{
    int i=1;
    while(i<10)
    {
        if(i==3)
            continue;
        printf("%d ", i);
        i++;
    }
}

(16)main()
{
    int i, sum;
    for(i=0; i<10; i+=3)
        sum += i*i;
    printf("%d", sum);
}

(17)main()
{
    int c=50;
    for( ; c; )
```

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

(18)main()
{
    char ch='A';
    switch(ch)
    {
        case 'A': case'B':
            ch++;
            continue;
        case 'C': case 'D':
            ch++;
    }
}
```

```
(19)main()
{
    int var=2,x=1,y=2;
    switch(var)
    {
        case x:
            x++;
        case y:
            y++;
    }
}
```

```
(20)main()
{
    char ch='A';
    while(ch<='D')
    {
        switch(ch)
        {
            case 'A': case'B':
                ch++;
                continue;
            case 'C': case 'D':
                ch++;
        }
        printf("%c",ch);
    }
}
```

```
(21)main()
{
    int n,sum=0;
    for( ; ; )
    {
```

```

        scanf("%d", &n);
        sum+=n;
        if(sum>100)
            break;
    }
}

(22)main()
{
    int i,sum1=0,sum2=0;
    for(i=1;i<5;i++)
        sum1+=i;
    i=1;
    while(i<5)
    {
        i++;
        sum2+=i;
    }
    printf("%d %d\n", i, sum1, sum2);
}

```

Programming Exercise

1. Write a program to print prime numbers from 1 to 99. (Hint : See P5.27, instead of entering a number, take a for loop that generates numbers from 1 to 99)
2. Write a program to enter a number and find the reverse of that number.
3. Input a number and a digit and find whether the digit is present in the number or not, if present then count the number of times it occurs in the number.
4. Write a program to accept any number n and print the sum of square of all numbers from 1 to n.
5. Write a program to accept any number n and print the cube of all numbers from 1 to n which are divisible by 3.
6. Write a program to accept any six digit number and print the sum of all even digits of that number and multiplication of all odd digits.
7. Write a program to find out the value of x raised to the power y, where x and y are positive integers.
8. Write a program to accept any number up to six digits and print that in words.
For example- 1265 = one two six five
9. Write a program to enter a number and test whether it is a fibonacci number or not.
10. Write a program to print all the pythagorean triplets less than 50. Any three numbers x, y, z are called pythagorean triplets if $x < y < z$ and $x^2+y^2 = z^2$
11. Find the sum of these series up to n terms where x is an integer entered by the user.

$$1 + 2 + 4 + 7 + 11 + 16 + \dots$$

$$1 + 11 + 111 + 1111 + \dots$$

$$x + x^2 + x^3 + x^4 + \dots$$

$$x + x^2 - x^3 + x^4 + \dots$$

$$1/x - 1/x^2 + 1/x^3 - 1/x^4$$

Answers

(1) It is important to be nice

The variable a is assigned the value 5, and so the if condition becomes true.

(2) 20 8

(3) Difficulties make us better

(4) 11

There is a semicolon after the if part, and it is considered as null statement.

(5) 0, 1

(6) 5 4 3 2 1

(7) This program runs infinitely, because the goto statement is not inside the if structure.

(8) a is not equal to 2.

Here the else part is paired with the second if, if we want to pair it with the first if then we should enclose the second if inside parentheses.

(9) a = 6, b = 4

a = 3, b = 1

a = 1, b = 1

a = 0, b = 1

a = 0, b = 1

a = 0, b = 1

.....

The value of b never becomes zero, and so the condition never becomes false resulting in an infinite loop.

(10) 10 7 4 1 -2 -5 -8

The value of i never becomes zero, resulting in an infinite loop.

(11) 10 8 6 4 2

The loop terminates when the value of assignment expression (i = j) becomes zero.

(12) x = 6

(13) index = 50

(14) 1 2 4 5 6 7 8 9

(15) This program prints 1 2 and then goes into an infinite loop.

(16) The variable sum is not initialized to zero, if it is initialized to zero then the result will be 126.

(17) c = 0

(18) Error, continue can't be used inside switch.

(19) Error, only constant expressions can be used in switch.

(20) DE

Here the continue statement is inside the while loop.

(21) The loop will enter numbers, and will terminate when the sum of these numbers exceeds 100.

(22) 5 10

Chapter 6

Functions

A function is a self-contained subprogram that is meant to do some specific, well-defined task. A program consists of one or more functions. If a program has only one function then it must be the main() function.

6.1 Advantages Of Using Functions

1. Generally a difficult problem is divided into sub problems and then solved. This divide and conquer technique is implemented in C through functions. A program can be divided into functions, each of which performs some specific task. So the use of C functions modularizes and divides the work of a program.
2. When some specific code is to be used more than once and at different places in the program the use of functions avoids repetition of that code.
3. The program becomes easily understandable, modifiable and easy to debug and test. It becomes simple to write the program and understand what work is done by each part of the program.
4. Functions can be stored in a library and reusability can be achieved.

C programs have two types of functions-

1. Library functions
2. User-defined functions

6.2 Library Functions

C has the facility to provide library functions for performing some operations. These functions are present in the C library and they are predefined. For example sqrt() is a mathematical library function which is used for finding out the square root of any number. The functions scanf() and printf() are input output library functions. Similarly we have functions like strlen(), strcmp() for string manipulation.

To use a library function we have to include corresponding header file using the preprocessor directive #include. For example to use input output functions like printf(), scanf() we have to include stdio.h. For mathematical library functions we have to include math.h, for string library string.h should be included. The following program illustrates the use of library function sqrt().

```
/* P6.1 Program to find the square root of any number. */
#include<stdio.h>
#include<math.h>
main()
{
    double n,s;
    printf("Enter a number : ");
```

```

scanf("%lf", &n);
s=sqrt(n);
printf("The square root of %.2lf is : %.2lf\n", n, s);
}

```

Output:

Enter a number : 16

The square root of 16.00 is : 4.00

In this program we have used three library functions – printf(), scanf() and sqrt(). We'll learn more about library functions later in this chapter.

6.3 User-Defined Functions

Users can create their own functions for performing any specific task of the program. These types of functions are called user-defined functions. To create and use these functions, we should know about these three things-

1. Function definition
2. Function declaration
3. Function call

Before discussing these three points we have written two simple programs that will be used for reference. The first program draws a line and the second one adds two numbers.

```

/*P6.2 Program to draw a line*/
#include<stdio.h>
void drawline(void);           /*Function Declaration*/
main()
{
    drawline();                /*Function Call*/
}
void drawline(void)           /*Function Definition*/
{
    int i;
    for(i=1;i<=80;i++)
        printf("-");
}

/*P6.3 Program to find the sum of two numbers*/
#include<stdio.h>
int sum(int x,int y);         /*Function declaration*/
main()
{
    int a,b,s;
    printf("Enter values for a and b : ");
    scanf("%d %d",&a,&b);
    s=sum(a, b);              /*Function call*/
    printf("Sum of %d and %d is %d\n",a,b,s);
}
int sum(int x,int y)          /*Function definition*/
{
    int s;

```

```
s=x+y;
return s;
}
```

6.4 Function Definition

The function definition consists of the whole description and code of a function. It tells what the function is doing and what are its inputs and outputs. A function definition consists of two parts - a function header and a function body. The general syntax of a function definition is-

```
return_type func_name( type1 arg1, type2 arg2, ..... )
{
    local variables declarations;
    statement;
    .....
    return(expression);
}
```

The first line in the function definition is known as the function header and after this the body of the function is written enclosed in curly braces.

The return_type denotes the type of the value that will be returned by the function. The return_type is optional and if omitted, it is assumed to be int by default. A function can return either one value or no value. If a function does not return any value then void should be written in place of return_type. func_name specifies the name of the function and it can be any valid C identifier. After function name the argument declarations are given in parentheses, which mention the type and name of the arguments. These are known as formal arguments and used to accept values. A function can take any number of arguments or even no argument at all. If there are no arguments then either the parentheses can be left empty or void can be written inside the parentheses.

The body of function is a compound statement (or a block), which consists of declarations of variables and C statements followed by an optional return statement. The variables declared inside the function are known as local variables, since they are local to that function only, i.e. they have existence only in the function in which they are declared, they can not be used anywhere else in the program. There can be any number of valid C statements inside a function body. The return statement is optional. It may be absent if the function does not return any value.

The function definition can be placed anywhere in the program. But generally all definitions are placed after the main() function. Note that a function definition cannot be placed inside another function definition. Function definitions can also be placed in different files.

In P6.2 the function definition is-

```
void drawline(void)
{
    int i;
    for(i=1;i<80;i++)
        printf("-");
}
```

Here the function is not returning any value so void is written at the place of return_type, and since it does not accept any arguments so void is written inside parentheses. The int variable i is declared inside the function body so it is a local variable and can be used inside this function only.

In P6.3 the function definition is-

```
int sum(int x, int y)
{
    int s;
    s=x+y;
    return s;
}
```

This function returns a value of type int because int is written at the place of return_type. This function takes two formal arguments x and y, both of type int. The variable s is declared inside the function so it is a local variable. The formal arguments x and y are also used as local variables inside this function.

6.5 Function Call

The function definition describes what a function can do, but to actually use it in the program the function should be called somewhere. A function is called by simply writing its name followed by the argument list inside the parentheses.

func_name(arg1, arg2, arg3...)

These arguments arg1, arg2,are called **actual arguments**.

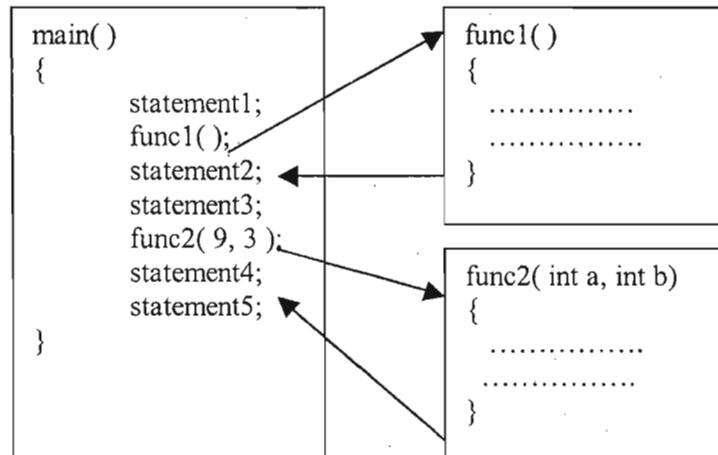
Here func_name is known as the called function while the function in which this function call is placed is known as the calling function. For example in the program P6.3, main() is the calling function, sum() is the called function and a, b are actual arguments. The **function call is written on the right hand side of the assignment operator as-**

s = sum(a, b);

Even if there are no actual arguments, the function call should have the empty parentheses. For example in program P6.2, the function call is written as-

drawline();

When a function is called, the control passes to the called function, which is executed and after this the control is transferred to the statement following the function call in the calling function. The following figure shows the transfer of control when two functions func1() and func2() are called from main().



Transfer of control when function is called

If the function returns a value, then this function call can be used like an operand in any expression anywhere in the program. The type and value of this operand will be the type and value of the return value of the function. For example-

```
s = sum(a, b);      /*Assigning the return value of sum( ) to variable s */
b = max(x, y)*10;  /*return value of max( ) is multiplied by 10 and assigned to variable b*/
if ( isprime(x) == 1) /*return value of isprime( ) is used in if condition*/
    printf("Number is prime");
printf("%d\n", sum(a+b)); /*Return value of sum( ) is printed*/
```

If the function is declared as void then it cannot be used in this way in any expression. For example it would be meaningless and invalid to write an expression like this-

```
s = drawline( );
```

A function call can be used as a statement, if a semicolon is placed after it. In this case if the function returns any value, it is just discarded. For example-

```
draw(x, y);
display( a, b, c);
printprimes( );
```

A function call cannot occur on the left hand of an assignment statement.

```
func(x, y) = a; /* Invalid */
```

The code of a function is executed only when it is called by some other function. If the function is defined and not called even once then its code will never be executed. A function can be called more than once, so the code is executed each time it is called. The execution of a function finishes either when the closing braces of the function body are reached or if return statement is encountered.

6.6 Function Declaration

The calling function needs information about the called function. If definition of the called function is placed before the calling function, then declaration is not needed. For example if in program P6.3, we write the definition of sum() before main(), then declaration is not needed.

```
/*P6.4 Program to find the sum of two numbers*/
#include<stdio.h>
int sum(int x,int y);           /*Function definition*/
{
    int s;
    s=x+y;
    return s;
}
main()
{
    int a,b,s;
    printf("Enter values for a and b : ");
    scanf("%d %d",&a,&b);
    s=sum(a,b);          /*Function call*/
    printf("Sum of %d and %d is %d\n",a,b,s);
}
```

Here the definition of sum() is written before main(), so main() knows everything about the function sum(). But generally the function main() is placed at the top and all other functions are placed after it. In this case, function declaration is needed. The function declaration is also known as the function prototype, and it informs the compiler about the following three things-

1. Name of the function.
2. Number and type of arguments received by the function.
3. Type of value returned by the function.

Function declaration tells the compiler that a function with these features will be defined and used later in the program. The general syntax of a function declaration is-

```
return_type func_name(type1 arg1, type2 arg2, .....);
```

This looks just like the header of function definition, except that there is a semicolon at the end. The names of the arguments while declaring a function are optional. These are only used for descriptive purposes. So we can write the declaration in this way also-

```
return_type func_name(type1, type2, .....);
```

In program P6.2 the declaration is written as-

```
void drawline(void);
```

In program P6.3 the declaration is written as-

```
int sum(int x, int y);
```

Now we'll write two more example programs that use functions.

```
/*P6.5 Program that finds whether a number is even or odd*/
#include<stdio.h>
void find( int n);
main()
{
    int num;
    printf("Enter a number : ");
    scanf("%d",&num);
    find(num);
}
void find( int n)
{
    if(n%2==0)
        printf("%d is even\n",n);
    else
        printf("%d is odd\n",n);
}

/*P6.6 Program that finds the larger of two numbers*/
#include<stdio.h>
int max(int x,int y);
main()
{
    int a,b;
    printf("Enter two numbers : ");
    scanf("%d%d",&a,&b);
```

$$(+) \text{ } f(x+y) = x$$

$$a \quad b$$

$$x < y$$

$$x < y$$

```

        printf("Maximum of %d and %d is : %d\n", a, b, max(a, b));
}
max(int x, int y)
{
    if(x>y)
        return x;
    else
        return y;
}

```

6.7 return statement

The return statement is used in a function to return a value to the calling function. It may also be used for immediate exit from the called function to the calling function without returning a value.

This statement can appear anywhere inside the body of the function. There are two ways in which it can be used-

```

return;
return ( expression );

```

Here return is a keyword. The first form of return statement is used to terminate the function without returning any value. In this case only return keyword is written. The following program uses this form of return statement.

```

/*P6.7 Program to understand the use of return statement*/
#include<stdio.h>
void funct(int age, float ht);
main()
{
    int age;
    float ht;
    printf("Enter age and height: ");
    scanf("%d %f", &age, &ht);
    funct(age, ht);
}
void funct(int age, float ht)
{
    if(age>25)
    {
        printf("Age should be less than 25\n");
        return;
    }
    if(ht<5)
    {
        printf("Height should be more than 5\n");
        return;
    }
    printf("Selected\n");
}

```

The second form of return statement is used to terminate a function and return a value to the calling function. The programs P6.3 and P6.6 use this second form of return statement. The value returned

by the return statement may be any constant, variable, expression or even any other function call (which returns a value). For example in program P6.3 we can directly write return (x+y) instead of taking a variable and returning it. Similarly in program P6.6 we can directly write return (x>y ? x :y)

Some other examples of valid return statements are-

```
return 1;
return x++;
return ( x+y*z );
return ( 3 * sum(a, b) );
```

It is optional to enclose the returning value in parentheses.

We can use multiple return statements in a function but as soon as first return statement is encountered the function terminates and all the statements following it are not executed.

The next function compares two dates(d1/m1/y1 and d2/m2/y2) and returns 1 if first date is smaller, returns -1 if second date is smaller , returns 0 if both dates are same. In this function we have used 7 return statements.

```
int cmpdate( int d1, int m1, int y1, int d2, int m2, int y2)
{
    if(y1<y2)
        return 1;
    if(y1>y2)
        return -1;
    if(m1<m2)
        return 1;
    if(m1>m2)
        return -1;
    if(d1<d2)
        return 1;
    if(d1>d2)
        return -1;
    return 0;
}
```

The next program uses a function that finds out the factorial of a number.

```
/*P6.8 Program to find out the factorial of a number*/
#include<stdio.h>
long int factorial(int n);
main()
{
    int num;
    printf("Enter a number : ");
    scanf("%d",&num);
    if(num<0)
        printf("No factorial of negative number\n");
    else
        printf("Factorial of %d is %ld\n",num,factorial(num));
}
long int factorial(int n)
```

```

{
    int i;
    long int fact=1;
    if(n==0)
        return 1;
    for(i=n;i>1;i--)
        fact*=i;
    return fact;
}

```

A function can return only one value. If we want to return more than one value then we have to use another technique discussed in further chapters. It is incorrect to try to return more than one value using comma.

```
return 1, 2, 3;
```

Here this expression is using comma operator and hence only the rightmost value will be returned. If no value is to be returned from the function then it should be declared as void. All functions, which are not of void type, return a value. If the function is not of void type and no value is returned through return statement, then a garbage value is returned automatically.

If the value returned is not of the type specified in the function definition then it is converted to that type if the conversion is legal.

6.8 Function Arguments

The calling function sends some values to the called function for communication; these values are called arguments or parameters.

Actual arguments: The arguments which are mentioned in the function call are known as actual arguments, since these are the values which are actually sent to the called function. Actual arguments can be written in the form of variables, constants or expressions or any function call that returns a value. For example-

```

fun(x)
func(a*b, c*d+k )
func( 22, 43 )
func( 1, 2, sum(a, b) )

```

Formal arguments: The name of the arguments, which are mentioned in the function definition are called formal or dummy arguments since they are used just to hold the values that are sent by the calling function.

These formal arguments are simply like other local variables of the function which are created when the function call starts and are destroyed when the function ends. However there are two differences. First is that formal arguments are declared inside parentheses while other local variables are declared at the beginning of the function block. The second difference is that formal arguments are automatically initialized with the values of the actual arguments passed, while other local variables are assigned values through the statements written inside the function body.

The order, number and type of actual arguments in the function call should match with the order, number and type of formal arguments in the function definition.

```
/*P6.9 Program to understand formal and actual arguments*/
#include<stdio.h>
```

```

main()
{
    int m=6,n=3;
    printf("%d\t",multiply(m,n));
    printf("%d\t",multiply(15,4));
    printf("%d\t",multiply(m+n,m-n));
    printf("%d\n",multiply(6,sum(m,n)));
}
multiply(int x,int y)
{
    int p;
    p=x*y;
    return p;
}
sum(int x, int y)
{
    return x+y;
}

```

Output:

18 60 27 54

In this program, function multiply() is called 4 times. The variables x and y are the formal arguments of multiply(). First time when the function is called, actual arguments are variables m and n, so the formal arguments x and y are initialized with the values of m and n and p=18 is returned. Second time function is called with actual arguments 15 and 4, so this time x and y are initialized with values 15 and 4 respectively, hence p=60 is returned. Similarly third time x and y are initialized with values m+n=9 and m-n=3 respectively, hence p=27 is returned. In the fourth call the first argument is a constant value (6), while the second argument is a function call. So this time, x is initialized by 6 and y is initialized by the value returned by the function sum() i.e. 9. Hence this time p=54 is returned.

The names of formal and actual arguments may be same or different, because they are written in separate functions.

```

/*P6.10 Program to understand formal and actual arguments*/
#include<stdio.h>
main()
{
    int a=6,b=3;
    func(a,b);
    func(15,4);
    func(a+b,a-b);
}
func(int a,int b)
{
    printf("a = %d      b = %d\n",a,b);
}

```

Output:

a = 6 b = 3
a = 15 b = 4
a = 9 b = 3

In the above program we can see that the values of a and b inside main() are 6 and 3, while a and b inside func() are initialized with values of actual arguments sent on each call. Any changes made to the formal arguments inside the function do not affect the actual arguments.

6.9 Types Of Functions

The functions can be classified into **four categories** on the basis of the arguments and return value.

1. Functions with no arguments and no return value.
2. Functions with no arguments and a return value.
3. Functions with arguments and no return value.
4. Functions with arguments and a return value.

6.9.1 Functions With No Arguments And No Return Value

Functions that have no arguments and no return value are written as-

```
void func(void);
main()
{
    .....
    func();
    .....
}
void func(void)
{
    .....
    statement;
    .....
}
```

In the above example, the function func() is called by main () and the function definition is written after the main() function. As the function func() has no arguments, main() can not send any data to func() and since it has no return statement, hence function can not return any value to main(). There is no communication between the calling and the called function. Since there is no return value these types of functions cannot be used as operands in expressions. The function drawline() in P6.2 is an example of these types of functions. Let us see one more example program-

```
/*P6.11 Program that uses a function with no arguments and no return values*/
#include<stdio.h>
void dispmenu(void);
main()
{
    int choice;
    dispmenu();
    printf("Enter your choice :");
    scanf("%d",&choice);
}
void dispmenu(void )
{
    printf("1.Create database\n");
    printf("2.Insert new record\n");
    printf("3.Modify a record\n");
```

```

    printf("4.Delete a record\n");
    printf("5.Display all records\n");
    printf("6.Exit\n");
}

```

6.9.2 Function With No Arguments But A Return Value

These types of functions do not receive any arguments but they can return a value.

```

int func(void);
main()
{
    int r;
    .....
    r=func();
    .....
}
int func(void)
{
    .....
    .....
    return (expression);
}

```

The next program uses a function of this type.

```

/*P6.12 Program that returns the sum of squares of all odd numbers from
1 to 25*/
#include<stdio.h>
int func(void);
main()
{
    printf("%d\n", func());
}
int func(void)
{
    int num, s=0;
    for(num=1; num<=25; num++)
    {
        if(num%2!=0)
            s+=num*num;
    }
    return s;
}

```

Output:

2925

6.9.3 Function With Arguments But No Return Value

These types of functions have arguments, hence the calling function can send data to the called function but the called function does not return any value. These functions can be written as-

```

void func(int,int);
main()
{
    .....
    func(a,b);
    .....
}
void func(int c,int d)
{
    .....
    statements
    .....
}

```

Here a and b are actual arguments which are used for sending the value, c and d are the formal arguments, which accept values from the actual arguments.

```

/*P6.13 Program to find the type and area of a triangle.*/
#include<stdio.h>
#include<math.h>
void type(float a,float b,float c);
void area(float a,float b,float c);
main()
{
    float a,b,c;
    printf("Enter the sides of triangle : ");
    scanf("%f%f%f",&a,&b,&c);
    if(a<b+c && b<c+a && c<a+b)
    {
        type(a,b,c);
        area(a,b,c);
    }
    else
        printf("No triangle possible with these sides\n");
}
void type(float a,float b,float c)
{
    if((a*a)+(b*b)==(c*c) || (b*b)+(c*c)==(a*a) || (c*c)+(a*a)==(b*b))
        printf("The triangle is right angled triangle\n");
    if(a==b && b==c)
        printf("The triangle is equilateral\n");
    else if(a==b || b==c || c==a)
        printf("The triangle is isosceles\n");
    else
        printf("The triangle is scalene\n");
}
void area(float a,float b,float c)
{
    float s; area;
    s=(a+b+c)/2;
    area=sqrt(s*(s-a)*(s-b)*(s-c));
    printf("The area of triangle = %f\n",area);
}

```

6.9.4 Function With Arguments And Return Value

These types of functions have arguments, so the calling function can send data to the called function, it can also return any value to the calling function using return statement. This function can be written as-

```
int func(int,int);
main()
{
    int r;
    .....
    r=func(a,b);
    .....
    func(c,d);
}
int func(int a,int b )
{
    .....
    .....
    return (expression);
}
```

Here return statement returns the value of the expression to the calling function. The functions sum(), multiply(), max() that we had written earlier are examples of these types of functions. Let us take one more example program-

```
/*P6.14 Program to find the sum of digits of any number*/
#include<stdio.h>
int sum(int n);
main()
{
    int num;
    printf("Enter the number : ");
    scanf("%d",&num);
    printf("Sum of digits of %d is %d\n",num,sum(num));
}
int sum(int n)
{
    int i,sum=0,rem;
    while(n>0)
    {
        rem=n%10;           /*rem takes the value of last digit*/
        sum+=rem;
        n/=10;              /*skips the last digit of number*/
    }
    return (sum);
}
```

While describing the different type of functions, we have assumed arguments and return values of type int for simplicity. These values can be of any data type.

6.10 More About Function Declaration

The main thing about the placing of function declaration is that it should be before the function call. Generally all the function declarations are written before the main() function. This way the function declaration is accessible to all functions. It is also possible that we declare the function only inside those functions that call it. Note that if function definition occurs before the function call, then declaration is not needed.

If a function returns a value of type int or char then it is not necessary to declare the function, because by default every function returns an int value. The compiler assumes that any function, for which return type is not specified in the definition, will return an int value. But if the function returns a value other than int such as float, double, pointer, array, structure etc then it is necessary to declare it before the function call.

Declaration of a function has two main uses. Firstly, it tells the compiler about the return type of the function. Secondly, it specifies the type and number of arguments of the function. So it can be used to check any mismatch between the number and type of arguments in the function definition and function call. We had seen that it is optional to declare the functions of type int or char, but it is a good practice to declare all functions since we can check the number and type of arguments in function call. So in this way we can prevent bugs, which could occur due to wrong number and type of arguments in the function call.

6.11 Declaration Of Functions With No Arguments

If a function has no arguments, then either the parentheses can be left empty or void can be written inside the parentheses. But it is better to write void because when parentheses are empty, the compiler assumes that there is no information about the arguments. There can be no arguments or any number of arguments. So the compiler won't be able to check any mismatch in the type and number of arguments in function call.

Similarly void can be specified as return type if a function does not return any value. This will avoid accidental use of function in any expression. For example-

```
/*P6.15 Program to understand declaration of functions with no arguments*/
#include<stdio.h>
void func1();
void func2(void);
main()
{
    func1(1.3, 'a'); /*will not generate any error*/
    func2(1.3, 'a'); /*will generate error*/
}
void func1()
{
    printf("\nFunction1\n");
}
void func2(void)
{
    printf("\nFunction2\n");
}
```

Here we see that func1() can be called with wrong number and type of arguments and it will not show

any compilation error, so it can be a bug hard to find. It is not possible to call the function func2() with any argument.

6.12 If Declaration Is Absent

If actual arguments are more than the formal arguments then the extra actual arguments are just ignored. If actual arguments are less than the formal arguments, then the extra formal arguments receive garbage value.

```
func(int a, int b, int c)
{
    .....
}
func(1, 2, 3, 4, 5); /*Actual arguments more than formal*/
```

Here the last two arguments are just ignored. a = 1, b = 2, c = 3

```
func(1, 2); /*Actual arguments less than formal*/
```

Here the third argument receives garbage value. a = 1, b = 2, c = garbage value

If there is a type mismatch between a corresponding actual and formal argument, then compiler tries to convert the type of actual argument to the type of the formal argument if the conversion is legal, otherwise a garbage value is passed to the formal argument.

6.13 Order Of Evaluation Of Function Arguments

When a function is called with more than one argument, then the order of evaluation of arguments is unspecified. This order of evaluation is not important in function calls like multiply(m, n) or multiply(m+n, m-n). But if we have a function call like this-

```
int m = 3, k;
k = multiply( m, m++ );
```

Now here if the first argument is evaluated first then value of k will be 9, and if the second argument is evaluated first, the value of k will be 12. But since the order of evaluation of arguments is unspecified in C and depends on compiler, hence the result may be different on different compilers. Similarly if we write-

```
int i = 10;
printf("%d %d %d", -i, i++, i);
```

Here also the result is unpredictable. So it is better to avoid these types of argument expressions that can produce different results on different compilers.

6.14 main() Function

Execution of every C program always begins with the function main(). Each function is called directly or indirectly in main() and after all functions have done their operations, control returns back to main(). There can be only one main() function in a program.

The main() function is a user defined function but the name, number and type of arguments are predefined in the language. The operating system calls the main function and main() returns a value of integer type to the operating system. If the value returned is zero, it implies that the function has terminated successfully and any nonzero return value indicates an error. If no return value is specified in main() then any garbage value will be returned automatically. Calling the function exit() with an integer value

is equivalent to returning that value from main(). The function main() can also take arguments, which will be discussed in further chapters.

The definition, declaration and call of main() function-

Function Declaration – By the C compiler

Function Definition – By the programmer

Function Call – By the operating system

6.15 Library Functions

The library functions are formally not a part of the C language, but they are supplied with every C compiler. The source code of the library functions is not given to the user. These functions are precompiled and the user gets only the object code. This object code is linked to the object code of your program by the linker. Different categories of library functions are grouped together in separate library files. When we call a library function in our program, the linker selects the code of that function from the library file and adds it to the program.

The definition, declaration and call of library functions-

Function Definition - Predefined, precompiled and present in the library.

Function Declaration - In header files (files with a .h extension)

Function Call - By the programmer

To use a library function in our program we should know-

- (i) Name of the function and its purpose
- (ii) Type and number of arguments it accepts
- (iii) Type of the value it returns
- (iv) Name of the header file to be included.

We can define any function of our own with the same name as that of any function in the C library. If we do so then the function that we have defined will take precedence over the library function with the same name. The standard library functions are discussed in chapter 18.

6.16 Old Style Of Function Declaration

The old style declaration of functions does not inform the compiler about the type and number of arguments. The syntax of old style declaration is-

```
return_type func_name( );
```

The parentheses are always empty even if the function receives arguments. This convention is supported in new compilers also so that the old programs can be compiled using the new compilers. But this convention is not used now since the new convention is better and provides more information to the compiler.

6.17 Old Style Of Function Definition

The old style of writing function definition is slightly different from the one we have studied. Although this style is no longer used but the compilers still support it.

```
return_type func_name(argument list)
argument declarations
{
    local variables declarations;
```

```

statements;
return(expression);
}

```

The compilers based on K&R C support this syntax while the new compilers based on ANSI C support both the ways of writing function headers. Let us see an example of function definition written in old style-

```

float func1(a,b,z)
int a,b;
float z;
{
    float s;
    s=(a+b)/z;
    return z;
}

```

Here are few more programs on functions.

```

/*P6.16 Program to find the reverse of a number and check whether it
is a palindrome or not.
Palindrome is a number that remains same when reversed.*/
#include<stdio.h>
int reverse(int n);
main()
{
    int num;
    printf("Enter a number : ");
    scanf("%d",&num);
    printf("Reverse of %d is %d\n",num,reverse(num));
    if(num==reverse(num))
        printf("Number is a palindrome\n");
    else
        printf("Number is not a palindrome\n");
}
reverse(int n)
{
    int rem,rev=0;
    while(n>0)
    {
        rem=n%10;
        rev=rev*10+rem;
        n/=10;
    }
    return rev;
}

```

Output:

Enter the number : 3113

Now the number is : 3113

Number is a palindrome

```
/*P6.17 Program to find whether the number is prime or not.*/

```

```

#include<stdio.h>
#include<math.h>
int isprime(int n);
main()
{
    int num;
    printf("Enter a number : ");
    scanf("%d", &num);
    if(isprime(num))
        printf("Number is prime\n");
    else
        printf("Number is not prime\n");
}
int isprime(int n)
{
    int i, flag=1;
    for(i=2;i<=sqrt(n);i++)
    {
        if(n%i==0),
        {
            flag=0;
            break;
        }
    }
    return (flag);
}

/*P6.18 Program to print all prime numbers less than 500*/
#include<stdio.h>
#include<math.h>
int isprime(int n);
main()
{
    int i;
    for(i=1;i<=500;i++)
        if(isprime(i))
            printf("%d ",i);
}
int isprime(int n)
{
    .....
    .....
}

/*P6.19 Program to print twin primes less than 500 . If two consecutive
odd numbers are both prime (e.g. 17, 19) then they are known as twin
primes.*/
#include<stdio.h>
#include<math.h>
int isprime(int n);
main()

```

```

int i=3,j;
while(i<500)
{
    j=i;
    i=i+2;
    if(isprime(j) && isprime(i))
        printf("%5d %5d\n",j,i);
}

int isprime( int n)

.....
/*P6.20 Program to convert a decimal number to binary number*/
#include<stdio.h>
long int binary(long int num);
main()

long int num;
printf("Enter the decimal number : ");
scanf("%ld",&num);
printf("Decimal = %ld , Binary = %ld\n",num,binary(num));

long binary(long int num)

long rem,a=1,bin=0;
while(num>0)
{
    rem=num%2;
    bin=bin+rem*a;
    num/=2;
    a*=10;
}
return bin;

```

The logic used in this program is discussed below-

Suppose decimal number is 29

	Quotient	Remainder	
29/2	14	1	MSB
14/2	7	0	
7/2	3	1	
3/2	1	1	
1/2	0	1	LSB

$$\begin{aligned}
 & 1*0 + 1*10 + 1*100 + 0*1000 + 1*10000 \\
 & = 10111
 \end{aligned}$$

By this method we can get binary equivalents of decimal numbers upto 1023 only, since for numbers more than 1023(Binary - 111111111), the binary equivalents will exceed the range of long int. In the next chapter we'll see a better method to print binary numbers.

```
/*P6.21 Program to raise a floating point number to an integer power
eg an where a is floating point and n is integer value*/
#include<stdio.h>
main()
{
    float a;
    int n;
    float power(float a,int n);
    printf("Enter base : ");
    scanf("%f",&a);
    printf("Enter exponent : ");
    scanf("%d",&n);
    printf("%f raised to power %d is %f\n",a,n,power(a,n));
}
float power(float a,int n)
{
    int i;
    float p=1;
    if(n==0)
        return 1;
    else
    {
        for(i=1;i<=abs(n);i++)
            p=p*a;
        if(n>0)
            return p;
        else
            return 1/p;
    }
}
```

6.18 Local, Global And Static Variables

6.18.1 Local Variables

The variables that are defined within the body of a function or a block, are local to that function block only and are called local variables. For example-

```
func()
{
    int a,b;
    .....
    .....
}
```

Here a and b are local variables which are defined within the body of the function func(). Local variables can be used only in those functions or blocks, in which they are declared. The same variable may be used in different functions. For example-

```

func1( )
{
    int a=2,b=4;
    .....
}

func2( )
{
    int a=15,b=20;
    .....
}

```

Here values of $a = 2$, $b = 4$ are local to the function $\text{func1}()$ and $a = 15$, $b = 20$ are local to the function $\text{func2}()$.

6.18.2 Global Variables

The variables that are defined outside any function are called global variables. All functions in the program can access and modify global variables. It is useful to declare a variable global if it is to be used by many functions in the program. Global variables are automatically initialized to 0 at the time of declaration.

```

/*P6.22 Program to understand the use of global variables*/
#include<stdio.h>
void func1(void);
void func2(void);
int a,b=6;
main()
{
    printf("Inside main() : a = %d, b = %d\n",a,b)
    func1();
    func2();
}

void func1(void )
{
    printf("Inside func1() : a = %d, b = %d\n",a,b);
}

void func2(void)
{
    int a=8;
    printf("Inside func2() : a = %d, b = %d\n",a,b);
}

```

Output:

```

Inside main() : a = 0, b = 6
Inside func1() : a = 0, b = 6
Inside func2() : a = 8, b = 6

```

Here a and b are declared outside all functions hence they are global variables. The variable a will be initialized to 0 automatically since it is a global variable. Now we can use these variables in any function. In $\text{func2}()$, there is local variable with the same name as global variable. Whenever there is a conflict between a local and global variable, the local variable gets the precedence. So inside $\text{func2}()$ the value

of local variable gets printed.

6.18.3 Static Variables

Static variables are declared by writing keyword static in front of the declaration.

static type var_name;

A static variable is initialized only once and the value of a static variable is retained between function calls. If a static variable is not initialized then it is automatically initialized to 0.

```
/*P6.23 Program to understand the use of static variables*/
#include<stdio.h>
void func(void);
main()
{
    func();
    func();
    func();
}
void func(void)
{
    int a=10;
    static int b=10;
    printf("a = %d      b = %d\n",a,b);
    a++;
    b++;
}
```

Output:

```
a = 10      b = 10
a = 10      b = 11
a = 10      b = 12
```

Here 'b' is a static variable. First time when the function is called 'b' is initialized to 10. Inside function, value of 'b' becomes 11. This value is retained and when next time the function is called, value of 'b' is 11 and the initialization is neglected. Similarly when third time function is called, value of 'b' is 12. Note that the variable 'a', which is not static is initialized on each call and its value is not retained.

6.19 Recursion

Recursion is a powerful technique of writing a complicated algorithm in an easy way. According to this technique a problem is defined in terms of itself. The problem is solved by dividing it into smaller problems, which are similar in nature to the original problem. These smaller problems are solved and their solutions are applied to get the final solution of our original problem.

To implement recursion technique in programming, a function should be capable of calling itself. This facility is available in C. The function that calls itself (inside function body) again and again is known as a recursive function. In recursion the calling function and the called function are same. For example,

```
main()
{
```

.....

```

    rec();
    .....
}

rec()
{
    .....
    .....
    rec(); → recursive call
    .....
}

```

Here `rec()` is called inside the body of function `rec()`. There should be a terminating condition to stop recursion, otherwise `rec()` will keep on calling itself infinitely and will never return.

```

rec()
{
    .....
    if(...) /*terminating condition*/
    .....
    rec();
}

```

Before writing a recursive function for a problem we should consider these points-

1. We should be able to define the solution of the problem in terms of a similar type of smaller problem.
At each step we get closer to the final solution of our original problem.
2. There should be a terminating condition to stop recursion.

Now we will take some problems and write recursive functions for solving them.

- (i) Factorial
- (ii) Power
- (iii) Fibonacci numbers
- (iv) Tower of hanoi

We know that the factorial of a positive integer n can be found out by multiplying all integers from 1 to n .

$$n! = 1 * 2 * 3 * \dots * (n-1) * n$$

This is the iterative definition of factorial and in the previous chapter we had written a program to compute factorial using loop. Now we'll try to find out the recursive definition of factorial.

We know that $6! = 6 * 5 * 4 * 3 * 2 * 1$

We can write it as- $6! = 6 * 5!$

Similarly we can write $5! = 5 * 4!$

So in general we can write

$$n! = n * (n-1)!$$

Now problem of finding out factorial of $(n-1)$ is similar to that of finding out factorial of n , but it is definitely smaller in size. So we have defined the solution of factorial problem in terms of itself. We know that the factorial of 0 is 1. This can act as the terminating condition. So the recursive definition of factorial can be written as-

$$n! = \begin{cases} 1 & n=0 \\ n * (n-1)! & n>0 \end{cases}$$

Now we'll write a program, which finds out the factorial using a recursive function.

```
/*P6.24 Program to find the factorial of a number by recursive method*/
#include<stdio.h>
long fact(int n);
main()
{
    int num;
    printf("Enter a number : ");
    scanf("%d",&num);
    printf("Factorial of %d is %ld\n",num,fact(num));
}
long fact(int n)
{
    if (n==0)
        return(1);
    else
        return(n*fact(n-1));
}
```

This function returns 1 if the argument n is 0, otherwise it returns $n * \text{fact}(n-1)$. To return $n * \text{fact}(n-1)$, the value of $\text{fact}(n-1)$ has to be calculated for which $\text{fact}()$ has to be called again but this time with an argument of $n-1$. This process of calling $\text{fact}()$ continues till it is called with an argument of 0.

Suppose we want to find out the factorial of 5.

Initially main() calls $\text{factorial}(5)$
 Since $5 > 0$, $\text{factorial}(5)$ calls $\text{factorial}(4)$
 Since $4 > 0$, $\text{factorial}(4)$ calls $\text{factorial}(3)$
 Since $3 > 0$, $\text{factorial}(3)$ calls $\text{factorial}(2)$
 Since $2 > 0$, $\text{factorial}(2)$ calls $\text{factorial}(1)$
 Since $1 > 0$, $\text{factorial}(1)$ calls $\text{factorial}(0)$

}

winding phase

When $\text{factorial}()$ is called with $n=0$ then the condition inside *if* statement becomes true, so now the recursion stops and control returns to $\text{factorial}(1)$.

Now every called function will return the value to the previous function. These values are returned in the reverse order of function calls.

Recursive functions work in two phases. First one is the winding phase and the next one is unwinding phase. In winding phase the function keeps on calling itself. The winding phase stops when the terminating condition arrives in a call, now the unwinding phase starts and the called functions return values in reverse order.

In the above case the function factorial() is called 6 times, but there is only one copy of that function in memory. Each function call is different from another because the argument supplied is different each time.

Now we'll write a recursive function for finding out the a^n .

The iterative definition for finding a^n is

$$a^n = a * a * a * \dots \text{ n times}$$

The recursive definition can be written as-

$$a^n = \begin{cases} 1 & n=0 \\ a * a^{n-1} & n>0 \end{cases}$$

```
/*P6.25 Program to raise a floating point number to a positive integer
using recursion*/
#include<stdio.h>
float power(float a,int n);
main()
{
    float a,p;
    int n;
    printf("Enter a and n : ");
    scanf("%f%d",&a,&n);
    p=power(a,n);
    printf("%f raised to power %d is %f\n",a,n,p);
}
float power(float a,int n)
{
    if(n==0)
        return(1);
    else
        return(a*power(a,n-1));
}
```

In P5.20, we had written a program to print the fibonacci series. Now we'll write a recursive definition for finding fibonacci numbers.

$$\text{fib}(n) = \begin{cases} 1 & n=0 \text{ or } n=1 \\ \text{fib}(n-1) + \text{fib}(n-2) & n>1 \end{cases}$$

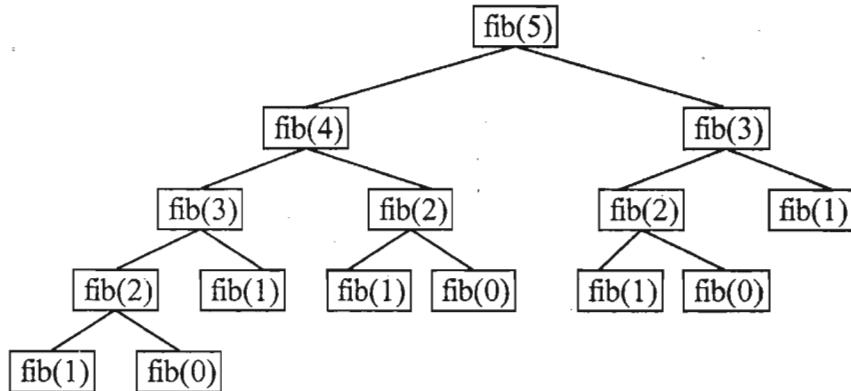
```
/*P6.26 Program to generate fibonacci series using recursion*/
#include<stdio.h>
```

```

main()
{
    int nterms, i;
    printf("Enter number of terms : ");
    scanf("%d", &nterms);
    for(i=0;i<nterms;i++)
        printf("%d ", fib(i));
    printf("\n");
}
int fib(int n)           /*recursive function that returns nth term of
                           fibonacci series*/
{
    if(n==0 || n==1)
        return(1);
    else
        return(fib(n-1)+fib(n-2));
}

```

The difference from previous two functions (factorial and power) is that here the function fib() is called two times inside the function body. The following figure shows the recursive calls of function fib() when it is called with argument 5.



6.19.1 Tower Of Hanoi

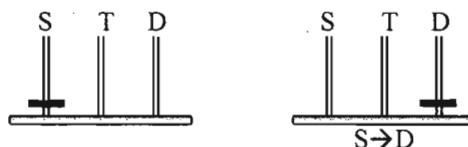
The problem of Tower of Hanoi is to move disks from one pillar to another using a temporary pillar. Suppose we have a source pillar S which has finite number of disks, and these disks are placed in a decreasing order i.e. largest disk is at the bottom and the smallest disk is at the top. Now want to place all these disks on destination pillar D in the same order. We can use a temporary pillar T to place the disks temporarily whenever required. The conditions for this game are-

1. We can move only one disk from one pillar to another at a time.
2. Larger disk cannot be placed on smaller disk.

Suppose the number of disks on pillar S is n. First we'll solve the problem for n=1, n=2, n=3 and then we'll develop a general procedure for the solution.

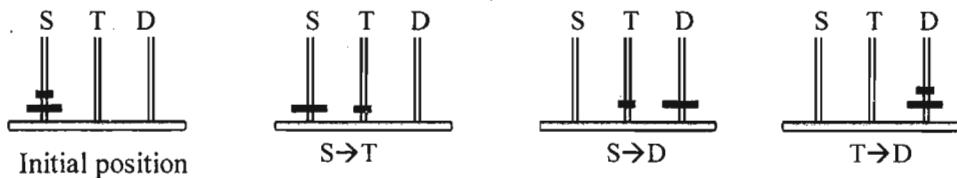
Here S is the source, D is the destination pillar and T is the temporary pillar.

For n=1



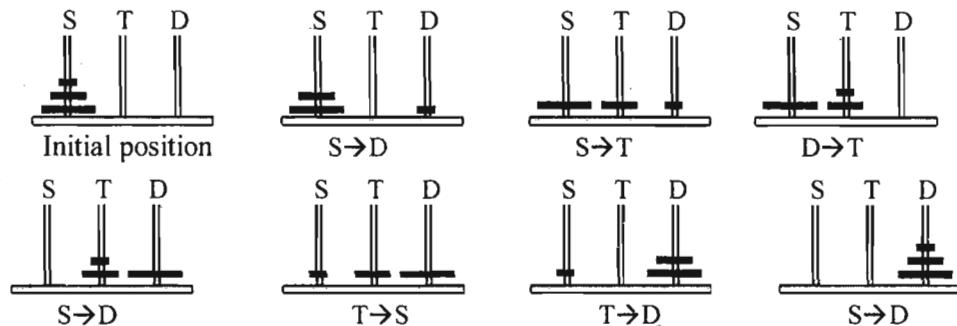
S→D means move the disk from pillar S to pillar D.

For n=2



- (i) Move disk 1 from pillar S to T (S→T)
- (ii) Move disk 2 from pillar S to D (S→D)
- (iii) Move disk 1 from pillar T to D (T→D)

For n=3



- (i) Move disk 1 from pillar S to D (S→D)
- (ii) Move disk 2 from pillar S to T (S→T)
- (iii) Move disk 1 from pillar D to T (D→T)
- (iv) Move disk 3 from pillar S to D (S→D)
- (v) Move disk 1 from pillar T to S (T→S)
- (vi) Move disk 2 from pillar T to D (T→D)
- (vii) Move disk 1 from pillar S to D (S→D)

These were the solutions for n=1, n=2, n=3. From these solutions we can observe that first we move n-1 disks from source pillar (S) to temporary pillar (T) and then move the largest disk to destination pillar(D). So the general solution for n disks can be written as-

1. Move upper n-1 disks from S to T.
2. Move n^{th} disk from S to D.
3. Move n-1 disks from T to D.

To move n-1 disks from S to T, we can use D as the temporary pillar, and to move n-1 disks from

T to D, we can use S as the temporary pillar.

Move disk from source to dest n=1

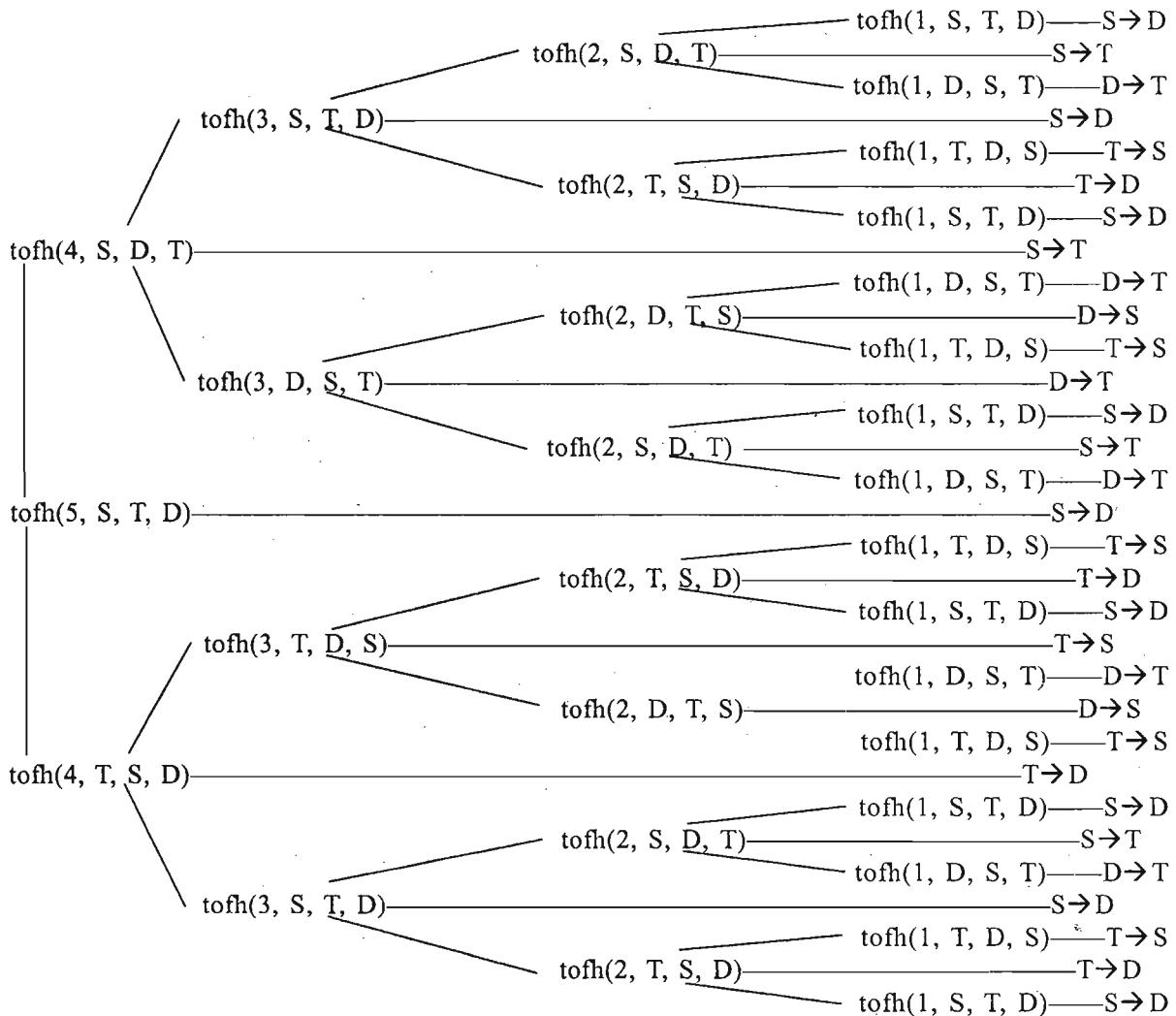
tofh(n, source, temp, dest) =

tofh(n-1, source, dest, temp)

Move nth disk from source to dest n>1

tofh(n-1, temp, source, dest)

```
/*P6.27 Program to solve Tower of Hanoi problem using recursion*/
#include<stdio.h>
main()
{
    char source='S',temp='T',dest='D';
    int ndisk;
    printf("Enter the number of disks : ");
    scanf("%d",&ndisk);
    printf("Sequence is :\n");
    tofh(ndisk,source,temp,dest);
}
tofh(int ndisk,char source,char temp,char dest)
{
    if(ndisk>0)
    {
        tofh(ndisk-1,source,dest,temp);
        printf("Move Disk %d %c->%c\n",ndisk,source,dest);
        tofh(ndisk-1,temp,source,dest);
    }
}/*End of tofh()*/
```



6.19.2 Advantages And Disadvantages Of Recursion

The use of recursion makes the code more compact and elegant. It simplifies the logic and hence makes the program easier to understand. But the code written using recursion is less efficient since recursion is a slow process because of many function calls involved in it.

Most problems with recursive solutions also have an equivalent non recursive(generally iterative) solutions. A non recursive solution increases performance while a recursive solution is simpler.

6.19.3 Local Variables In Recursion

We know each function has some local variables that exist only inside that function. When a function is called recursively, then for each call a new set of local variables is created(except static), their name is same but they are stored at different places and contain different values. These values are remembered by the compiler till the end of function call, so that these values are available in the unwinding phase.

6.20 Some Additional Problems

Problem 1

Write a program to convert a binary or octal number to a decimal number depending on user's choice.

```
/*P6.28 Program to convert a binary or octal number to a decimal number*/
#include<stdio.h>
main()
{
    int num,base,result;
    char choice;
    printf("Enter 'b' for binary and 'o' for octal : ");
    scanf("%c",&choice);
    printf("Enter the number : ");
    scanf("%d",&num);
    if(choice=='b')
        base=2;
    else
        base=8;
    result=func(num,base);
    printf("Decimal number is %d\n",result);
}
func(int n,int base)
{
    int rem,d,j=1,dec=0;
    while(n>0)
    {
        rem=n%10; /* taking last digit */
        d=rem*j;
        dec+=d;
        j*=base;
        n/=10; /* skipping last digit */
    }
    return dec;
}
```

Problem 2

Write a program to implement these formulae of permutations and combinations.

The formula for number of permutations of n objects taken r at a time-

$$p(n, r) = n! / (n-r)!$$

The formula for number of combinations of n objects taken r at a time is-

$$c(n, r) = n! / r! * (n-r)!$$

This can also be written as-

$$c(n, r) = p(n, r) / r!$$

```
/*P6.29 Program to find out permutations and combinations*/
#include<stdio.h>
long factorial(int);
```

```
long perm(int,int);
long comb(int,int);
main()
{
    int n,r;
    printf("Enter n : ");
    scanf("%d",&n);
    printf("Enter r : ");
    scanf("%d",&r);
    printf("Total combinations are : %ld\n",comb(n,r));
    printf("Total permutations are : %ld\n",perm(n,r));
}
long comb(int n,int r)
{
    long c;
    c=perm(n,r)/factorial(r);
    return c;
}
long perm(int n,int r)
{
    long p;
    p=factorial(n)/factorial(n-r);
    return p;
}
long factorial(int k)
{
    long fact=1;
    while( k>0 )
    {
        fact*=k;
        k--;
    }
    return fact;
}
```

Problem 3

Write a program to print Pascal's triangle.

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
```

```
/*P6.30 Program to print Pascal's triangle*/
#include<stdio.h>
long factorial(int);
long comb(int,int);
```

```

main()
{
    int i,j,k;
    printf("Enter number of rows for Pascal's triangle : ");
    scanf("%d",&k);
    for(i=0;i<k;i++)
    {
        for(j=0;j<=i;j++)
            printf("%5ld",comb(i,j));
        printf("\n");
    }
}
long comb(int n,int r)
{
    long c;
    c=factorial(n)/(factorial(r)*factorial(n-r)) ;
    return c;
}
long factorial(int k)
{
    long fact=1;
    while(k>0)
    {
        fact*=k;
        k--;
    }
    return fact;
}

```

Problem 4

Write a program to convert a decimal number to a roman number. The roman numbers corresponding to decimal numbers are as-

1 - i,	9 - ix	100 - c
4 - iv	10 - x	500 - d
5 - v	50 - l	1000 - m

Some examples are-

14	xiv	123	cxxiii	1009	mix
48	xxxxviii	772	dcclxxii	2856	mmdccclvi

```

/*P6.31 Program to convert a decimal number to roman number*/
#include<stdio.h>
int roman(int,int,char);
main()
{
    int num;
    printf("Enter a number : ");
    scanf("%d",&num);
    if(num>=1000)
        num=roman(num,1000,'m');

```

```

if(num>=500)
    num=roman(num, 500, 'd');
if(num>=100)
    num=roman(num, 100, 'c');
if(num>=50)
    num=roman(num, 50, 'l');
if(num>=10)
    num=roman(num, 10, 'x');
if(num>=5)
    num=roman(num, 5, 'v');
if(num>=1)
    roman(num, 1, 'i');
printf("\n");
}
int roman(int n,int k,char c)
{
    if(n==9)
    {
        printf("ix");
        return 0;
    }
    if(n==4)
    {
        printf("iv");
        return 0;
    }
    while(n>=k)
    {
        printf("%c",c);
        n=n-k;
    }
    return n;
}

```

Problem 5

Write a program that prints the reverse of a positive integer using recursion.

```

/*P6.32 Program to print the reverse of a positive integer*/
#include<stdio.h>
void reverse(long int n);
main()
{
    long int num;
    printf("Enter number : ");
    scanf("%ld",&num);
    reverse(num);
    printf("\n");
}
void reverse(long int n)
{
    int rem;

```

```

if(n==0)
    return;
else
{
    rem=n%10;
    printf("%d",rem);
    n/=10;
    reverse(n);
}
}

```

Problem 6

Write a program that reads a number and prints whether the given number is divisible by 11 or not by using the algorithm which states that a number is divisible by 11 if and only if the difference of the sums of digits at odd positions and even positions is either zero or divisible by 11.

```

/* P6.33 Program that tests whether a number is divisible by 11 or not
 */
#include<stdio.h>
void test(long int x);
main()
{
    long int num;
    printf("Enter the number to be tested : ");
    scanf("%ld",&num);
    test(num);
}
void test(long int n)
{
    int s1=0,s2=0,k;
    while(n>0)
    {
        s1+=n%10;
        n/=10;
        s2+=n%10;
        n/=10;
    }
    k=s1>s2?(s1-s2):(s2-s1);
    if(k>10)
        test(k);
    else if(k==0)
        printf("The number is divisible by 11\n");
    else
        printf("The number is not divisible by 11\n");
}

```

Problem 7

Write a program that uses a recursive function to convert a decimal number to (i) Binary (ii) Octal (iii) Hexadecimal depending on user's choice.

```
/* P6.34 Program to convert a decimal number to Binary, Octal or Hexadecimal
*/
#include<stdio.h>
void convert(int, int);
main()
{
    int num,base;
    int choice;
    while(1)
    {
        printf("1.Binary\n");
        printf("2.Octal\n");
        printf("3.Hexadecimal\n");
        printf("4.Exit\n");
        printf("Enter your choice : ");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1:
                base=2;
                break;
            case 2:
                base=8;
                break;
            case 3:
                base=16;
                break;
            case 4:
                exit(1);
            default:
                printf("Wrong choice\n");
                continue;      /*Takes to start of while loop*/
        }
        printf("Enter the number in decimal: ");
        scanf("%d",&num);
        convert(num,base);
        printf("\n");
    }
}
void convert (int num,int base)
{
    int rem;
    rem=num%base;
    num/=base;
    if (num>0)
        convert(num,base);
    if(rem<10)
        printf("%d",rem);
    else
        printf("%c",rem-10+'A');
}
```

Problem 8

Write a program to find out the prime factors of a number using both iterative and recursive methods. Prime factors of 56 are 2, 2, 2, 7, prime factors of 98 are 2, 7, 7, prime factors of 121 are 11, 11.

```
/*P6.35 Program to print the prime factors*/
#include<stdio.h>
void pfact(int num);
void rpfact(int n);
main()
{
    int num;
    printf("Enter a number : ");
    scanf("%d",&num);
    pfact(num);printf("\n");
    rpfact(num);printf("\n");
}
void pfact(int num)
{
    int i;
    for(i=2;num!=1;i++)
    {
        while(num%i==0)
        {
            printf("%d ",i);
            num=num/i;
        }
    }
}
void rpfact(int num)
{
    static int i=2;
    if(num==1)
        return;
    else
    {
        while(num%i==0)
        {
            printf("%d ",i);
            num=num/i;
        }
        i++;
        rpfact(num);
    }
}
```

Problem 9

Write a program to find out the sum of this series, both by iterative and recursive methods.

```
/*P6.36 Program to find out the sum of series*/
#include<stdio.h>
long int fact(int num);
double power(float x,int n);
double series(float x,int n);
double rseries(float x,int n);
main()
{
    float x;
    int n;
    printf("Enter x : ");
    scanf("%f",&x);
    printf("Enter number of terms : ");
    scanf("%d",&n);
    printf("Iterative   %lf\n",series(x,n));
    printf("Recursive   %lf\n",rseries(x,n));
}
long int fact(int num)
{
    int i;
    long int f=1;
    for(i=1;i<=num;i++)
        f=f*i;
    return f;
}
double power(float x,int n)
{
    int i;
    float p=1;
    for(i=1;i<=n;i++)
        p=p*x;
    return p;
}
double series(float x,int n)
{
    int i,j,sign=1;
    float term,sum;
    for(i=1;i<=n;i++)
    {
        sign=(i%2==0)?-1:1;
        j=2*i-1;
        term=sign*power(x,j)/fact(j);
        sum+=term;
    }
    return sum;
}
double rseries(float x,int n)
{
    int sign=1;
    float term,sum;
    if(n==0)
```

```

        sum=0;
else
{
    sign=(n%2==0)?-1:1;
    term=sign*power(x,2*n-1)/fact(2*n-1);
    sum=term+rseries(x,n-1);
}
return sum;
}

```

The i th term of the series is $x^{(2i-1)} / (2i-1)!$. In iterative function, we'll take a loop from $i=1$ to $i=n$ and add all the terms. To calculate the numerator and denominator of a term we'll use the functions power() and fact(). Since the terms have alternate positive and negative signs, so we'll take a variable sign, initialize it with 1 and multiply it with -1 on each iteration of the loop.

Problem 10

Write a program to find out the Least Common Multiple and Highest Common factor of two numbers recursively. The iterative method is given in Program P5.42.

```

/*P6.37 Program to find out the LCM and HCF of two numbers recursively*/
#include<stdio.h>
int m,n;
main()
{
    int x,y;
    printf("Enter two numbers : ");
    scanf("%d %d",&x,&y);
    printf("HCF of %d and %d is %d\n",x,y,hcf(x,y));
    m=x;n=y;
    printf("LCM of %d and %d is %d\n",x,y,lcm(x,y));
}
int hcf(int a,int b)
{
    if(a==b)
        return(b);
    else if(a<b)
        hcf(a,b-a);
    else
        hcf(a-b,b);
}
int lcm(int a,int b)
{
    if(a==b)
        return(b);
    else if(a<b)
        lcm(a+b,b);
    else
        lcm(a,b+n);
}

```

Exercise

Assume stdio.h is included in all programs.

(1) void func(void);
main()
{
 printf("Lucknow\n");
 goto ab;
}
void func(void)
{
ab:
 printf("Bareilly\n");
}

(2) main()
{
 int i=9;
 if(i==9)
 {
 int i=25;
 }
 printf("i=%d",i);
}

(3) void func(int a,int b);
main()
{
 int x;
 x=func(2,3);
}
void func(int a,int b)
{
 int s;
 s=a+b;
 return;
}

(4) main()
{
 int x=5;
 x=func();
 printf("%d\n",x);
}
int func(int a)
{
 a=a*2;
}

(5) main()
{

```
static int x=5;
if(x>0)
{
    printf("%d    ",x);
    x--;
    main();
}
}

(6) main()
{
    int s;
    s=func(2,3);
    printf("%d\n",s);
}
int func(int a,int b,int c)
{
    c=4;
    return (a+b+c);
}

(7) main()
{
    int s;
    s=func(2,3,6);
    printf("%d\n",s);
}
int func(int a,int b)
{
    return (a+b);
}

(8) func(int x,int y);
main()
{
    int p=func(5,6);
    printf("%d",p);
}
func(int x,int y)
{
    int x=2;
    return x*y;
}

(9) main()
{
    int var1=12,var2=35;
    printf("%d",max(var1,var2));
}
int max(int x,int y)
{
```

```
x>y?return x:return y;
}

(10)main( )
{
    int n=5;
    printf("%d\n", func(n));
}
func(int n)
{
    return(n+sqr(n-2)+cube(n-3));
}
sqr(int x)
{
    return(x*x);
}
cube(int x)
{
    return (x*x*x);
}

(11)main( )
{
    int func(int a,int b);
    {
        return (a+b);
    }
    int c;
    c=func(3,5);
    printf("%d",c);
}

(12)main( )
{
    int x=5;
    func1(x);
}
func1(int a)
{
    printf("Value of a = %d\n",a);
    if(a>0)
        func2(--a);
}
func2(int b)
{
    printf("Value of b = %d\n",b);
    if(b>0)
        func1(--b);
}

(13)void disp(int, int);
main( )
{
    int x=15;
    float y=290.5;
    disp(x,y);
}
```

```

void disp(int a,int b)
{
    printf("%d %d\n",a,b);
}

(14)void func(void );
main()
{
    int i=5;
    for(i=i+1;i<8;i++)
        func();
}
void func(void)
{
    int j;
    for(j=1;j<3;j++)
        printf("%d\t",++j);
}

(15)main()
{
    int i=10,k;
    for ( ; ; )
    {
        k=mult(i)
        if(--i<5)
            break;
    }
    printf("k=%d\n",k);
}
mult(int j)
{
    j*=j;
    return(j);
}

(16)main()
{
    int i=2,j=3;
    printf("%d\n",func(i,j));
}
func(int a,int b)
{
    a=a-5;
    b++;
    return(!a+-b);
}

(17)main()
{
    int x;

```

```
x=func(2,3,4);
printf("%d\n",x);
}
func(int a,int b,int c)
{
    return(a,b,c);
}

(18)void func(int a,int b);
main()
{
    int i=5,j=10;
    func(i/2,j%3);
}
void func(int a,int b)
{
    a=a/2;
    b--;
    printf("%d\t",a+b);
}

(19)int a=5;
void func(void );
main()
{
    func();
    printf("%d\n", a);
}
void func(void )
{
    int a=2;
    printf("%d\t",a);
}

(20)main()
{
    int a=2,b=5;
    a=func(a+b,a-b);
    printf("%d\n",a);
}
func(int x,int y)
{ return x+y, x-y; }

(21)main()
{
    int i=0,k=3;
    i+=func(k);
    i+=func(k);
    i+=func(k);
    printf("%d\n",i);
}
```

```

func(int k)
{
    static int m=2;
    m=m+k;
    return m;
}

(22)main()
{
    int n=8;
    printf("%d\n", func(n));
}
func(int n)
{
    if(n==0)
        return 0;
    else
        return(n+func(n-1));
}

(23)main()
{
    int a=2,b=6;
    printf("%d\t", func1(a,b));
    printf("%d\n", func2(a,b));
}
func1(int a,int b)
{
    int i,s=0;
    for(i=a;i<=b;i++)
    {
        s=s+i*i;
    }
    return s;
}
func2(int a,int b)
{
    int s;
    if(a<b)
        s=a*a+func2(a+1,b);
    else
        s=a*a;
}

(24)main()
{
    int a=7,b=8;
    printf("%d\n", func(a,b));
}
func(int x,int y)
{

```

```

        if(x==0)
            return y;
        else
            func(--x, ++y) ;
    }

(25)main()
{
    int x=55,y=17;
    printf("%d\n",func(x,y));
}
func(int x,int y)
{
    int q=0;
    if(x<y)
        return 0;
    else
        return func(x-y,y)+1;
}

(26)main()
{
    func1(6);
    printf("\n");
    func2(6);
}
func1(int x)
{
    printf("%d ",x);
    if(x>2)
        func1(--x);
}
func2(int x)
{
    if(x>2)
        func2(--x);
    printf("%d ",x);
}

```

Programming Exercise

1. Write a function cubesum() that accepts an integer and returns the sum of the cubes of individual digits of that number. Use this program to print Armstrong numbers in a given range. (See program P5.24)
2. Write a function that inputs a number and returns the product of digits of that number until the product is reduced to one digit (like P5.25). The number of times digits need to be multiplied to reach one digit is called the persistence of the number. Write another function pers()to input a number and return its persistence.

For example: 86 → 48 → 32 → 6 (persistence 3)
 341 → 12 → 2 (persistence 2)

3. Write a function sumdiv() that finds the sum of divisors of a number. (Divisors of a number are those numbers by which the number is divisible). For example divisors of 36 are 1, 2, 3, 4, 6, 9, 18. A number is called perfect if the sum of divisors of that number is equal to the number. For example 28 is a perfect number, since $1+2+4+7+14=28$. Write a program to print all the perfect numbers in a given range.
4. Write a program to find the sum of this series upto n terms.

$$1 + 1 / 2 + 1 / 4 + 1 / 9 + 1 / 16 + \dots$$
5. Write a program to find out the roots of quadratic equation $ax^2 + bx + c = 0$.
6. Write a function that accepts a character in lower case and returns its upper case equivalent.
7. Write a program mult() that accepts two integers and returns their product. Don't use the * operator. Similarly make two more functions quo() and rem() that accept two integers and return the quotient and remainder respectively, without the use of / and % operators.
8. Write a recursive function to find the sum of digits of a number.

$$\text{sumd}(n) = \begin{cases} 0 & n=0 \\ (n \% 10) + \text{sumd}(n/10) & n>0 \end{cases}$$

9. Write a recursive function to find the combinations of n objects taken r at a time

$$\text{rcomb}(n, r) = \begin{cases} 1 & r = 0 \text{ or } n = r \\ \text{rcomb}(n-1, r) + \text{rcomb}(n-1, r-1) & \text{otherwise} \end{cases}$$

10. Write a program to print number in words using recursion. For example if we have a number 31246, then the program should print
 Three one Two four Six

Answers

- (1) We can't use goto between functions, so this program will show an error stating that ab is an undefined label.
- (2) i = 9
- (3) The function is declared as void so it doesn't return any value, and so the function call can't be used in any expression.
- (4) If no value is returned through return statement, and the function is not declared of type void then garbage value is returned, so the variable x will have garbage value.
- (5) 5 4 3 2 1
 Here the definition of main() is written recursively.
- (6) 9
 Since function declaration is absent, so garbage value is passed in variable c, inside the function variable c is assigned the value 4. If the function is declared as-


```
int func( int, int , int);
```

 then we'll get an error stating that there are too few parameters in call to func().

(7) 5

If the function is declared as-

```
int func(int, int);
```

then we'll get an error stating that there is an extra parameter in call to func().

(8) Error : Multiple declaration for x in func().

(9) Error : Expression syntax

The operands of conditional operator should be expressions, but return x and return y are not expressions.

(10) 22

We can have a function call in return statement of another call.

(11) We'll get errors because a function definition can't be written inside definition of another function.

(12) Value of a = 5

Value of b = 4

Value of a = 3

Value of b = 2

Value of a = 1

Value of b = 0

This is an example of indirect recursion.

(13) 15 290

(14) 2 2

(15) k = 25

(16) 3

(17) 4

Here expression a, b, c is considered as an expression with comma operator, so the value of expression is rightmost value, hence value of c gets returned.

(18) 1

(19) 2 5

In func(), the local variable will be accessed, because whenever there is a conflict between the local and global variables, the local variable gets the priority.

(20) 10

(21) 24

(22) 36

func() finds the sum of series 1+2+3...+n

(23) 90 90

Both functions func1() and func2() return the sum of squares of numbers from a to b, func1() uses iterative method while func2() uses recursive method.

(24) 15

func() is a recursive function for adding two integers.

(25) 3

func() is a recursive function to find out the quotient.

(26) 6 5 4 3 2

2 2 3 4 5

Chapter 7

Arrays

The variables that we have used till now are capable of storing only one value at a time. Consider a situation when we want to store and display the age of 100 employees. For this we have to do the following-

1. Declare 100 different variables to store the age of employees.
2. Assign a value to each variable.
3. Display the value of each variable.

Although we can perform our task by the above three steps but just imagine how difficult it would be to handle so many variables in the program and the program would become very lengthy. The concept of arrays is useful in these types of situations where we can group similar type of data items.

An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the elements may be any valid data type like char, int or float. The elements of array share the same variable name but each element has a different index number known as subscript.

For the above problem we can take an array variable age[100] of type int. The size of this array variable is 100 so it is capable of storing 100 integer values. The individual elements of this array are-

age[0], age[1], age[2], age[3], age[4],age[98], age[99]

In C the subscripts start from zero, so age[0] is the first element, age[1] is the second element of array and so on.

Arrays can be single dimensional or multidimensional. The number of subscripts determines the dimension of array. A one-dimensional array has one subscript, two dimensional array has two subscripts and so on. The one-dimensional arrays are known as vectors and two-dimensional arrays are known as matrices. In this chapter first we will study about single dimensional arrays and then move on to multi dimensional arrays.

7.1 One Dimensional Array

7.1.1 Declaration of 1-D Array

Like other simple variables, arrays should also be declared before they are used in the program.

The syntax for declaration of an array is-

`data_type array_name[size];`

Here array_name denotes the name of the array and it can be any valid C identifier, data_type is the data type of the elements of array. The size of the array specifies the number of elements that can be stored in the array. It may be a positive integer constant or constant integer expression.

Here are some examples of array declarations-

```
int age[100];
float sal[15];
char grade[20];
```

Here age is an integer type array, which can store 100 elements of integer type. The array sal is a floating type array of size 15, can hold float values and third one is a character type array of size 20, can hold characters. The individual elements of the above arrays are-

```
age[0], age[1], age[2], .....age[99]
sal[0], sal[1], sal[2], .....sal[14]
grade[0], grade[1], grade[2], .....grade[19]
```

When the array is declared, the compiler allocates space in memory sufficient to hold all the elements of the array, so the compiler should know the size of array at the compile time. Hence we can't use variables for specifying the size of array in the declaration. The symbolic constants can be used to specify the size of array. For example-

```
#define SIZE 10
main()
{
    int size=15;
    float sal[SIZE];      /*Valid*/
    int marks[size];      /*Not valid*/
    .....
}
```

The use of symbolic constant to specify the size of array makes it convenient to modify the program if the size of array is to be changed later, because the size has to be changed only at one place i.e. in the #define directive.

7.1.2 Accessing 1-D Array Elements

The elements of an array can be accessed by specifying the array name followed by subscript in brackets. In C, the array subscripts start from 0. Hence if there is an array of size 5 then the valid subscripts will be from 0 to 4. The last valid subscript is one less than the size of the array. This last valid subscript is sometimes known as the upper bound of the array and 0 is known as the lower bound of the array.

Let us take an array-

```
int arr[5];           /*Size of array arr is 5, can hold five integer elements*/
```

The elements of this array are-

```
arr[0], arr[1], arr[2], arr[3], arr[4]
```

Here 0 is the lower bound and 4 is the upper bound of the array.

The subscript can be any expression that yields an integer value. It can be any integer constant, integer variable, integer expression or return value(int) from a function call. For example, if i and j are integer variables then these are some valid subscripted array elements-

```
arr[3]
arr[i]
arr[i+j]
```

arr[2*j]

arr[i++]

A subscripted array element is treated as any other variable in the program. We can store values in them, print their values or perform any operation that is valid for any simple variable of the same data type. For example if arr[5] and sal[10] are two arrays then these are valid statements-

```
int arr[5];
float sal[10];
int i;
scanf("%d", &arr[1]);           /*input value into arr[1]*/
printf("%f", sal[3]);          /*print value of sal[3]*/
arr[4]=25;                     /*assign a value to arr[4] */
arr[4]++;                      /*Increment the value of arr[4] by 1*/
sal[5]+=200;                   /*Add 200 to sal[5]*/
sum=arr[0]+arr[1]+arr[2]+arr[3]+arr[4]; /*Add all the values of array
arr[5]*/
i=2;
scanf("%f", &sal[i]);          /*Input value into sal[2]*/
printf("%f", sal[i]);          /*Print value of sal[2]*/
printf("%f", sal[i++]);/*Print value of sal[2] and increment the value of i*/
```

In C there is no check on bounds of the array. For example if we have an array arr[5], the valid subscripts are only 0, 1, 2, 3, 4 and if someone tries to access elements beyond these subscripts like arr[5], arr[10] or arr[-1], the compiler will not show any error message but this may lead to runtime errors, which can be very difficult to debug. So it is the responsibility of programmer to provide array bounds checking wherever needed.

7.1.3 Processing 1-D Arrays

For processing arrays we generally use a for loop and the loop variable is used at the place of subscript. The initial value of loop variable is taken 0 since array subscripts start from zero. The loop variable is increased by 1 each time so that we can access and process the next element in the array. The total number of passes in the loop will be equal to the number of elements in the array and in each pass we will process one element.

Suppose arr[10] is an array of int type-

(i) Reading values in arr[10]

```
for( i = 0; i < 10; i++)
    scanf("%d", &arr[i]);
```

(ii) Displaying values of arr[10]

```
for( i = 0; i < 10; i++)
    printf("%d ", arr[i]);
```

(iii) Adding all the elements of arr[10]

```
sum = 0;
for( i = 0; i < 10; i++)
    sum+=arr[i];
```

```
/*P7.1 Program to input values into an array and display them*/
#include<stdio.h>
main()
{
    int arr[5],i;
    for(i=0;i<5;i++)
    {
        printf("Enter the value for arr[%d] : ",i);
        scanf("%d",&arr[i]);
    }
    printf("The array elements are : \n");
    for(i=0;i<5;i++)
        printf("%d\t",arr[i]);
    printf("\n");
}
```

Output:

```
Enter the value for arr[0] : 12
Enter the value for arr[1] : 45
Enter the value for arr[2] : 59
Enter the value for arr[3] : 98
Enter the value for arr[4] : 21
The array elements are :
12 45 59 98 21
```

```
/*P7.2 Program to add the elements of an array*/
#include<stdio.h>
main()
{
    int arr[10],i,sum=0;
    for(i=0;i<10;i++)
    {
        printf("Enter the value for arr[%d] : ",i);
        scanf("%d",&arr[i]);
        sum+=arr[i];
    }
    printf("Sum = %d\n",sum);
}
```

```
/*P7.3 Program to count the even and odd numbers in a array*/
#include<stdio.h>
#define SIZE 10
main()
{
    int arr[SIZE],i,even=0,odd=0;
    for(i=0;i<SIZE;i++)
    {
        printf("Enter the value for arr[%d] : ",i);
        scanf("%d",&arr[i]);
        if(arr[i]%2==0)
```

```

        even++;
else
    odd++;
}
printf("Even numbers = %d, Odd numbers = %d\n", even, odd );
}

```

7.1.4 Initialization of 1-D Array

After declaration, the elements of a local array have garbage value while the elements of global and static arrays are automatically initialized to zero. We can explicitly initialize arrays at the time of declaration. The syntax for initialization of an array is-

```
data_type array_name[size]={value1, value2,.....,valueN};
```

Here array_name is the name of the array variable, size is the size of the array and value1, value2,valueN are the constant values known as initializers, which are assigned to the array elements one after another. These values are separated by commas and there is a semicolon after the ending braces. For example-

```
int marks[5] = {50, 85, 70, 65, 95};
```

The values of the array elements after this initialization are-

```
marks[0] : 50, marks[1] : 85, marks[2] : 70, marks[3] : 65, marks[4] : 95
```

While initializing 1-D arrays, it is optional to specify the size of the array. If the size is omitted during initialization then the compiler assumes the size of array equal to the number of initializers. For example-

```
int marks[] = { 99, 78, 50, 45, 67, 89};
```

```
float sal[] = { 25.5, 38.5, 24.7 };
```

Here the size of array marks is assumed to be 6 and that of sal is assumed to be 3.

If during initialization the number of initializers is less than the size of array then, all the remaining elements of array are assigned value zero. For example-

```
int marks[5] = { 99, 78};
```

Here the size of array is 5 while there are only 2 initializers. After this initialization the value of the elements are as-

```
marks[0] : 99, marks[1] : 78, marks[2] : 0, marks[3] : 0, marks[4] : 0
```

So if we initialize an array like this-

```
int arr[100] = {0};
```

then all the elements of arr will be initialized to zero.

If the number of initializers is more than the size given in brackets then compiler will show an error. For example-

```
int arr[5] = {1, 2, 3, 4, 5, 6, 7, 8}; /*Error*/
```

We can't copy all the elements of an array to another array by simply assigning it to the other array. For example if we have two arrays a[5] and b[5] then

```
int a[5] = {1, 2, 3, 4, 5};
int b[5];
b = a; /*Not valid*/
```

We'll have to copy all the elements of array one by one, using a for loop.

```
for( i = 0; i < 5; i++)
    b[i] = a[i];
```

In the following program we'll find out the maximum and minimum number in an integer array.

```
/*P7.4 Program to find the maximum and minimum number in an array*/
#include<stdio.h>
main()
{
    int i,j,arr[10]={2,5,4,1,8,9,11,6,3,7};
    int min,max;
    min=max=arr[0];
    for(i=1;i<10;i++)
    {
        if(arr[i]<min)
            min=arr[i];
        if(arr[i]>max)
            max=arr[i];
    }
    printf("Minimum = %d, Maximum = %d\n",min,max);
}
```

Output:

Minimum = 1, Maximum = 11

We have taken the value of first element as the initial value of min and max. Inside the for loop, we'll start comparing from second element onwards so this time we have started the loop from 1 instead of 0.

The following program will reverse the elements of an array.

```
/*P7.5 Program to reverse the elements of an array*/
#include<stdio.h>
main()
{
    int i,j,temp,arr[10]={1,2,3,4,5,6,7,8,9,10};
    for(i=0,j=9;i<j;i++,j--)
    {
        temp=arr[i];
        arr[i]=arr[j];
        arr[j]=temp;
    }
    printf("After reversing the array is : ");
    for(i=0;i<10;i++)
        printf("%d ",arr[i]);
    printf("\n");
}
```

Output:

After reversing the array is : 10 9 8 7 6 5 4 3 2 1

In the for loop we have used comma operator and taken two variable i and j. The variable i is initialized

with the lower bound and j is initialized with upper bound. After each pass of the loop, i is incremented while j is decremented. Inside the loop, a[i] is exchanged with a[j]. So a[0] will be exchanged with a[9], a[1] with a[8], a[2] with a[7] and so on.

The next program prints the binary equivalent of a decimal number. The process of obtaining a binary number from decimal number 29 is given below:

	Quotient	Remainder		
29/2	14	1	a[0]	MSB
14/2	7	0	a[1]	
7/2	3	1	a[2]	
3/2	1	1	a[3]	
1/2	0	1	a[4]	LSB

We will store the remainders in an array, and then at last the array is printed in reverse order to get the binary number.

```
/*P7.6 Program to convert a decimal number to binary number */
#include<stdio.h>
main()
{
    int num,rem,arr[15],i,j;
    printf("Enter a decimal number : ");
    scanf("%d",&num);
    i=0;
    while(num>0)
    {
        arr[i]=num%2; /*store the remainder in array*/
        num/=2;
        i++;
    }
    printf("Binary number is : ");
    for(j=i-1;j>=0;j--)/*print the array backwards*/
        printf("%d",arr[j]);
    printf("\n");
}
```

Output:

Enter a decimal number : 29

Binary number is : 11101

The next program searches for a particular item in the array.

```
/*P7.7 Program to search for an item in the array*/
#include<stdio.h>
#define SIZE 10
main( )
{
    int i,arr[SIZE]={23,12,56,98,76,14,65,11,19,45};
    int item;
    printf("Enter the item to be searched : ");
```

```

    scanf("%d",&item);
    for(i=0;i<SIZE;i++)
    {
        if(item==arr[i])
        {
            printf("%d found at position %d\n",item,i+1);
            break;
        }
    }
    if(i==SIZE)
        printf("Item %d not found in array\n", item);
}

```

The item to be searched is compared with each element of the array, if the element is found, for loop is terminated through break statement. The control can come out of the loop in two cases, first if the item is found and loop is terminated through break and second if item is not found and the loop is fully executed. In the second case the value of i would be equal to SIZE. Here we are searching the item sequentially so this search is known as linear or sequential search.

7.1.5 1-D Arrays And Functions

7.1.5.1 Passing Individual Array Elements to a Function

We know that an array element is treated as any other simple variable in the program. So we can pass individual array elements as arguments to a function like other simple variables.

```

/*P7.8 Program to pass array elements to a function*/
#include<stdio.h>
main()
{
    int arr[10],i;
    printf("Enter the array elements : ");
    for(i=0;i<10;i++)
    {
        scanf("%d",&arr[i]);
        check(arr[i]);
    }
}
check(int num)
{
    if(num%2==0)
        printf("%d is even\n",num);
    else
        printf("%d is odd\n",num);
}

```

7.1.5.2 Passing whole 1-D Array to a Function

We can pass whole array as an actual argument to a function. The corresponding formal argument should be declared as an array variable of the same data type.

```

main()
{

```

```

int arr[10]
.....
.....
func(arr);/*In function call, array name is specified without brackets*/
}
func(int val[10])
{
.....
.....
}

```

It is optional to specify the size of the array in the formal argument, for example we may write the function definition as-

```

func(int val[])
{
.....
.....
}

```

We have studied that changes made in formal arguments do not affect the actual arguments, but this is not the case while passing an array to a function. The mechanism of passing an array to a function is quite different from that of passing a simple variable. We have studied earlier that in the case of simple variables, the called function creates a copy of the variable and works on it, so any changes made in the function do not affect the original variable. When an array is passed as an actual argument, the called function actually gets access to the original array and works on it, so any changes made inside the function affect the original array. Here is a program in which an array is passed to a function.

```

/*P7.9 Program to understand the effect of passing an array to a function*/
#include<stdio.h>
main()
{
    int i,arr[6]={1,2,3,4,5,6};
    func(arr);
    printf("Contents of array are now : ");
    for(i=0;i<6;i++)
        printf("%d ",arr[i]);
    printf("\n");
}
func(int val[])
{
    int sum=0,i;
    for(i=0;i<6;i++)
    {
        val[i]=val[i]*val[i];
        sum+=val[i];
    }
    printf("The sum of squares = %d\n",sum);
}

```

Output:

The sum of squares = 91

The contents of array are now : 1 4 9 16 25 36

Here we can see that the changes made to the array inside the called function are reflected in the calling function. The name of the formal argument is different but it refers to the original array.

Since it is not necessary to specify the size of array in the function definition, so we can write general functions that can work on arrays of same type but different sizes. For example in the next program, the function add() is capable of adding the elements of any size of an integer array.

```
/*P7.10 Program that uses a general function that works on arrays of
different sizes*/
#include<stdio.h>
main()
{
    int a[5]={2,4,6,8,10};
    int b[8]={1,3,5,7,9,11,13,15};
    int c[10]={1,2,3,4,5,6,7,8,9,10};
    printf("Sum of elements of array a : %d\n",add(a,5));
    printf("Sum of elements of array b : %d\n",add(b,8));
    printf("Sum of elements of array c : %d\n",add(c,10));
}
add(int arr[],int n)
{
    int i,sum=0;
    for(i=0;i<n;i++)
        sum+=arr[i];
    return sum;
}
```

Output:

```
Sum of elements of array a : 30
Sum of elements of array b : 64
Sum of elements of array c : 55
```

7.2 Two Dimensional Array

7.2.1 Declaration and Accessing Individual Elements of a 2-D array

The syntax of declaration of a 2-D array is similar to that of 1-D arrays, but here we have two subscripts.

```
data_type array_name[rowsize][columnsize];
```

Here rowsize specifies the number of rows and columnsize represents the number of columns in the array. The total number of elements in the array are rowsize * columnsize. For example-

```
int arr[4][5];
```

Here arr is a 2-D array with 4 rows and 5 columns. The individual elements of this array can be accessed by applying two subscripts, where the first subscript denotes the row number and the second subscript denotes the column number. The starting element of this array is arr[0][0] and the last element is arr[3][4]. The total number of elements in this array is $4 \times 5 = 20$.

	Col 0	Col 1	Col 2	Col 3	Col 4
Row 0	arr[0][0]	arr[0][1]	arr[0][2]	arr[0][3]	arr[0][4]
Row 1	arr[1][0]	arr[1][1]	arr[1][2]	arr[1][3]	arr[1][4]
Row 2	arr[2][0]	arr[2][1]	arr[2][2]	arr[2][3]	arr[2][4]
Row 3	arr[3][0]	arr[3][1]	arr[3][2]	arr[3][3]	arr[3][4]

7.2.2 Processing 2-D Arrays

For processing 2-D arrays, we use two nested for loops. The outer for loop corresponds to the row and the inner for loop corresponds to the column.

```
int arr[4][5];
```

(i) **Reading values in arr**

```
for( i = 0; i < 4; i++ )
    for( j = 0; j < 5; j++ )
        scanf("%d", &arr[i][j]);
```

(ii) **Displaying values of arr**

```
for( i = 0; i < 4; i++ )
    for( j = 0; j < 5; j++ )
        printf(" %d ", arr[i][j]);
```

This will print all the elements in the same line. If we want to print the elements of different rows on different lines then we can write like this-

```
for(i=0;i<4;i++)
{
    for(j=0;j<5;j++)
        printf(" %d ",arr[i][j]);
    printf("\n");
}
```

Here the printf("\n") statement causes the next row to begin from a new line.

```
/*P7.11 Program to input and display a matrix*/
#define ROW 3
#define COL 4
#include<stdio.h>
main()
{
    int mat[ROW][COL],i,j;
    printf("Enter the elements of matrix(%dx%d) row-wise :\n",ROW,COL);
    for(i=0;i<ROW;i++)
        for(j=0;j<COL;j++)
            scanf("%d",&mat[i][j]);

    printf("The matrix that you have entered is :\n");
    for(i=0;i<ROW;i++)
    {
        for(j=0;j<COL;j++)
```

```

        printf("%5d", mat[i][j]);
    printf("\n");
}
printf("\n");
}

```

Output:

Enter the elements of matrix(3x4) row-wise -

```

2 3 4 7
8 5 1 9
1 8 2 5

```

The matrix that you have entered is -

```

2 3 4 7
8 5 1 9
1 8 2 5

```

7.2.3 Initialization of 2-D Arrays

2-D arrays can be initialized in a way similar to that of 1-D arrays. For example-

```
int mat[4][3] = { 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22};
```

These values are assigned to the elements row-wise, so the values of elements after this initialization are-

mat[0][0] : 11	mat[0][1] : 12	mat[0][2] : 13
mat[1][0] : 14	mat[1][1] : 15	mat[1][2] : 16
mat[2][0] : 17	mat[2][1] : 18	mat[2][2] : 19
mat[3][0] : 20	mat[3][1] : 21	mat[3][2] : 22

While initializing we can group the elements row-wise using inner braces. For example-

```
int mat[4][3]={ {11,12,13}, {14,15,16}, {17,18,19}, {20,21,22} };
int mat[4][3]={
    {11,12,13}, /* Row 0 */
    {14,15,16}, /* Row 1 */
    {17,18,19}, /* Row 2 */
    {20,21,22} /* Row 3 */
};
```

Here the values in the first inner braces will be the values of Row 0, values in the second inner braces will be values of Row 1 and so on.

Now consider this array initialization-

```
int mat[4][3]={
    {11}, /*Row 0*/
    {12,13}, /*Row 1*/
    {14,15,16}, /*Row 2*/
    {17} /*Row 3*/
};
```

The remaining elements in each row will be assigned values 0, so the values of elements will be-

mat[0][0] : 11	mat[0][1] : 0	mat[0][2] : 0
mat[1][0] : 12	mat[1][1] : 13	mat[1][2] : 0
mat[2][0] : 14	mat[2][1] : 15	mat[2][2] : 16
mat[3][0] : 17	mat[3][1] : 0	mat[3][2] : 0

In 2-D arrays it is optional to specify the first dimension but the second dimension should always be present. For example-

```
int mat[][] = {
    {1, 10},
    {2, 20, 200},
    {3},
    {4, 40, 400}
};
```

Here first dimension is taken 4 since there are 4 rows in initialization list.

A 2-D array is also known as a matrix. The next program adds two matrices; the order of both the matrices should be same.

```
/*P7.12 Program for addition of two matrices.*/
#define ROW 3
#define COL 4
#include<stdio.h>
main()
{
    int i,j,mat1[ROW][COL],mat2[ROW][COL],mat3[ROW][COL];
    printf("Enter matrix mat1(%dx%d) row-wise :\n",ROW,COL);
    for(i=0;i<ROW;i++)
        for(j=0;j<COL;j++)
            scanf("%d",&mat1[i][j]);

    printf("Enter matrix mat2(%dx%d) row-wise :\n",ROW,COL);
    for(i=0;i<ROW;i++)
        for(j=0;j<COL;j++)
            scanf("%d",&mat2[i][j]);

    /*Addition*/
    for(i=0;i<ROW;i++)
        for(j=0;j<COL;j++)
            mat3[i][j]=mat1[i][j]+mat2[i][j];

    printf("The resultant matrix mat3 is :\n");
    for(i=0;i<ROW;i++)
    {
        for(j=0;j<COL;j++)
            printf("%5d",mat3[i][j]);
        printf("\n");
    }
}
```

Output:

Enter elements of first matrix mat1(3x4) row-wise -

1 2 8 4

5 6 7 8

3 2 1 4

Enter elements of second matrix mat2(3x4) row-wise -

2 5 4 2

1 5 2 6

9 4 7 2

The resultant matrix mat3 is -

3 7 12 6

6 11 9 14

12 6 8 6

Now we'll write a program to multiply two matrices. Multiplication of matrices requires that the number of columns in first matrix should be equal to the number of rows in second matrix. Each row of first matrix is multiplied with the column of second matrix then added to get the element of resultant matrix. If we multiply two matrices of order mxn and nxp then the multiplied matrix will be of order mpx. For example-

$$A_{2 \times 2} = \begin{bmatrix} 4 & 5 \\ 3 & 2 \end{bmatrix} \quad B_{2 \times 3} = \begin{bmatrix} 2 & 6 & 3 \\ -3 & 2 & 4 \end{bmatrix}$$

$$C_{2 \times 3} = \begin{bmatrix} 4*2 + 5*(-3) & 4*6 + 5*2 & 4*3 + 5*4 \\ 3*2 + 2*(-3) & 3*6 + 2*2 & 3*3 + 2*4 \end{bmatrix} = \begin{bmatrix} -7 & 34 & 32 \\ 0 & 22 & 17 \end{bmatrix}$$

```
/*P7.13 Program for multiplication of two matrices*/
#include<stdio.h>
#define ROW1 3
#define COL1 4
#define ROW2 COL1
#define COL2 2
main()
{
    int mat1[ROW1][COL1], mat2[ROW2][COL2], mat3[ROW1][COL2];
    int i, j, k;

    printf("Enter matrix mat1(%dx%d)row-wise : \n", ROW1, COL1);
    for(i=0; i<ROW1; i++)
        for(j=0; j<COL1; j++)
            scanf("%d", &mat1[i][j]);

    printf("Enter matrix mat2(%dx%d)row-wise : \n", ROW2, COL2);
```

```

for(i=0;i<ROW2;i++)
    for(j=0;j<COL2;j++)
        scanf("%d",&mat2[i][j]);

/*Multiplication */
for(i=0;i<ROW1;i++)
    for(j=0;j<COL2;j++)
    {
        mat3[i][j]=0;
        for(k=0;k<COL1;k++)
            mat3[i][j]+=mat1[i][k]*mat2[k][j];
    }

printf("The Resultant matrix mat3 is :\n");
for(i=0;i<ROW1;i++)
{
    for(j=0;j<COL2;j++)
        printf("%5d",mat3[i][j]);
    printf("\n");
}
}
}

```

Output:

Enter matrix mat1(3x4)row-wise -

```

2 1 4 3
5 2 7 1
3 1 4 2

```

Enter matrix mat2(4x2)row-wise -

```

1 2
3 4
2 5
6 2

```

The Resultant matrix mat3 is -

```

31 34
31 55
26 34

```

The next program finds out the transpose of a matrix. Transpose matrix is defined as the matrix that is obtained by interchanging the rows and columns of a matrix. If a matrix is of $m \times n$ order then its transpose matrix will be of order $n \times m$.

```

/*P7.14 Program to find the transpose of matrix.*/
#include<stdio.h>
#define ROW 3
#define COL 4
main()
{
    int mat1[ROW][COL],mat2[COL][ROW],i,j;
    printf("Enter matrix mat1(%dx%d) row-wise : \n",ROW,COL);

```

```

for(i=0;i<ROW;i++)
    for(j=0;j<COL;j++)
        scanf("%d",&mat1[i][j]);

for(i=0;i<COL;i++)
    for(j=0;j<ROW;j++)
        mat2[i][j]=mat1[j][i];

printf("Transpose of matrix is:\n");
for(i=0;i<COL;i++)
{
    for(j=0;j<ROW;j++)
        printf("%5d",mat2[i][j]);
    printf("\n");
}
}

```

Output:

Enter matrix mat1(3x4) row-wise-

3 2 1 5
6 5 8 2
9 3 4 1

Transpose of matrix is-

3 6 9
2 5 3
1 8 4
5 2 1

7.3 Arrays With More Than Two Dimensions

We'll just give a brief overview of three-d arrays. We can think of a three-d array as an array of 2-D arrays. For example if we have an array-

int arr[2][4][3] ;

We can think of this as an array which consists of two 2-D arrays and each of those 2-D array has 4 rows and 3 columns.

$$\begin{array}{c}
 \text{[0]} \quad \left(\begin{matrix} [0][0] & [0][1] & [0][2] \\ [1][0] & [1][1] & [1][2] \\ [2][0] & [2][1] & [2][2] \\ [3][0] & [3][1] & [3][2] \end{matrix} \right) \quad \text{[1]} \quad \left(\begin{matrix} [0][0] & [0][1] & [0][2] \\ [1][0] & [1][1] & [1][2] \\ [2][0] & [2][1] & [2][2] \\ [3][0] & [3][1] & [3][2] \end{matrix} \right)
 \end{array}$$

The individual elements are-

arr[0][0][0], arr[0][0][1], arr[0][0][2], arr[0][1][0].....arr[0][3][2]
arr[1][0][0], arr[1][0][1], arr[1][0][2], arr[1][1][0].....arr[1][3][2]

Total number of elements in the above array are-

```
= 2 * 4 * 3
```

```
= 24
```

This array can be initialized as-

```
int arr[2][4][3] = {
    {
        {1, 2, 3}, /*Matrix 0, Row 0*/
        {4, 5}, /*Matrix 0, Row 1*/
        {6, 7, 8}, /*Matrix 0, Row 2*/
        {9} /*Matrix 0, Row 3*/
    },
    {
        {10, 11}, /*Matrix 1, Row 0*/
        {12, 13, 14}, /*Matrix 1, Row 1*/
        {15, 16}, /*Matrix 1, Row 2*/
        {17, 18, 19} /*Matrix 1, Row 3*/
    }
}
```

The value of elements after this initialization are as-

arr[0][0][0] : 1	arr[0][0][1] : 2	arr[0][0][2] : 3
arr[0][1][0] : 4	arr[0][1][1] : 5	arr[0][1][2] : 0
arr[0][2][0] : 6	arr[0][2][1] : 7	arr[0][2][2] : 8
arr[0][3][0] : 9	arr[0][3][1] : 0	arr[0][3][2] : 0
arr[1][0][0] : 10	arr[1][0][1] : 11	arr[1][0][2] : 0
arr[1][1][0] : 12	arr[1][1][1] : 13	arr[1][1][2] : 14
arr[1][2][0] : 15	arr[1][2][1] : 16	arr[1][2][2] : 0
arr[1][3][0] : 17	arr[1][3][1] : 18	arr[1][3][2] : 19

Remember that the rule of initialization of multidimensional arrays is that the last subscript varies most frequently and the first subscript varies least rapidly.

7.3.1 Multidimensional Array And Functions

Multidimensional arrays can also be passed to functions like 1-D arrays. When passing multidimensional arrays the first(leftmost) dimension may be omitted but all other dimensions have to be specified in the function definition. For example it would be invalid to write a function like this-

```
func(int a[], int b[], int c[][]) /*Invalid*/
{
    .....
}
```

In arrays b and c we can't omit all the dimensions. The correct form is-

```
func(int a[], int b[1][4], int c[1][3][15]) /*valid*/
{
    .....
}
```

7.4 Introduction To Strings

We'll discuss strings in detail in a separate chapter, here is just a brief introduction to strings. In C, strings are treated as arrays of type `char` and are terminated by a null character('0'). This null character has ASCII value zero.

These are the two forms of initialization of a string variable-

```
char str[10] = {'I', 'n', 'd', 'i', 'a', '\0' };
char str[10] = "India"; /*Here the null character is automatically placed at the end*/
```

7.4.1 Input and output of strings

```
/*P7.15 Program for input and output of strings using scanf() and printf()
 */
#include<stdio.h>
main()
{
    char str[10] = "Anpara";
    printf("String is : %s\n", str);
    printf("Enter new value for string : ");
    scanf("%s", str);
    printf("String is : %s\n", str);
}
```

Output:

```
String is : Anpara
Enter new value for string : Bareilly
String is : Bareilly
```

The next program uses the functions `gets()` and `puts()` for the input and output of strings.

```
/*P7.16 Program for input and output of strings using gets() and puts()*/
#include<stdio.h>
main()
{
    char str[10];
    printf("Enter string : ");
    gets(str);
    printf("String is : ");
    puts(str);
}
```

Output:

```
Enter string : New Delhi
String is : New Delhi
```

7.5 Some Additional Problems

Problem 1

Write a program using arrays, that reads a decimal number and converts it to (1) Binary (2) Octal or (3) Hexadecimal depending on user's choice.

```

/*P7.17 Program to convert a decimal number to Binary, octal or
hexadecimal*/
#include<stdio.h>
main()
{
    int num,opt;
    printf("Enter a decimal number : ");
    scanf("%d",&num);
    printf("1. Binary\n 2. Octal\n 3. Hexadecimal\n");
    printf("Enter your option : ");
    scanf("%d",&opt);
    switch(opt)
    {
        case 1:
            printf("Binary equivalent is : ");
            func(num,2);
            break;
        case 2:
            printf("Octal equivalent is : ");
            func(num,8);
            break;
        case 3:
            printf("Hexadecimal equivalent is : ");
            func(num,16);
            break;
    }
    printf("\n");
}
func(int num,int b)
{
    int i=0,j,rem;
    char arr[20];
    while(num>0)
    {
        rem=num%b;
        num/=b;
        if(rem>9&&rem<16)
            arr[i++]=rem-10+'A';
        else
            arr[i++]=rem+'0';
    }
    for(j=i-1;j>=0;j--)
        printf("%c",arr[j]);
}

```

Problem 2

Write a program for searching an element in an array through binary search.

The prerequisite for binary search is that the array should be sorted. Firstly we compare the item to be searched with the middle element of the array. If the item is same as the middle element, the search is successful otherwise the array is divided into two portions, first portion contains all elements

to the left of the middle element and the other one consists of all the elements on the right side of the element. Since the array is sorted, all the elements in the left portion will be smaller than the middle element and the elements in the right portion will be greater than the middle element. Now if the item to be searched is less than the middle element then we search it in the left portion of the array and if it is greater than the middle element then search will be in the right portion of the array.

Now we will take one portion only for search and compare the item with middle element of that portion. This process will continue until we find the required item or middle element has no left or right portion to search.

To implement this procedure we will take 3 variables viz. low, up and mid that will keep track of the status of lower limit, upper limit and middle value of that portion of the array, in which we will search the element. The value of the mid will be calculated as-

$$\text{mid} = (\text{low} + \text{up}) / 2$$

If item > arr[mid], search will resume in right portion

$$\text{low} = \text{mid} + 1, \text{ up will remain same}$$

If item < arr[mid] , search will resume in left portion

$$\text{up} = \text{mid}-1, \text{ low will remain same}$$

If item == arr[mid] , search is successful

item found at mid position

If low > up , search is unsuccessful

item not found in array

Let us take a sorted array of 10 elements. Suppose the element that we are searching is 49. The portion of array in which the element is searched is shown with a bold boundary in the figure.

	0	1	2	3	4	5	6	7	8	9	
low = 0, up = 9	10	15	18	20	25	30	49	57	64	72	49 > 25
mid = 4											low = mid+1=5
low = 5, up = 9	10	15	18	20	25	30	49	57	64	72	49 < 57
mid = 7											up = mid-1=6
low = 5, up = 6	10	15	18	20	25	30	49	57	64	72	49 > 30
mid = 5											low = mid+1=6
low = 6, up = 6	10	15	18	20	25	30	49	57	64	72	49 == 49
mid = 6											Found

Now let us take a case where the search fails. Suppose we are searching the element 16 that is not present in the array.

	0	1	2	3	4	5	6	7	8	9	
low = 0, up = 9	10	15	18	20	25	30	49	57	64	72	16<25
mid = 4											up=mid-1=3
low = 0, up = 3	10	15	18	20	25	30	49	57	64	72	16>15
mid = 1											low=mid+1=2
low = 2, up = 3	10	15	18	20	25	30	49	57	64	72	16<18
mid = 2											up=mid-1=1

Now low = 2 and up = 1, the value of low has exceeded the value of up so the search is unsuccessful.

```
/*P7.18 Program to search an element through binary search*/
#include <stdio.h>
#define SIZE 10
main()
{
    int arr[SIZE];
    int low, up, mid, i, item;
    printf("Enter elements of the array(in sorted order) : \n");
    for(i=0; i<SIZE; i++)
        scanf("%d", &arr[i]);
    printf("Enter the item to be searched : ");
    scanf("%d", &item);
    low=0;
    up=SIZE-1;
    while(low<=up&&item!=arr[mid])
    {
        mid=(low+up)/2;
        if(item>arr[mid])
            low=mid+1; /*Search in right portion */
        if(item <arr[mid])
            up=mid-1; /*Search in left portion */
        if(item==arr[mid])
            printf("%d found at position %d\n", item, mid+1);
        if(low>up)
            printf("%d not found in array\n", item);
    }
}
```

Problem 3

Write a program to sort the elements of a 1-D array, in ascending order through selection sort.

Sorting is a procedure in which the given elements are arranged in ascending or descending order. For example if the elements of an array are-

5, 11, 15, 8, 7, 54, 63, 44

After sorting these elements in ascending order the elements would be-

5, 7, 8, 11, 15, 44, 54, 63

16<25

up=mid-1=3

16>15.

low=mid+1=2

16<18

up=mid-1=1

rch is unsuccessful.

rch*/

: \n");

There are several methods for sorting an array, we'll discuss only insertion). These are discussed in Problems 3, 4 and 6 respectively. As the name suggests, selection sort technique selects an element a have a list of elements in unsorted order and you want to sort it, then and keep in the new list, after that second smallest element and so on.

Let us take an array arr[0], arr[1].....arr[N-1] of elements.

Pass 1:

Compare arr[0] with other elements from arr[1].....arr[N-1] one by one and exchange arr[0] with the element being compared.

Result : arr[0] is sorted.

Pass 2:

Compare arr[1] with other elements from arr[2].....arr[N-1] one by one and exchange arr[1] with the element being compared.

Result : arr[0], arr[1] are sorted.

Pass 3:

Compare arr[2] with other elements from arr[3].....arr[N-1], which is arr[2] with the element being compared.

Result : arr[0], arr[1], arr[2] are sorted.

.....
.....
.....

Pass N-1:

Compare arr[N-2] with arr[N-1], if arr[N-2] is bigger then exchange them.

Result : arr[0].....arr[N-1] are sorted.

Let us take a list of elements in unsorted order and sort them by a selection sort. Elements of the array are-

40	20	50	60	30	10	
----	----	----	----	----	----	--

Pass 1:

40	20	50	60	30	10	arr[0] > arr[1],
20	40	50	60	30	10	arr[0] < arr[2]
20	40	50	60	30	10	arr[0] < arr[3]
20	40	50	60	30	10	arr[0] < arr[4]
20	40	50	60	30	10	arr[0] > arr[5],
<u>10</u>	40	50	60	30	20	

Pass 2:

10	40	50	60	30	20	arr[1] < arr[2]
10	40	50	60	30	20	arr[1] < arr[3]

10	40	50	60	30	20	arr[1] > arr[4], Exchange
10	30	50	60	40	20	arr[1] > arr[5], Exchange
<u>10</u>	<u>20</u>	50	60	40	30	

Pass 3:

10	20	50	60	40	30	arr[2] < arr[3]
10	20	50	60	40	30	arr[2] > arr[4], Exchange
10	20	40	60	50	30	arr[2] > arr[5], Exchange
<u>10</u>	<u>20</u>	<u>30</u>	60	50	40	

Pass 4:

10	20	30	60	50	40	arr[3] > arr[4], Exchange
10	20	30	50	60	40	arr[3] > arr[5], Exchange
<u>10</u>	<u>20</u>	<u>30</u>	<u>40</u>	60	50	

Pass 5:

10	20	30	40	60	50	arr[4] > arr[5], Exchange
<u>10</u>	<u>20</u>	<u>30</u>	<u>40</u>	<u>50</u>	60	

Sorted array is :

10	20	30	40	50	60
----	----	----	----	----	----

The exchanges taking place in the passes are shown in the figure below-

Pass1					Pass 2					Pass 3					Pass 4		Pass 5		
i=0	j=1	j=2	j=3	j=4	j=5	i=1	j=2	j=3	j=4	j=5	i=2	j=3	j=4	j=5	i=3	j=4	j=5	i=4	j=5
0	40	20	20	20	20	10	10	10	10	10	10	10	10	10	10	10	10	10	
1	20	40	40	40	40	40	40	40	40	30	20	20	20	20	20	20	20	20	
2	50	50	50	50	50	50	50	50	50	50	50	50	40	40	30	30	30	30	
3	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	50	50	40	
4	30	30	30	30	30	30	30	30	30	40	40	40	40	50	50	60	60	60	
5	10	10	10	10	10	10	20	20	20	20	30	30	30	30	40	40	50	50	
	Ex		Ex				Ex	Ex			Ex	Ex		Ex	Ex		Ex		

```
/*P7.19 Program of sorting using selection sort*/
#include <stdio.h>
#define SIZE 10
main()
{
    int arr[SIZE];
    int i,j,temp;
    printf("Enter elements of the array : \n");
    for(i=0;i<SIZE;i++)
        scanf("%d",&arr[i]);
    for(i=0;i<SIZE-1;i++)
        for(j=i+1;j<SIZE;j++)

```

```

    {
        if(arr[i]>arr[j])
        {
            temp=arr[i];
            arr[i]=arr[j];
            arr[j]=temp ;
        }
    }
    printf("Sorted array is : \n");
    for(i=0;i<SIZE;i++)
        printf("%d ",arr[i]);
    printf("\n");
}

```

Problem 4

Write a program to sort the elements of a 1-D array, in ascending order through bubble sort
 If N elements are given, then the procedure for sorting them through bubble sort is as-
 The elements of array are arr[0],arr[1].....arr[N-1]

Pass 1:

Compare 0th and 1st element, If 0th > 1st then exchange them

Compare 1st and 2nd element, If 1st > 2nd then exchange them,

Compare 2nd and 3rd element, If 2nd > 3rd then exchange them,

.....
 Compare N-2th with N-1th, If N-2th > N-1th then exchange them

Pass 2:

Compare 0th and 1st element, If 0th > 1st then exchange them

Compare 1st and 2nd element, If 1st > 2nd then exchange them,

Compare 2nd and 3rd element, If 2nd > 3rd then exchange them,

.....
 Compare N-3th and N-2th element, If N-3th > N-2th then exchange them

Pass N-1:

Compare 0th and 1st element, If 0th > 1st then exchange them

Let us take a list of elements in unsorted order and sort them by applying bubble sort.

Elements of the array are-

40 20 50 60 30 10

Pass 1:

40	20	50	60	30	10	arr[0] > arr[1], Exchange
20	40	50	60	30	10	arr[1] < arr[2]

20	40	50	60	30	10	arr[2] < arr[3]
20	40	50	60	30	10	arr[3] > arr[4], Exchange
20	40	50	30	60	10	arr[4] > arr[5], Exchange
20	40	50	30	10	<u>60</u>	

Pass 2:

20	40	50	30	10	60	arr[0] < arr[1]
20	40	50	30	10	60	arr[1] < arr[2]
20	40	50	30	10	60	arr[2] > arr[3], Exchange
20	40	30	50	10	60	arr[3] > arr[4], Exchange
20	40	30	10	<u>50</u>	<u>60</u>	

Pass 3:

20	40	30	10	50	60	arr[0] < arr[1]
20	40	30	10	50	60	arr[1] > arr[2], Exchange
20	30	40	10	50	60	arr[2] > arr[3], Exchange
20	30	10	<u>40</u>	<u>50</u>	<u>60</u>	

Pass 4:

20	30	10	40	50	60	arr[0] < arr[1]
20	30	10	40	50	60	arr[1] > arr[2], Exchange
20	10	<u>30</u>	<u>40</u>	<u>50</u>	<u>60</u>	

Pass 5:

20	10	30	40	50	60	arr[0] > arr[1], Exchange
10	<u>20</u>	<u>30</u>	<u>40</u>	<u>50</u>	<u>60</u>	

Sorted array is-

10 20 30 40 50 60

Pass1					Pass 2				Pass 3			Pass 4		Pass 5					
i=0	j=0	j=1	j=2	j=3	j=4	i=1	j=0	j=1	j=2	j=3	i=2	j=0	j=1	j=2	i=3	j=0	j=1	i=4	j=0
0	40	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	
1	20	40	40	40	40	40	40	40	40	40	40	40	30	30	30	10	10	10	
2	50	50	50	50	50	50	50	50	50	30	30	30	30	40	10	30	30	30	
3	60	60	60	60	30	30	30	30	30	50	10	10	10	10	40	40	40	40	
4	30	30	30	30	60	10	10	10	10	10	50	50	50	50	50	50	50	50	
5	10	10	10	10	10	60	60	60	60	60	60	60	60	60	60	60	60	60	
Ex					Ex Ex				Ex			Ex		Ex					

```
/*P7.20 Program of sorting using bubble sort*/
#include <stdio.h>
#define SIZE 10
main()
```

```

{
    int arr[SIZE];
    int i,j,temp;
    printf("Enter elements of the array :\n");
    for(i=0;i<SIZE;i++)
        scanf("%d",&arr[i]);
    for(i=0;i<SIZE-1;i++)
    {
        for(j=0;j<SIZE-1-i;j++)
        {
            if(arr[j]>arr[j+1])
            {
                temp=arr[j];
                arr[j]=arr[j+1];
                arr[j+1]=temp;
            }
        }
    }
    printf("Sorted array is :\n");
    for(i=0;i<SIZE;i++)
        printf("%d ",arr[i]);
    printf("\n");
}

```

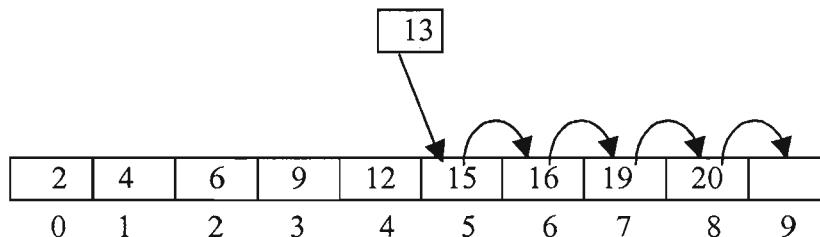
Problem 5

Write a program to insert an element in a sorted 1-D array at proper place, so that the array remains sorted after insertion also.

Suppose we have an array of size 10 and there are nine elements in it which are in ascending order, one rightmost space is empty for the new element to be inserted.

2	4	6	9	12	15	16	19	20	
0	1	2	3	4	5	6	7	8	9

To insert an item, we'll compare it with the elements of array from the right side and keep on shifting them to the right. As soon as we get an element less than the item we'll stop this process and insert the item. Suppose we have to insert the number 13 in the array-



After insertion the array becomes-

2	4	6	9	12	13	15	16	19	20
0	1	2	3	4	5	6	7	8	9

```

/*P7.21 Program to insert an item in a sorted array at the proper place
by shifting other elements to the right*/
#include<stdio.h>
#define SIZE 10
main()
{
    int arr[SIZE];
    int i, item;
    printf("Enter elements of the array(in sorted order) : \n");
    for(i=0;i<SIZE-1;i++) /*rightmost space in array should be empty*/
        scanf("%d",&arr[i]);
    printf("Enter the item to be inserted : ");
    scanf("%d",&item);
    for(i=SIZE-2;item<arr[i]&&i>=0;i--)
        arr[i+1]=arr[i]; /*Shift elements to the right*/
    arr[i+1]=item; /*Insert item at the proper place*/

    for(i=0;i<SIZE;i++)
        printf("%d ",arr[i]);
    printf("\n");
}

```

The condition $i \geq 0$ in the loop is there because if the item is to be inserted at 0th place, then value of i will become -1, but $a[-1]$ is not a valid element.

Problem 6

Write a program to sort the elements of a 1-D array, in ascending order through insertion sort.

To sort an array through insertion sorting we'll use the insertion procedure described in the previous program. Here we will place each element of array at proper place in the previously sorted element list.

Let us take there are N elements in the array arr. Then process of inserting each element in proper place is as-

Pass 1:

arr[1] is inserted before or after arr[0].

So arr[0] and arr[1] are sorted.

Pass 2:

arr[2] is inserted before arr[0], in between arr[0] and arr[1] or after arr[1].

So arr[0], arr[1] and arr[2] are sorted.

Pass 3:

arr[3] is inserted into its proper place in array arr[0], arr[1], arr[2]

So arr[0], arr[1], arr[2], arr[3] are sorted.

.....

.....

.....

Pass N-1:

arr[N-1] is inserted into its proper place in array

arr[0], arr[1],arr[N-2]

So arr[0], arr[1].....arr[N-1] are sorted.

Let us take a list of elements in unsorted order and sort them by applying insertion sort.

90	30	10	40	50	20	100	70	80	60	
k=1,	Insert item = arr[1] = 30	in	90							
k=2,	Insert item = arr[2] = 10	in	30	90						
k=3,	Insert item = arr[3] = 40	in	10	30	90					
k=4,	Insert item = arr[4] = 50	in	10	30	40	90				
k=5,	Insert item = arr[5] = 20	in	10	30	40	50	90			
k=6,	Insert item = arr[6] = 100	in	10	20	30	40	50	90		
k=7,	Insert item = arr[7] = 70	in	10	20	30	40	50	90	100	
k=8,	Insert item = arr[8] = 80	in	10	20	30	40	50	70	90	100
k=9,	Insert item = arr[9] = 60	in	10	20	30	40	50	70	80	90
										100

```
/*P7.22 Program to sort numbers using insertion sort*/
#include <stdio.h>
#define SIZE 10
main()
{
    int arr[SIZE];
    int i,k,item;
    printf("Enter elements of the array :\n");
    for(i=0;i<SIZE;i++)
        scanf("%d",&arr[i]);
    for(k=1;k<SIZE;k++)
    {
        item=arr[k]; /*item is to be inserted at proper place*/
        for(i=k-1;item<arr[i]&&i>=0;i--)
            arr[i+1]=arr[i];
        arr[i+1]=item;
    }
    printf("Sorted array is :\n");
    for(i=0;i<SIZE;i++)
        printf("%d ",arr[i]);
    printf("\n");
}
```

Problem 7

Write a program for merging two sorted arrays into a third sorted array

If there are two sorted arrays, then process of combining these sorted arrays into another in sorted order is called merging. Let us take two arrays arr1 and arr2 in sorted order, we'll combine them into a third sorted array arr3.

arr1- 5 8 9 28 34

arr2- 4 22 25 30 33 40 42

We'll take one element from each array, compare them and then take the smaller one in third array. This process will continue until the elements of one array are finished. Then take the remaining elements of unfinished array in third array. The whole process for merging is shown below. arr3 is the merged array, i, j, k are variables used for subscripts of arr1, arr2, arr3 respectively.

Initially i = 0, j = 0, k = 0

i=0

5	8	9	28	34
---	---	---	----	----

j=0

4	22	25	30	33	40	42
---	----	----	----	----	----	----

i=0

5	8	9	28	34
---	---	---	----	----

j=1

4	22	25	30	33	40	42
---	----	----	----	----	----	----

i=1

5	8	9	28	34
---	---	---	----	----

j=1

4	22	25	30	33	40	42
---	----	----	----	----	----	----

i=2

5	8	9	28	34
---	---	---	----	----

j=1

4	22	25	30	33	40	42
---	----	----	----	----	----	----

i=3

5	8	9	28	34
---	---	---	----	----

j=1

4	22	25	30	33	40	42
---	----	----	----	----	----	----

i=3

5	8	9	28	34
---	---	---	----	----

j=2

4	22	25	30	33	40	42
---	----	----	----	----	----	----

i=3

5	8	9	28	34
---	---	---	----	----

j=3

4	22	25	30	33	40	42
---	----	----	----	----	----	----

i=4

5	8	9	28	34
---	---	---	----	----

j=3

4	22	25	30	33	40	42
---	----	----	----	----	----	----

i=4

5	8	9	28	34
---	---	---	----	----

j=4

4	22	25	30	33	40	42
---	----	----	----	----	----	----

i=4

5	8	9	28	34
---	---	---	----	----

j=5

4	22	25	30	33	40	42
---	----	----	----	----	----	----

k=0

4								
---	--	--	--	--	--	--	--	--

k=1

4	5							
---	---	--	--	--	--	--	--	--

k=2

4	5	8						
---	---	---	--	--	--	--	--	--

k=3

4	5	8	9					
---	---	---	---	--	--	--	--	--

k=4

4	5	8	9	22	25			
---	---	---	---	----	----	--	--	--

k=5

4	5	8	9	22	25	25		
---	---	---	---	----	----	----	--	--

k=6

4	5	8	9	22	25	28		
---	---	---	---	----	----	----	--	--

k=7

4	5	8	9	22	25	28	30	
---	---	---	---	----	----	----	----	--

k=8

4	5	8	9	22	25	28	30	33
---	---	---	---	----	----	----	----	----

k=9

4	5	8	9	22	25	28	30	33	34
---	---	---	---	----	----	----	----	----	----

Now take 40, 42 in arr3 because arr1 has no more elements to compare.

So finally the array arr3 is-

arr3-

4	5	8	9	22	25	28	30	33	34	40	42
---	---	---	---	----	----	----	----	----	----	----	----

```
/*P7.23 Program for merging two sorted arrays into a third sorted array*/
#include<stdio.h>
#define SIZE1 5
#define SIZE2 7
#define SIZE3 SIZE1+SIZE2
main()
{
    int arr1[SIZE1], arr2[SIZE2], arr3[SIZE3];
    int i, j, k;
```

```

printf("Enter elements of the array arr1 (in sorted order) : \n");
for(i=0;i<SIZE1;i++)
    scanf("%d",&arr1[i]);

printf("Enter elements of the array arr2 (in sorted order) :\n");
for(i=0;i<SIZE2;i++)
    scanf("%d",&arr2[i]);

i=0,j=0,k=0;
while((i<SIZE1)&&(j<SIZE2))
{
    if(arr1[i]<arr2[j])
        arr3[k++]=arr1[i++];
    else
        arr3[k++]=arr2[j++];
}
while(i<SIZE1) /*Put remaining elements of arr1 into arr3*/
    arr3[k++]=arr1[i++];
while(j<SIZE2) /*Put remaining elements of arr2 into arr3*/
    arr3[k++]=arr2[j++];

printf("Merged array arr3 is : \n");
for(i=0;i<SIZE3;i++)
    printf("%d ",arr3[i]);
printf("\n");
}/*End of main()*/

```

Problem 8

Write a program to print the Pascal triangle using a 2-D array. In program P6.30, we had printed this triangle using functions.

```

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1

```

We'll calculate and store all the elements of Pascal triangle in a 2-D array. Form the figure we can observe that-

- (i) All the elements of Column 0 are 1.
- (ii) All the elements for which Row and Column are same, are 1.
- (iii) Any other element can be obtained by adding two elements of previous row as-

$$a[i][j] = a[i-1][j-1] + a[i-1][j];$$

where i and j represent row and column number.

```
/*P7.24 Program to print the Pascal triangle*/
#include<stdio.h>
```

```
#define MAX 15
main()
{
    int a[MAX][MAX];
    int i,j,n;
    printf("Enter n :");
    scanf("%d",&n);
    for(i=0;i<=n;i++)
    {
        for(j=0;j<=i;j++)
            if(j==0||i==j)
                a[i][j]=1;
            else
                a[i][j]=a[i-1][j-1]+a[i-1][j];
    }
    for(i=0;i<n;i++)
    {
        for(j=0;j<=i;j++)
            printf("%5d", a[i][j]);
        printf("\n");
    }
}
```

Problem 9

Write a program to print the magic matrix.

- Magic matrix is a square matrix of order $n \times n$, i.e. number of rows is equal to number of columns.
- A magic matrix exists only for odd values of n .
- The numbers in matrix will be $1, 2, 3, 4, \dots, n^2$ and each number can occur in the matrix only once.
- The sums of elements of every row, column, and diagonal are equal. This sum is always equal to $n(n^2+1)/2$.

The magic matrices for $n = 3$, $n = 5$, $n = 7$ are shown below-

2	9	4
7	5	3
6	1	8

$n=3$

Sum = 15

9	2	25	18	11
3	21	19	12	10
22	20	13	6	4
16	14	7	5	23
15	8	1	24	17

$n=5$

Sum = 65

20	11	2	49	40	31	22
12	3	43	41	32	23	21
4	44	42	33	24	15	13
45	36	34	25	16	14	5
37	35	26	17	8	6	46
29	27	18	9	7	47	38
28	19	10	1	48	39	30

$n=7$

Sum=175

The procedure for creating a magic matrix is as-

Start filling numbers from the centre column of bottom row, so initially place the number 1 in the centre column of bottom row. Keep on placing the numbers by moving one row down and one column left (down-left) till you reach one of the following situations-

- (i) If you reach at the bottom left of the matrix, or a square that is already filled then move one row up in the same column. This situation will arise when previously placed number was divisible by n.
- (ii) If you have to move left of the leftmost column, then go to the rightmost column.
- (iii) If you have to move down the bottom row, then go to the topmost row.

```
/*P7.25 Program to print the magic matrix*/
#include<stdio.h>
#define MAX 20
main()
{
    int a[MAX][MAX], i, j, n, num;
    printf("Enter value of n(odd value) : ");
    scanf("%d", &n);
    i=n-1; /*Bottom row*/
    j=(n-1)/2; /*Centre column*/

    for(num=1; num<=n*n; num++)
    {
        a[i][j]=num;
        i++; /*move down*/
        j--; /*move left*/
        if(num%n==0)
        {
            i-=2; /*one above the previous row*/
            j++; /*back to the previous column*/
        }
        if(i==n)
            i=0; /*go to topmost row*/
        if(j==-1)
            j=n-1; /*go to rightmost column*/
    } /*End of for*/
    for(i=0; i<n; i++)
    {
        for(j=0; j<n; j++)
            printf("%4d", a[i][j]);
        printf("\n");
    }
}
```

Problem 10

Write a program to print Spiral Matrix.

A spiral matrix is $n \times n$ square matrix formed by placing the numbers 1, 2, 3, 4 n^2 in spiral form starting from the leftmost column and topmost row. Spiral matrices can exist for both even and odd values of n. The spiral matrices for $n = 3$, $n = 4$, $n = 7$ are shown below-

1	2	3
8	9	4
7	6	5

1	2	3	4
12	13	14	5
11	16	15	6
10	9	8	7

n = 3

n = 4

n = 7

```
/*P7.26 Program to print the spiral matrix*/
#include<stdio.h>
#define MAX 20
main()
{
    int n,i=0,j=0,l,u,num=1,arr[MAX][MAX]={0};
    printf("Enter value of n : ");
    scanf("%d",&n);
    l=0;u=n-1;
    for(num=1;num<=n*n;num++)
    {
        arr[i][j]=num;
        if(i==l&&j<u)
            j++;
        else if(j==u&&i<u)
            i++;
        else if(i==u&&j>l)
            j--;
        else if(j==l&&i>l)
            i--;
        if(arr[i][j]!=0)      /*If square already filled*/
        {
            l++;
            u--;
            i++;
            j++;
        }
    }/*End of for*/
    for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
            printf("%4d",arr[i][j]);
        printf("\n");
    }
}
```

1	2	3	4	5	6	7
24	25	26	27	28	29	8
23	40	41	42	43	30	9
22	39	48	49	44	31	10
21	38	47	46	45	32	11
20	37	36	35	34	33	12
19	18	17	16	15	14	13

Exercise

Assume that stdio.h is included in all the programs.

```
(1) main()
{
    int i, size=5, arr[size];
    for(i=0; i<size; i++)
        scanf("%d", &arr[i]);
    for(i=0; i<size; i++)
        printf("%d ", arr[i]);
}

(2) main()
{
    int arr[4]={2, 4, 8, 16}, i=4, j;
    while(i)
    {
        j=arr[i]+i;
        i--;
    }
    printf("j=%d\n", j);
}

(3) main()
{
    int i=0, sum=0, arr[6]={4, 2, 6, 0, 5, 10};
    while(arr[i])
    {
        sum=sum+arr[i];
        i++;
    }
    printf("sum = %d\n", sum);
}

(4) main()
{
    int i, arr[8]={1, 2, 3, 4, 5, 6, 7, 8};
    for(i=7; i>=0; i--)
        printf("%d\t", -arr[-i]);
}

(5) main()
{
    int arr[5]={5, 10, 15, 20, 25};
    func(arr);
}
func(int arr[])
{
    int i=5, sum=0;
    while(i>2)
        sum=sum+arr[-i];
```

```

        printf("sum = %d\n", sum);
    }

(6) main()
{
    int x[10],y[3][4],z[2][3][5];
    printf("%u \t %u \t %u\n", sizeof(x), sizeof(y), sizeof(z));
}

(7) main()
{
    int a=4,b=6;
    int arr1[5]={1,2,3,4,5};
    int arr2[5]={6,7,8,9,10};
    swapvar(a,b);
    swaparr(arr1,arr2);
    printf("a = %d, b = %d\n",a,b);
    printf("arr1[0] = %d, arr1[4] = %d\n",arr1[0],arr1[4]);
    printf("arr2[0] = %d, arr2[4] = %d\n",arr2[0],arr2[4]);
}
swapvar(int a,int b)
{
    int temp;
    temp=a,a=b,b=temp;
}
swaparr(int arr1[5],int arr2[5])
{
    int i,temp;
    for(i=0;i<5;i++)
    { temp=arr1[i], arr1[i]=arr2[i], arr2[i]=temp; }
}

(8) main()
{
    int i,j,arr[3][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12}};
    for(i=0;i<4;i++)
    {
        for(j=0;j<3;j++)
            printf("%3d",arr[j][i]);
        printf("\n");
    }
}

(9) #include<math.h>
main()
{
    int i,j,arr[200];
    for(i=2;i<200;i++)
        arr[i]=1;
    for(i=2;i<=sqrt(200);i++)
        for(j=i*2;j<200;j+=i)
}

```

```

        arr[j]=0;
    for(i=2;i<200;i++)
        if(arr[i]==1)
            printf("%d\t",i);
    }

(10)main()
{
    int a[10]={2,-3,4,-5,6,7,1,9,-10,-11};
    int i,j,x,k=0;
    for(i=0;i<10;i++)
    {
        x=a[k];
        if(x<0)
        {
            for(j=k;j<10;j++)
                a[j]=a[j+1];
            a[9]=x;
        }
        else
            k++;
    }
    for(i=0;i<10;i++)
        printf("%d ",a[i]);
    printf("\n");
}

```

Programming Exercise

1. Write a program to accept n numbers and display the sum of the highest and lowest numbers.
2. Write a program to accept n numbers in array and display the addition of all even numbers and multiplication of all odd numbers.
3. Write a program to sort numbers of a one-d array in descending order using
(i) selection sort (ii) bubble sort (iii) insertion sort
4. Write a function to reverse only first n elements of an array.
5. Write a program to modify the elements of an array such that the last element becomes the first element of the array and all other elements are shifted to right.

1 2 3 4 5 6 7 8 9 → 9 1 2 3 4 5 6 7 8

We can say that we have rotated the array to the right by one element. Now modify the above program so that we can rotate the array by any number of elements. For example when we rotate the array by 3 elements the result would be-

1 2 3 4 5 6 7 8 9 → 7 8 9 1 2 3 4 5 6

6. Write a program to find out the determinant of a matrix.
7. Write a program to count the occurrences of a number in a matrix.
8. Write a program to store all the elements of a 2-D array in a 1-D array row-wise.
9. Write a program to find out whether a matrix is symmetric or not. A matrix is symmetric if transpose of the matrix is equal to the matrix.

10. Write a program to check that the elements of an array are distinct.
11. Write a program to check that the elements of a matrix are distinct.
12. Write a program to find out the sum of elements of principal and secondary diagonals of a square matrix.
13. Write a program to enter a square matrix of odd size and then check whether it is a magic matrix or not. Any matrix is a magic matrix if all the elements in it are distinct and the sum of elements in each row, column and diagonal are equal.
14. Write a program to print the elements of a matrix spirally. For example if the matrix is-

2 5 8
1 3 7
6 2 9

The output should be-

2 5 8 7 9 2 6 1 3

15. Write a program to reverse the rows of a matrix.

1	2	3	4	→	13	14	15	16
5	6	7	8		9	10	11	12
9	10	11	12		5	6	7	8
13	14	15	16		1	2	3	4

16. Write a program to reverse the columns of a matrix

1	2	3	4	→	4	3	2	1
5	6	7	8		8	7	6	5
9	10	11	12		12	11	10	9
13	14	15	16		16	15	14	13

17. Write a program to sort the elements of a matrix (i) row-wise (ii) column-wise

5	8	2	1	→	1	2	5	8	→	5	8	2	1	→	1	3	2	1
3	6	9	4		3	4	6	9		3	6	9	4		3	6	2	4
1	7	2	8		1	2	7	8		1	7	2	8		5	7	5	7
8	3	5	7		3	5	7	8		8	3	5	7		8	8	9	8
Matrix				(i)	Matrix				(ii)									

Answers

- (1) Error, the size of an array should be a constant expression.
- (2) $j = 5$
- (3) $\text{sum} = 12$
- (4) 6 4 2 0
- (5) $\text{sum} = 60$
- (6) 20 24 60
- (7) $a = 4, b = 6$
 $\text{arr1}[0] = 6, \text{arr1}[4] = 10$

arr2[0] = 1, arr2[4] = 5

Any changes made to the array inside the function are visible in the calling function.

- (8) 1 5 9
2 6 10
3 7 11
4 8 12

Here the rows are printed vertically and the columns are printed horizontally.

- (9) Prints all the prime numbers less than 200. This method of finding out prime numbers is known as Sieve of Eratosthenes.
- (10) 2 4 6 7 1 9 -3 -5 -10 -11

This program suppresses the negative numbers at the end of the array.

Chapter 8

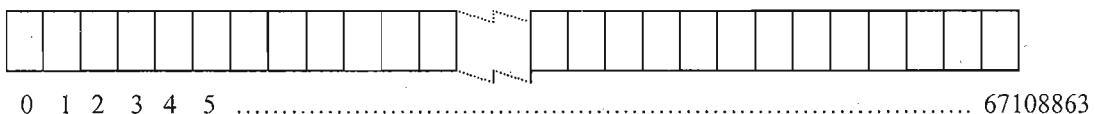
Pointers

C is a very powerful language and the real power of C lies in pointers. The concept of pointers is interesting as well as challenging. It is very simple to use pointers provided the basics are understood thoroughly. So it is necessary to visualize every aspect of pointers instead of just having a superficial knowledge about their syntax and usage. The use of pointers makes the code more efficient and compact. Some of the **uses of pointers** are-

- (i) Accessing array elements.
- (ii) Returning more than one value from a function.
- (iii) Accessing dynamically allocated memory.
- (iv) Implementing data structures like linked lists, trees, and graphs.

8.1 About Memory

Before studying pointers it is important to understand how memory is organized in a computer. The memory in a computer is made up of bytes arranged in a sequential manner. Each byte has an index number, which is called the address of that byte. The address of these bytes start from zero and the address of last byte is one less than the size of memory. Suppose we have 64 MB of RAM, then memory will consist of $64 * 2^{20} = 67108864$ bytes. The address of these bytes will be from 0 to 67108863.



We have studied that it is necessary to declare a variable before using it, since compiler has to reserve space for it. The data type of the variable also has to be mentioned so that the compiler knows how much space needs to be reserved. Now let us see what happens when we declare a variable. Suppose we declare a variable age of type int-

```
int age;
```

The compiler reserves 2 consecutive bytes from memory for this variable and associates the name age with it. The address of first byte from the two allocated bytes is known as the **address of variable age**.

Suppose compiler has reserved bytes numbered 2588 and 2599 for the storage of variable age, then the address of variable age will be 2583. Let us assign some value to this variable-

age = 20;

Now this value will be stored in these 2 bytes (of course in binary form). The number of bytes allocated will depend on the data type of variable. For example 4 bytes would have been allocated for a float variable, and the address of first byte would be called the address of the variable. Now we will see how to find out the address of a variable.

8.2 Address Operator

C provides an address operator '&', which returns the address of a variable when placed before it. This operator can be read as "the address of", so &age means address of age, similarly &sal means address of sal. The following program prints the address of variables using address operator.

```
/*P8.1 Program to print address of variables using address operator*/
#include<stdio.h>
main()
{
    int age=30;
    float sal=1500.50;
    printf("Value of age = %d, Address of age = %u\n",age,&age);
    printf("Value of sal = %f, Address of sal = %u\n",sal,&sal);
}
```

Output:

Value of age = 30, Address of age = 65524
 Value of sal = 1500.500000, Address of sal = 65520

Here we have used %u control sequence to print the address, but this does not mean that addresses are unsigned integers. We have used it since there is no specific control sequence to display addresses. Addresses are just whole numbers. These addresses may be different each time you run your program, it depends on which part of memory is allocated by operating system for this program.

The address operator cannot be used with a constant or an expression.

&j;	/*Valid, used with a variable*/
&arr[1];	/*Valid, used with an array element */
&289;	/*Invalid, used with a constant*/
&(j+k);	/*Invalid, used with an expression*/

This address operator is not new for us, we have already used it in scanf() function. The address of variable is provided to scanf(), so that it knows where to write the input value. So now you can understand why '&' was placed before the variable names in scanf().

8.3 Pointers Variables :

Finally after this brief introduction, it is time to introduce pointers. A pointer is a variable that stores memory address. Like all other variables it also has a name, has to be declared and occupies some space in memory. It is called pointer because it points to a particular location in memory by storing the address of that location.

8.3.1 Declaration Of Pointer Variables

Like other variables, pointer variables should also be declared before being used. The general syntax of declaration is-

```
data_type *pname;
```

Here pname is the name of pointer variable, which should be a valid C identifier. The asterisk '*' preceding this name informs the compiler that the variable is declared as a pointer. Here **data_type** is known as the **base type of pointer**. Let us take some pointer declarations-

```
int *iptr;
float *fptr;
char *cptr, ch1, ch2;
```

Here iptr is a pointer that should point to variables of type int, similarly fptr and cptr should point to variables of float and char type respectively. Here type of variable iptr is 'pointer to int' or (int *), or we can say that base type of iptr is int. We can also combine the declaration of simple variables and pointer variables as we have done in the third declaration statement where ch1 and ch2 are declared as variables of type char.

Pointers are also variables so compiler will reserve space for them and they will also have some address. All pointers irrespective of their base type will occupy same space in memory since all of them contain addresses only. Generally 2 bytes are used to store an address (may vary in different computers), so the **compiler allocates 2 bytes for a pointer variable**.

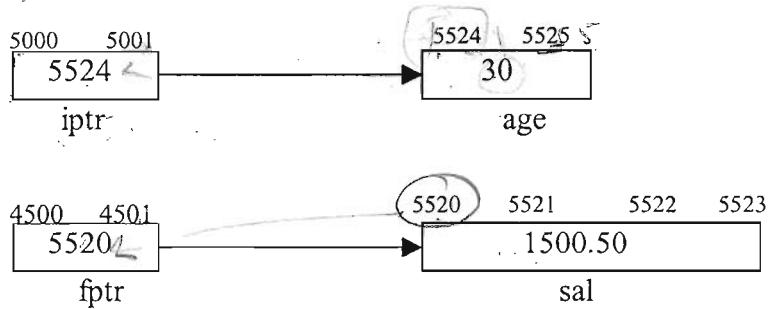
Now the question arises that when all the pointers contain addresses only and each one occupies 2 bytes, then why we have to mention the data type in the declaration statement. We will come to know about this when we study about indirection operator and pointer arithmetic.

8.3.2 Assigning Address To Pointer Variables

When we declare a pointer variable it contains **garbage value** i.e. it may be pointing anywhere in the memory. So we should always assign an address before using it in the program. The **use of an unassigned pointer** may give **unpredictable results** and even **cause the program to crash**. Pointers may be assigned the address of a variable using assignment statement. For example-

```
int *iptr, age = 30;
float *fptr, sal = 1500.50;
iptr = &age;
fptr = &sal;
```

Now iptr contains the address of variable age i.e. it points to variable age, similarly fptr points to variable sal. Since iptr is declared as a pointer of type int, we should assign address of only integer variables to it. If we assign address of some other data type then compiler won't show any error but the output will be incorrect.



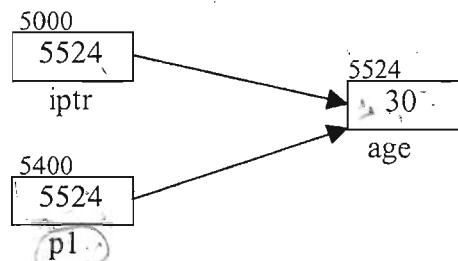
We can also initialize the pointer at the time of declaration. But in this case the variable should be declared before the pointer. For example-

```
int age = 30, *iptr = &age;
float sal = 1500.50, *fptr = &sal;
```

It is also possible to assign the value of one pointer variable to the other, provided their base type is same. For example if we have an integer pointer `p1` then we can assign the value of `iptr` to it as-

```
p1 = iptr;
```

Now both pointer variables `iptr` and `p1` contain the address of variable `age` and point to the same variable `age`.



We can assign constant zero to a pointer of any type. A symbolic constant `NULL` is defined in `stdio.h`, which denotes the value zero. The assignment of `NULL` to a pointer guarantees that it does not point to any valid memory location. This can be done as-

```
ptr = NULL;
```

8.3.3 Dereferencing Pointer Variables

Till now we have done a lot of programming and in all our programs we have used the name of a variable for accessing it. We can also access a variable indirectly using pointers. For this we will use the indirection operator (`*`). By placing the indirection operator before a pointer variable, we can access the variable whose address is stored in the pointer. Let us take an example-

```
int a = 87;
float b = 4.5;
int *p1 = &a;
float *p2 = &b;
```

In our program, if we place `*` before `p1` then we can access the variable whose address is stored in `p1`. Since `p1` contains the address of variable `a`, we can access the variable `a` by writing `*p1`. Similarly we can access variable `b` by writing `*p2`. So we can use `*p1` and `*p2` in place of variable names `a`

and **b** anywhere in our program. Let us see some examples-

<code>*p1 = 9;</code>	is equivalent to	<code>a = 9;</code>
<code>(*p1)++;</code>	is equivalent to	<code>a++;</code>
<code>x = *p2 + 10 ;</code>	is equivalent to	<code>x = b + 10;</code>
<code>printf("%d %f", *p1, *p2);</code>	is equivalent to	<code>printf("%d %f", a, b);</code>
<code>scanf("%d%f", p1, p2);</code>	is equivalent to	<code>scanf("%d%f ", &a, &b);</code>

The indirection operator can be read as 'value at the address'. For example `*p1` can be read as 'value at the address `p1`'. This indirection operator (*) is different from the asterisk that was used while declaring the pointer variable.

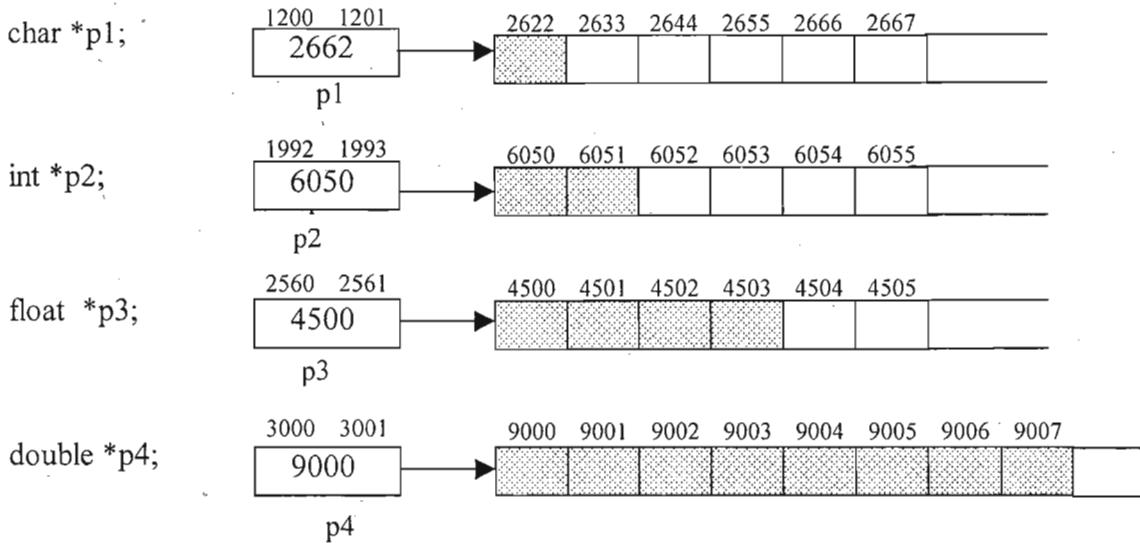
Now we will see what is meant by the term `*(&age)`, where `age` is a variable. Since `&age` is an address, so dereferencing it with `*` operator will give the variable at that address and the variable at that address is `age`. Hence `*(&age)` is same as writing `age`.

```
/*P8.2 Program to dereference pointer variables*/
#include<stdio.h>
main()
{
    int a=87;
    float b=4.5;
    int *p1=&a;
    float *p2=&b;
    printf("Value of p1 = Address of a = %u\n",p1);
    printf("Value of p2 = Address of b = %u\n",p2);
    printf("Address of p1 = %u\n",&p1);
    printf("Address of p2 = %u\n",&p2);
    printf("Value of a = %d %d %d \n",a,*p1,*(&a));
    printf("Value of b = %f %f %f \n",b,*p2,*(&b));
}
```

Output :

```
Value of p1 = Address of a = 65524
Value of p2 = Address of b = 65520
Address of p1 = 65518
Address of p2 = 65516
Value of a = 87 87 87
Value of b = 4.500000 4.500000 4.500000
```

We have already seen that while declaring a pointer variable, we have to mention the data type. The reason is that when we use the indirection operator, the number of bytes retrieved will be different for different data types. The value of the pointer only tells the address of starting byte. For example suppose we have a pointer `ptr` which contains the address 2000 and when we write `*ptr` the compiler knows that it has to access the information starting at address 2000. So the compiler will look at the base type of the pointer and will retrieve the information depending on that base type. For example if base type is `int` then 2 bytes information will be retrieved and if base type is `float`, 4 bytes information will be retrieved and so on. The following figures illustrate this fact. The shaded portion shows the number of bytes retrieved.



The size of pointer variable is same for all type of pointers but the memory that will be accessed while dereferencing is different.

```
/*P8.3 Program to print the size of pointer variable and size of value
dereferenced by that pointer*/
#include<stdio.h>
main( )
{
    char a='x',*p1=&a;
    int b=12,*p2=&b;
    float c=12.4,*p3=&c;
    double d=18.3,*p4=&d;
    printf("sizeof(p1) = %d , sizeof(*p1)= %d\n",sizeof(p1),sizeof(*p1));
    printf("sizeof(p2) = %d , sizeof(*p2) = %d\n",sizeof(p2),sizeof(*p2));
    printf("sizeof(p3) = %d , sizeof(*p3) = %d\n",sizeof(p3),sizeof(*p3));
    printf("sizeof(p4) = %d , sizeof(*p4) = %d\n",sizeof(p4),sizeof(*p4));
}
```

Output:

sizeof(p1) = 2 , sizeof(*p1) = 1
 sizeof(p2) = 2 , sizeof(*p2) = 2
 sizeof(p3) = 2 , sizeof(*p3) = 4
 sizeof(p4) = 2 , sizeof(*p4) = 8

8.4 Pointer Arithmetic

All types of arithmetic operations are not possible with pointers. The only valid operations that can be performed are as-

- (1) Addition of an integer to a pointer and increment operation.
- (2) Subtraction of an integer from a pointer and decrement operation
- (3) Subtraction of a pointer from another pointer of same type.

Pointer arithmetic is somewhat different from ordinary arithmetic. Here all arithmetic is performed relative

to the size of base type of pointer. For example if we have an integer pointer pi which contains address 1000 then on incrementing we get 1002 instead of 1001. This is because the size of int data type is 2. Similarly on decrementing pi, we will get 998 instead of 999. The expression ($\text{pi}+3$) will represent the address 1006. Let us see some pointer arithmetic for int, float and char pointers.

```
int a = 5, *pi = &a;
float b = 2.2, *pf = &b;
char c = 'x', *pc = &c;
```

Suppose the address of variables a, b and c are 1000, 4000, 5000 respectively, so initially values of p1, p2, p3 will be 1000, 4000 and 5000.

$\text{pi}++$; or $++\text{pi}$; $\text{pi} = 1000 + 2 = 1002$ (Since int is of 2 bytes)

$\text{pi} = \text{pi}-3$; $\text{pi} = 1002 - 3*2 = 996$

$\text{pi} = \text{pi}+5$; $\text{pi} = 996 + 5*2 = 1006$

$\text{pi}-;$ or $--\text{pi}$; $\text{pi} = 1006 - 2 = 1004$

$\text{pf}++$; or $++\text{pf}$; $\text{pf} = 4000 + 4 = 4004$ (Since float is of 4 bytes)

$\text{pf} = \text{pf}-3$; $\text{pf} = 4004 - 3*4 = 3992$

$\text{pf} = \text{pf}+5$; $\text{pf} = 3992 + 5*4 = 4012$

$\text{pf}-;$ or $--\text{pf}$; $\text{pf} = 4012 - 4 = 4008$

$\text{pc}++$; or $++\text{pc}$; $\text{pc} = 5000 + 1 = 5001$ (Since char is of 1 byte)

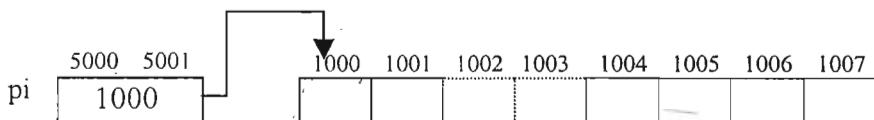
$\text{pc} = \text{pc}-3$; $\text{pc} = 5001 - 3 = 4998$

$\text{pc} = \text{pc}+5$; $\text{pc} = 4998 + 5 = 5003$

$\text{pc}-;$ or $--\text{pc}$; $\text{pc} = 5003 - 1 = 5002$

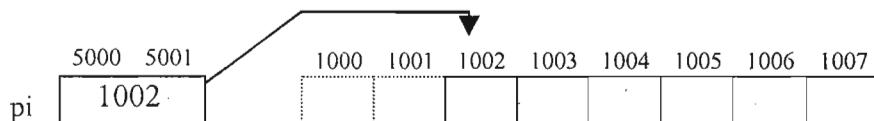
The compiler scales all this arithmetic automatically since it knows the base type of pointer. The arithmetic in the case of char pointer seems to be like ordinary arithmetic because the size of a char is 1 byte. The addresses of variables a, b and c are not affected by these operations, only the pointer moves ahead or backwards.

Suppose pi is an integer pointer that contains address 1000 initially.



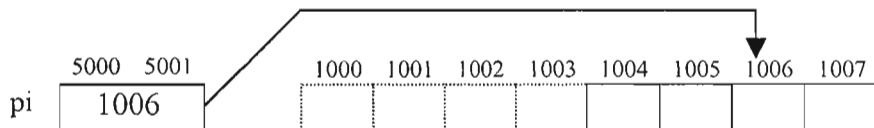
(a) $\text{pi}++$:

After incrementing, pointer pi points to address 1002 and if dereferenced(*pi), then it will return the information stored at bytes 1002 and 1003.



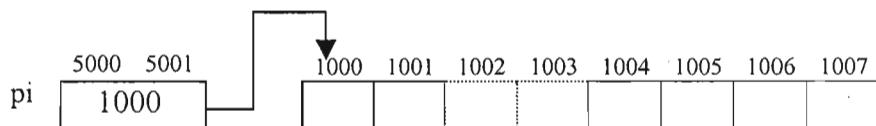
(b) $\text{pi} = \text{pi}+2$:

Now pi contains address 1006, and if dereferenced it will return information stored at bytes 1006 and 1007.



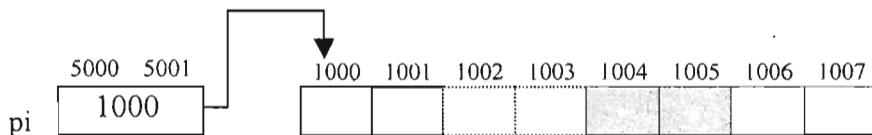
(c) `pi = pi-3;`

Now pi contains address 1000, and if dereferenced it will return information stored at bytes 1000 and 1001.



(d) `printf("%d", *(pi+2));`

Here the pointer will not move to address 1004 since we have not assigned any new value to it. The integer stored at bytes 1004 and 1005 will be printed.



It is important to note that when we move a pointer somewhere else in memory by incrementing/decrementing or adding/subtracting integers then it is not necessary that the pointer still points to a variable of same data type or a valid memory location. The task of allocating memory locations to variables is done by the compiler. We don't know where and in what order it has stored them. We should take care that we move the pointers in such a way that they always point to valid memory locations. In case of arrays, elements are stored in consecutive order. So this arithmetic is generally applied in arrays.

```
/*P8.4 Program to show pointer arithmetic*/
#include<stdio.h>
main( )
{
    int a=5,*pi=&a;
    char b='x',*pc=&b;
    float c=5.5,*pf=&c;
    printf("Value of pi = Address of a = %u\n",pi);
    printf("Value of pc = Address of b = %u\n",pc);
    printf("Value of pf = Address of c = %u\n",pf);
    pi++;
    pc++;
    pf++;
    printf("Now value of pi = %u\n",pi);
    printf("Now value of pc = %u\n",pc);
    printf("Now value of pf = %u\n",pf);
}
```

Output:

Value of pi = Address of a = 1000

Value of pc = Address of b = 4000

Value of pf = Address of c = 8000

Now value of pi = 1002

Now value of pc = 4001

Now value of pf = 8004

```
/*P8.5 Program to understand the postfix/prefix increment/decrement in
a pointer variable of base type int */
#include<stdio.h>
main( )
{
    int a=5;
    int *p;
    p=&a;
    printf("Value of p = Address of a = %u\n",p);
    printf("Value of p = %u\n",++p);
    printf("Value of p = %u\n",p++);
    printf("Value of p = %u\n",--p);
    printf("Value of p = %u\n",p--);
    printf("Value of p = %u\n",p);
}
```

Output:

Value of p = Address of a = 1000

Value of p = 1002

Value of p = 1000

In first printf the address of a = 1000 is printed. In second printf, first the pointer is incremented then its value is printed. Since base type of pointer is int, hence it is incremented by 2. Similarly other printf statements are executed.

Subtraction of a pointer from another pointer of same base type returns an integer, which denotes the number of elements between two pointers. If we have two int pointers ptr1 and ptr2, containing addresses 3000 and 3010 respectively then $\text{ptr2} - \text{ptr1}$ will give 5. (since size of int is 2). This operation is generally performed when both pointer variables point to the elements of same array.

The arithmetic operations that can never be performed on pointers are-

1. Addition, multiplication, division of two pointers.
2. Multiplication between pointer and any number.
3. Division of a pointer by any number.
4. Addition of float or double values to pointers.

8.5 Precedence Of Dereferencing Operator And Increment/Decrement Operators

The precedence level of * operator and increment/decrement operators is same and their associativity is from right to left. There can be confusion while combining these operators in pointer expressions,

so we should use them carefully.

Suppose ptr is an integer pointer and x is an integer variable. Now we'll see how the pointer expressions given below are interpreted.

- $x = *ptr++;$
- $x = *++ptr;$
- $x = ++*ptr;$
- $x = (*ptr)++;$
- (i) $x = *ptr++;$

The expression $*ptr++$ is equivalent to $*(ptr++)$, since these operators associate from right to left. Hence the increment operator will be applied to ptr , and not to $*\text{ptr}$. The increment operator is postfix, so first the value of ptr will be used in the expression and then it will be incremented. Hence firstly the integer pointed to by ptr will be dereferenced and assigned to x and then ptr will be incremented. This is same as-

```
x = *ptr;
ptr = ptr+1;
```

- (ii) $x = *++ptr;$

The expression $*++ptr$ is equivalent to $*(++ptr)$. Here also the increment operator is applied to ptr . The increment operator is prefix, so first ptr will be incremented and then its new value will be used in the expression. Hence firstly the value of ptr is incremented, then value at the new address is dereferenced and assigned to x . This is same as-

```
ptr = ptr+1;
x = *ptr;
```

- (iii) $x = ++*ptr;$

The expression $++*ptr$ is equivalent to $++(*ptr)$. Here the increment operator is applied over $*\text{ptr}$ and not ptr . So here the value of pointer will not change but the value pointed to by the pointer will change i.e., $*ptr$ will increment. Since the increment operator is prefix hence first the value of $*ptr$ will increment and then this value will be assigned to x . This is same as-

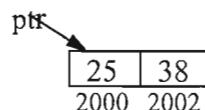
```
*ptr = *ptr+1;
x = *ptr;
```

- (iv) $x = (*ptr)++;$

Here also the increment operator is applied over $*\text{ptr}$ and since it is postfix increment hence first the value of $*\text{ptr}$ will be assigned to x and then it will be incremented. This is same as-

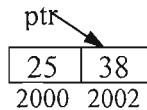
```
x = *ptr;
*ptr = *ptr+1;
```

Let us take an example and understand how these expressions are interpreted. Suppose value at address 2000 is 25, value at address 2002 is 38, ptr is an integer pointer that contains address 2000 hence value of $*\text{ptr}$ is 25. Now we'll see what will be the results in the above four cases, if this is the initial condition in all cases.



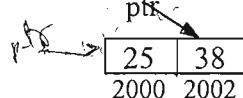
(i) $x = *ptr++;$

Value of $x = 25$, Address contained in $ptr = 2002$, $*ptr = 38$



(ii) $x = *++ptr;$

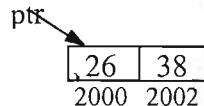
Value of $x = 38$, Address contained in $ptr = 2002$, $*ptr = 38$



(iii) $x = ++*ptr;$

Value of $x = 26$, Address contained in $ptr = 2000$, $*ptr = 26$

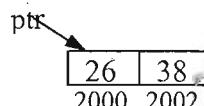
27



(iv) $x = (*ptr)++;$

Value of $x = 25$, Address contained in $ptr = 2000$, $*ptr = 26$

26



From the above four forms of pointer expressions, first one is generally used since it is used to access the elements in an array. In the above statements we have taken increment operator, the same rules stand for decrement operator also.

8.6 Pointer Comparisons

The relational operators $= =$, \neq , $<$, $<=$, $>$, $>=$ can be used with pointers. The operators $= =$ and \neq are used to compare two pointers for finding whether they contain same address or not. They will be equal only if both are NULL or they contain address of same variable. The use of these operators is valid between pointers of same type or between NULL pointer and any other pointer, or between void pointer and any other pointer. The relational operators $<$, $>$, $>=$, $<=$ are valid between pointers of same type. These operations make sense only when both the pointers point to elements of the same array.

8.7 Pointer To Pointer

We know that pointer is a variable that can contain memory address. This pointer variable takes some space in memory and hence it also has an address. We can store the address of a pointer variable in some other variable, which is known as a pointer to pointer variable. Similarly we can have a pointer to pointer to pointer variable and this concept can be extended to any limit, but in practice only pointer to pointer is used. Pointer to pointer is generally used while passing pointer variables to functions.

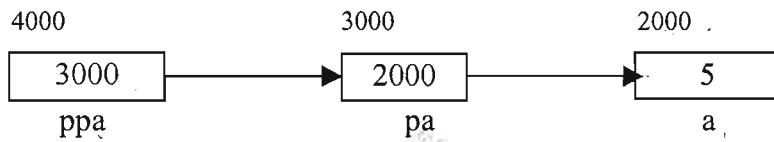
The syntax of declaring a pointer to pointer is as-

```
data_type **pptr;
```

Here variable pptr is a pointer to pointer and it can point to a pointer pointing to a variable of type data_type. The double asterisk used in the declaration informs the compiler that a pointer to pointer is being declared. Now let us take an example-

```
int a = 5;
int *pa = &a;
int **ppa = &pa;
```

Here type of variable a is int, type of variable pa is (int *) or pointer to int, and type of variable ppa is (int **) or pointer to pointer to int.



Here pa is a pointer variable, which contains the address of the variable a and ppa is a pointer to pointer variable, which contains the address of the pointer variable pa.

We know that *pa gives value of a, similarly *ppa will give the value of pa. Now let us see what value will be given by **ppa.

- ~~**ppa~~
- ~~*(*ppa)~~
- ~~*pa~~ (Since *ppa gives pa)
- ~~a~~ (Since *pa gives a)

Hence we can see that **ppa will give the value of a. So to access the value indirectly pointed to by a pointer to pointer, we can use double indirection operator. The table given below will make this concept clear.

Value of a	a	*pa	**ppa	5
Address of a	&a	pa	*ppa	2000
Value of pa	&a	pa	*ppa	2000
Address of pa		&pa	ppa	3000
Value of ppa		&pa	ppa	3000
Address of ppa			&ppa	4000

```

/*P8.6 Program to understand pointer to pointer*/
#include<stdio.h>
main()
{
    int a=5;
    int *pa;
    int **ppa;
    pa=&a;
    ppa=&pa;
    printf("Address of a = %u\n", &a); 65524
    printf("Value of pa = Address of a = %u\n", pa); 65524
    printf("Value of *pa = Value of a = %d\n", *pa); 5
    printf("Address of pa = %u\n", &pa);
    printf("Value of ppa = Address of pa = %u\n", ppa);
    printf("Value of *ppa = Value of pa = %u\n", *ppa);
    printf("Value of **ppa = Value of a = %d\n", **ppa);
    printf("Address of ppa = %u\n", &ppa);
}

```

Output:

Address of a = 65524
 Value of pa = Address of a = 65524
 Value of *pa = Value of a = 5

Address of pa = 65522

Value of ppa = Address of pa = 65522

Value of *ppa = Value of pa = 65524

Value of **ppa = Value of a = 5

Address of ppa = 65520

8.8 Pointers and One Dimensional Arrays

The elements of an array are stored in contiguous memory locations. Suppose we have an array arr[5] of type int.

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

This is stored in memory as-

	5000	5002	5004	5006	5008
arr[0]	1	2	3	4	5
arr[1]					
arr[2]					
arr[3]					
arr[4]					

Here 5000 is the address of first element, and since each element (type int) takes 2 bytes so address of next element is 5002, and so on. The address of first element of the array is also known as the base address of the array. So we have seen that the elements of array are stored sequentially in memory one after another.

In C language, pointers and arrays are closely related. We can access the array elements using pointer expressions. Actually the compiler also accesses the array elements by converting subscript notation to pointer notation. Following are the main points for understanding the relationship of pointers with arrays.

1. Elements of an array are stored in consecutive memory locations.
2. The name of an array is a constant pointer that points to the first element of the array, i.e. it stores the address of the first element, also known as the base address of array.
3. According to pointer arithmetic, when a pointer variable is incremented, it points to the next location of its base type.

For example-

int arr[5] = { 5 , 10 , 15 , 20, 25 }

Here arr[5] is an array that has 5 elements each of type int.

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]
5	10	15	20	25
2000	2002	2004	2006	2008

We can get the address of an element of array by applying & operator in front of subscripted variable name. Hence &arr[0] gives address of 0th element, &arr[1] gives the address of first element and so on. Since array subscripts start from 0, so we'll refer to the first element of array as 0th element and so on.

The following program shows that the elements of an array are stored in consecutive memory locations.

```
/*P8.7 Program to print the value and address of the elements of an
```

```

array  */
#include<stdio.h>
main( )
{
    int arr[5]={5,10,15,20,25};
    int i;
    for(i=0;i<5;i++)
    {
        printf("Value of arr[%d] = %d\n",i,arr[i]);
        printf("Address of arr[%d] = %u\n",i,&arr[i]);
    }
}

```

Output:

Value of arr[0] = 5	Address of arr[0] = 2000
Value of arr[1] = 10	Address of arr[1] = 2002
Value of arr[2] = 15	Address of arr[2] = 2004
Value of arr[3] = 20	Address of arr[3] = 2006
Value of arr[4] = 25	Address of arr[4] = 2008

The name of the array 'arr' denotes the address of 0th element of array which is 2000. The address of 0th element can also be given by &arr[0], so arr and &arr[0] represent the same address. The name of an array is a constant pointer, and according to pointer arithmetic when an integer is added to a pointer then we get the address of next element of same base type. Hence (arr+1) will denote the address of the next element arr[1]. Similarly (arr+2) denotes the address of arr[2] and so on. In other words we can say that the pointer expression (arr+1) points to 1st element of array, (arr+2) points to 2nd element of array and so on.

arr	\rightarrow	Points to 0 th element	\rightarrow	&arr[0]	\rightarrow	2000
arr+1	\rightarrow	Points to 1 st element	\rightarrow	&arr[1]	\rightarrow	2002
arr+2	\rightarrow	Points to 2 nd element	\rightarrow	&arr[2]	\rightarrow	2004
arr+3	\rightarrow	Points to 3 rd element	\rightarrow	&arr[3]	\rightarrow	2006
arr+4	\rightarrow	Points to 4 th element	\rightarrow	&arr[4]	\rightarrow	2008

In general we can write-

The pointer expression (arr+i) denotes the same address as &arr[i].

Now if we dereference arr, then we get the 0th element of array, i.e. expression *arr or *(arr+0) represents 0th element of array. Similarly on derferencing (arr+1) we get the 1st element and so on.

*arr	\rightarrow	Value of 0 th element	\rightarrow	arr[0]	\rightarrow	5
*(arr+1)	\rightarrow	Value of 1 st element	\rightarrow	arr[1]	\rightarrow	10
*(arr+2)	\rightarrow	Value of 2 nd element	\rightarrow	arr[2]	\rightarrow	15
*(arr+3)	\rightarrow	Value of 3 rd element	\rightarrow	arr[3]	\rightarrow	20
*(arr+4)	\rightarrow	Value of 4 th element	\rightarrow	arr[4]	\rightarrow	25

In general we can write-

$*(arr+i) \rightarrow arr[i]$

So in subscript notation the address of an array element is &arr[i] and its value is arr[i], while in pointer

notation the address is $(arr+i)$ and the element is $*(arr+i)$.

```
/*P8.8 Program to print the value and address of elements of an array
using pointer notation*/
#include<stdio.h>
main()
{
    int arr[5]={5,10,15,20,25};
    int i ;
    for(i=0;i<5;i++)
    {
        printf("Value of arr[%d] = %d\t",i,*(arr+i));
        printf("Address of arr[%d] = %u\n",i,arr+i);
    }
}
```

The output of this program is similar to that of P8.7.

Accessing array elements by pointer notation is faster than accessing them by subscript notation, because the compiler ultimately changes the subscript notation to pointer notation and then accesses the array elements.

Array subscripting is commutative, i.e. $arr[i]$ is same as $i[arr]$.

We had seen earlier that-

arr[i] is equivalent to $*(arr + i)$.

Now $*(arr+i)$ is same as $*(i+arr)$, so we can write the above statement as-

arr[i] is equivalent to $*(i + arr)$

Now $*(i+arr)$ can be written as $i[arr]$ in subscript notation, so

arr[i] is equivalent to $i [arr]$

```
/* P8.9 Program to print the value of array elements using pointer and
subscript notation */
#include<stdio.h>
main()
{
    int arr[5]={5,10,15,20,25};
    int i=0;
    for(i=0;i<5;i++)
    {
        printf("Value of arr[%d]=",i);
        printf("%d\t",arr[i]);
        printf("%d\t",*(arr+i));
        printf("%d\t",*(i+arr));
        printf("%d\n",i[arr]);
        printf("Address of arr[%d] = %u\n",i,&arr[i]);
    }
}
```

Output:

Value of arr[0] = 5 5 5 5

Address of arr[0] = 2000

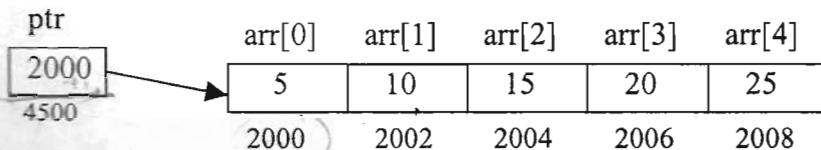
Value of arr[1] = 10	10	10	10
Address of arr[1] = 2002			
Value of arr[2] = 15	15	15	15
Address of arr[2] = 2004			
Value of arr[3] = 20	20	20	20
Address of arr[3] = 2006			
Value of arr[4] = 25	25	25	25
Address of arr[4] = 2008			

8.9 Subscripting Pointer Variables

Suppose we take a pointer variable `ptr`, and initialize it with the address of the 0th element of the array.

`int *ptr;`

~~`ptr = arr; /*We could also write ptr = &arr[0] */`~~



On applying pointer arithmetic and dereferencing we can see that the expression `(ptr+i)` denotes the address of ith element of array and the expression `*(ptr+i)` denotes the value of ith element of array. According to the equivalence of pointer and subscript notations, `*(ptr+i)` can be written as `ptr[i]`. So if we have a pointer variable pointing to the 0th element of array, then we can access the elements of array by subscripting that pointer variable. This equivalence of pointer and subscript notations is used in dynamic arrays and while sending arrays to functions.

Now let us see what is the difference between the name of an array and a pointer variable. The name of an array is a constant pointer hence it will always point to the 0th element of the array. It is not a variable, hence we can't assign some other address to it neither can we move it by incrementing or decrementing.

```
arr = &num;           /*Illegal*/
arr++;              /*Illegal*/
arr = arr-1;         /*Illegal*/
```

But since `ptr` is a pointer variable, all these operations are valid for it.

```
ptr = &num;           /* Now ptr points to variable num*/
ptr++;              /* ptr points to next location */
ptr = ptr-1;         /*ptr points to previous location*/
```

```
/*P8.10 Program to print the value and address of array elements by
subscripting a pointer variable*/
#include<stdio.h>
main()
{
    int arr[5]={5,10,15,20,25};
    int i,*p;
```

```

p=arr;
for(i=0;i<5;i++)
{
    printf("Address of arr[%d]= %u %u %u %u\n", i,&arr[i],arr+i,p+i,
    &p[i]);
    printf("Value of arr[%d]= %d %d %d %d \n", i,arr[i],*(arr+i),*(p+i),
    p[i]);
}
} (ptr)

```

Output:

Address of arr[0]= 2000 2000 2000 2000
 Value of arr[0] = 5 5 5 5
 Address of arr[1]= 2002 2002 2002 2002
 Value of arr[1] = 10 10 10 10
 Address of arr[2]= 2004 2004 2004 2004
 Value of arr[2] = 15 15 15 15
 Address of arr[3]= 2006 2006 2006 2006
 Value of arr[3] = 20 20 20 20
 Address of arr[4]= 2008 2008 2008 2008
 Value of arr[4] = 25 25 25 25

8.10 Pointer to an Array

In the previous section, we had a pointer that pointed to the 0th element of array. We can also declare a pointer that can point to the whole array instead of only one element of array. This pointer is useful when talking about multidimensional arrays. Now we'll see how to declare a pointer to an array.

```
int (*ptr)[10];
```

Here ptr is pointer that can point to an array of 10 integers. Note that it is necessary to enclose the pointer name inside parentheses. Here the type of ptr is 'pointer to an array of 10 integers'.

Note that the pointer that points to the 0th element of array and the pointer that points to the whole array are totally different. The following program shows this-

```

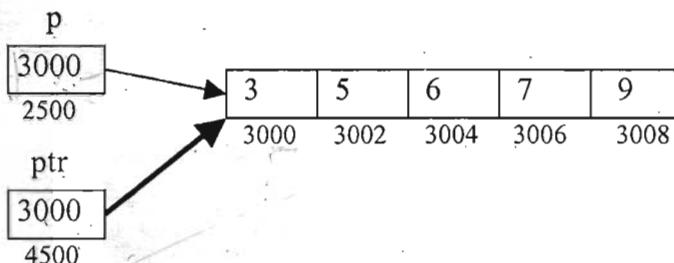
/*P8.11 Program to understand difference between pointer to an integer
and pointer to an array of integers*/
#include<stdio.h>
main()
{
    int *p; /*Can point to an integer*/
    int (*ptr)[5]; /*Can point to an array of 5 integers*/
    int arr[5];
    p=arr; /*Points to 0th element of arr*/
    ptr=arr; /*Points to the whole array arr*/
    printf("p = %u, ptr = %u\n", p, ptr);
    p++;
    ptr++;
    printf("p = %u, ptr = %u\n", p, ptr);
}

```

Output:

```
p = 3000, ptr = 3000
p = 3002, ptr = 3010
```

Here p is pointer that points to 0th element of array arr, while ptr is a pointer that points to the whole array arr. The base type of p is 'int' while base type of ptr is 'an array of 5 integers'. We know that the pointer arithmetic is performed relative to the base size, so if we write ptr++, then the pointer ptr will be shifted forward by 10 bytes. The following figure shows the pointers p and ptr, in this chapter we'll use a darker arrow to denote pointer to an array.



On dereferencing a pointer expression we get a value pointed to by that pointer expression. Pointer to an array points to an array, so on dereferencing it we should get the array, and the name of array denotes the base address. So whenever a pointer to an array is dereferenced, we get the base address of the array to which it points.

```
/*P8.12 Program to dereference a pointer to an array*/
#include<stdio.h>
main()
{
    int arr[5]={3,5,6,7,9};
    int *p=arr;
    int (*ptr)[5]=arr;
    printf("p = %u, ptr = %u\n",p,ptr);
    printf("*p = %d, *ptr = %u\n",*p,*ptr);
    printf("sizeof(p) = %u, sizeof(*p) = %u\n",sizeof(p),sizeof(*p));
    printf("sizeof(ptr) = %u, sizeof(*ptr) = %u\n",sizeof(ptr),
    sizeof(*ptr));
}
```

Output:

```
p = 3000, ptr = 3000
*p = 3, *ptr = 3000
sizeof(p) = 2, sizeof(*p) = 2
sizeof(ptr) = 2, sizeof(*ptr) = 10
```

8.11 Pointers And Two Dimensional Arrays

In a two dimensional array we can access each element by using two subscripts, where first subscript represents row number and second subscript represents the column number. The elements of 2-D array can be accessed with the help of pointer notation also. Suppose arr is a 2-D array, then we can access any element arr[i][j] of this array using the pointer expression *(*(arr+i) + j). Now we'll see how

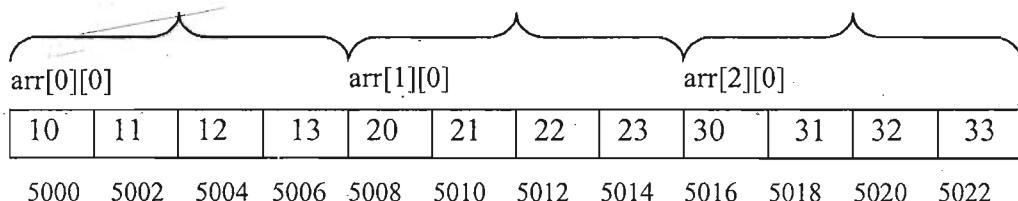
this expression can be derived. Let us take a two dimension array arr[3][4]-

```
int arr[3][4] = { {10, 11, 12, 13}, {20, 21, 22, 23}, {30, 31, 32, 33} };
```

Column 0 Column 1 Column 2 Column 3

Row 0	10	11	12	13
Row 1	20	21	22	23
Row 2	30	31	32	33

We have been talking about 2-D arrays in terms of rows and columns, but since memory in computer is organized linearly it is not possible to store the 2-D array in rows and columns. The concept of rows and columns is only theoretical, actually a 2-D array is stored in row major order i.e. rows are placed next to each other. The following figure shows how the above 2-D array will be stored in memory.



Each row can be considered as a 1-D array, so a two-dimensional array can be considered as a collection of one-dimensional arrays that are placed one after another. In other words we can say that a 2-D array is an array of arrays. So here arr is an array of 3 elements where each element is a 1-D array of 4 integers.

We know that the name of an array is a constant pointer that points to the 0th element of array. In the case of 2-D arrays, 0th element is a 1-D array, so the name of a 2-D array represents a pointer to a 1-D array. For example in the above case, arr is a pointer to 0th 1-D array and contains address 5000. Since arr is a 'pointer to an array of 4 integers', so according to pointer arithmetic, the expression (arr+1) will represent the address 5008 and expression (arr+2) will represent address 5016.

So we can say that arr points to the 0th 1-D array, (arr+1) points to the 1st 1-D array and (arr+2) points to the 2nd 1-D array.

arr	→	10	11	12	13	5000
(arr+1)	→	20	21	22	23	5008
(arr+2)	→	30	31	32	33	5016
arr	-	Points to 0 th element of arr -	Points to 0 th 1-D array -	5000		
arr+1	-	Points to 1 st element of arr -	Points to 1 st 1-D array -	5008		
arr+2	-	Points to 2 nd element of arr -	Points to 2 nd 1-D array -	5016		

In general we can write-

$\text{arr}+i$ Points to i^{th} element of arr \rightarrow Points to i^{th} 1-D array

Since $(\text{arr}+i)$ points to i^{th} element of arr, so on dereferencing it we'll get i^{th} element of arr which is of course a 1-D array. Hence the expression $*(\text{arr}+i)$ gives us the base address of i^{th} 1-D array.

We have proved earlier that the pointer expression $*(\text{arr}+i)$ is equivalent to the subscript expression $\text{arr}[i]$. So $*(\text{arr}+i)$ which is same as $\text{arr}[i]$ gives the base address of i^{th} 1-D array.

$*(\text{arr}+0)$ - arr[0] - Base address of 0^{th} 1-D array - Points to 0^{th} element of 0^{th} 1-D array - 5000
$*(\text{arr}+1)$ - arr[1] - Base address of 1^{st} 1-D array - Points to 0^{th} element of 1^{st} 1-D array - 5008
$*(\text{arr}+2)$ - arr[2] - Base address of 2^{nd} 1-D array - Points to 0^{th} element of 2^{nd} 1-D array - 5016

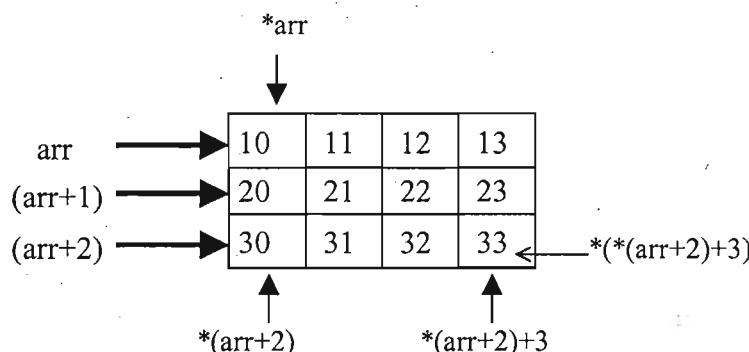
In general we can write-

$*(\text{arr}+i)$ - arr[i] - Base address of i^{th} 1-D array - Points to 0^{th} element of i^{th} 1-D array

Note that both the expressions $(\text{arr}+i)$ and $*(\text{arr}+i)$ are pointers, but their base type is different. The base type of $(\text{arr}+i)$ is 'an array of 4 ints' while the base type of $*(\text{arr}+i)$ or $\text{arr}[i]$ is 'int'.

To access any individual element of our 2-D array, we should be able to access any j^{th} element of the i^{th} 1-D array. Since the base type of $*(\text{arr}+i)$ is 'int' and it contains the address of 0^{th} element of i^{th} 1-D array, so we can get the addresses of subsequent elements in the i^{th} 1-D array by adding integer values to $*(\text{arr}+i)$. For example $*(\text{arr}+i)+1$ will represent the address of 1^{st} element of i^{th} 1-D array and $*(\text{arr}+i)+2$ will represent the address of 2^{nd} element of i^{th} 1-D array. Similarly $*(\text{arr}+i)+j$ will represent the address of j^{th} element of i^{th} 1-D array. On dereferencing this expression we can get the j^{th} element of the i^{th} 1-D array.

arr	Points to 0^{th} 1-D array
*arr	Points to 0^{th} element of 0^{th} 1-D array
(arr+i)	Points to i^{th} 1-D array
*(arr+i)	Points to 0^{th} element of i^{th} 1-D array
<u>*(arr+i)+j</u>	Points to j^{th} element of i^{th} 1-D array
<u>*(*(arr+i)+j)</u>	Represents the value of j^{th} element of i^{th} 1-D array



```

/* P8.13 Program to print the values and address of elements of a
D array */
#include<stdio.h>
main()
{
    int arr[3][4]={
        {10,11,12,13},
        {20,21,22,23},
        {30,31,32,33}
    };
    int i,j;
    for(i=0;i<3;i++)
    {
        printf("Address of %dth array = %u %u\n",i,arr[i],*(arr+i));
        for(j=0;j<4;j++)
            printf("%d %d ",arr[i][j],*(*(arr+i)+j));
        printf("\n");
    }
}

```

Output:

Address of 0th 1-D array = 65000 65000

10 10	11 11	12 12	13 13
-------	-------	-------	-------

Address of 1th 1-D array = 65008 65008

20 20	21 21	22 22	23 23
-------	-------	-------	-------

Address of 2th 1-D array = 65016 65016

30 30	31 31	32 32	33 33
-------	-------	-------	-------

8.12 Subscripting Pointer To An Array

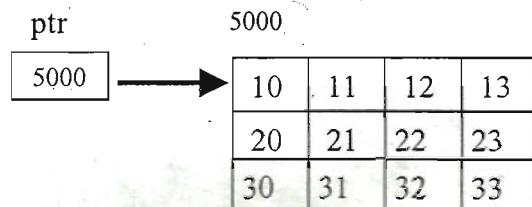
Under the heading ‘Subscripting Pointer Variables’, we had seen how to subscript a pointer variable that contains the base address of a 1-D array. Now we’ll see how to subscript a pointer to an array that contains the base address of a 2-D array.

Suppose arr is a 2-D array with 3 rows and 4 columns and ptr is a pointer to an array of 4 integers and ptr contains the base address of array arr.

```

int arr[3][4] = { {10, 11, 12, 13}, {20, 21, 22, 23}, {30, 31, 32, 33} };
int (*ptr)[4];
ptr = arr;

```



Since ptr is a pointer to an array of 4 integers, so according to pointer arithmetic, ptr+i will point

2-

i^{th} row. On dereferencing $(\text{ptr}+i)$, we get base address of i^{th} row. To access the address of j^{th} element of i^{th} row we can add j to the pointer expression $\ast(\text{ptr}+i)$. So the pointer expression $\ast(\text{ptr}+i)+j$ gives the address of j^{th} element of i^{th} row. So the pointer expression $\ast(\ast(\text{ptr}+i)+j)$ gives the value of the j^{th} element of i^{th} row.

We have studied that the pointer expression $\ast(\ast(\text{ptr}+i)+j)$ is equivalent to subscript expression $\text{ptr}[i][j]$. So if we have a pointer variable containing the base address of 2-D array, then we can access the elements of array by double subscripting that pointer variable.

```
/*P8.14 Program to print elements of a 2-D array by subscripting a pointer
to an array variable*/
#include<stdio.h>
main()
{
    int i, arr[3][4]={{10,11,12,13},{20,21,22,23},{30,31,32,33}};
    int (*ptr)[4];
    ptr=arr;
    printf("%u %u %u\n",ptr,ptr+1,ptr+2);
    printf("%u %u %u\n",*ptr,*ptr+1,*ptr+2);
    printf("%d %d %d\n",**ptr,**ptr+1,**ptr+2);
    printf("%d %d %d\n",ptr[0][0],ptr[1][2],ptr[2][3]);
}
```

Output:

```
5000 5008 5016
5000 5008 5016
10 22 33
10 22 33
```

8.13 Pointers And Three Dimensional Arrays

In a three dimensional array we can access each element by using three subscripts. Let us take a 3-D array-

```
int arr[2][3][2]={
    {
        {5,10},
        {6,11},
        {7,12},
    },
    {
        {20,30},
        {21,31},
        {22,32},
    }
};
```

We can consider a three dimensional array to be an array of 2-D arrays i.e. each element of a 3-D array is considered to be a 2-D array. The 3-D array arr can be considered as an array consisting of two elements where each element is a 2-D array. The name of the array arr is a pointer to the 0^{th} element of the array, so arr points to the 0^{th} 2-D array.

Now let us see how we can access any element of a 3d array using pointer notation.

arr	Points to 0 th 2-D array
arr+i	Points to i th 2-D array
*(arr+i)	Gives base address of i th 2-D array, so points to 0 th element of i th 2-D array, each element of 2-D array is a 1-D array, so it points to 0 th 1-D array of i th 2-D array
*(arr+i)+j	Points to j th 1-D array of i th 2-D array
((arr+i)+j)	Gives base address of j th 1-D array of i th 2-D array, so it points to 0 th element of j th 1-D array of i th 2-D array
((*(arr+i)+j)+k)	Points to k th element of j th 1-D array of i th 2-D array
((*(*(arr+i)+j)+k)	Gives the value of k th element of j th 1-D array of i th 2-D array

So we can see that the pointer expression *(*(*(*(arr+i)+j)+k) is equivalent to the subscript expression arr[i][j][k].

Earlier we have seen that the expression *(arr+i) is equivalent to arr[i] and the expression *(*(*(arr+i)+j) is equivalent to arr[i][j]. So we can say that arr[i] represents the base address of ith 2-D array and arr[i][j] represents the base address of the jth 1-D array of ith 2-D array.

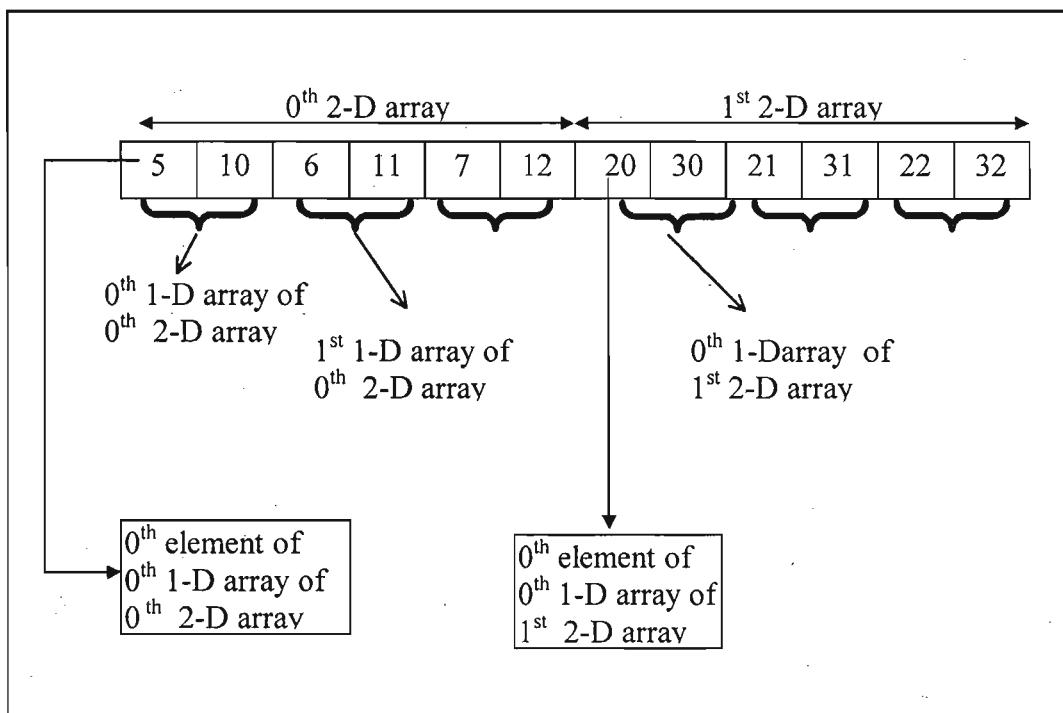
```
/*P8.15 Program to print the elements of 3-D array using pointer notation*/
#include<stdio.h>
main()
{
    int arr[2][3][2]={
        {
            {5,10},
            {6,11},
            {7,12},
        },
        {
            {20,30},
            {21,31},
            {22,32},
        }
    };
    int i,j,k;
    for(i=0;i<2;i++)
        for(j=0;j<3;j++)
    {
        for(k=0;k<2;k++)
            printf(" %d\t", *(*(*(arr+i)+j)+k));
            printf(" \n");
    }
}
```

Output:

```
5 10
6 11
```

7	12
20	30
21	31
22	32

The following figure shows how the 3-D array used in the above program is stored in memory.



8.14 Pointers And Functions

The arguments to the functions can be passed in two ways-

1. Call by value
2. Call by reference

In call by value, only the values of arguments are sent to the function while in call by reference, addresses of arguments are sent to the function. In call by value method, any changes made to the formal arguments do not change the actual argument. In call by reference method, any changes made to the formal arguments change the actual arguments also. C uses only call by value when passing arguments to a function, but we can simulate call by reference by using pointers.

All the functions that we had written so far used call by value method. Here is another simple program that uses call by value-

```
/*P8.16 Program to explain call by value*/
#include<stdio.h>
main()
{
    int a=5, b=8;
```

```

printf("Before calling the function, a and b are %d,%d\n",a,b);
value(a,b);
printf("After calling the function, a and b are %d,%d\n",a,b);
}
value(int x,int y)
{
    x++;
    y++;
    printf("In function changes are %d,%d\n",x,y);
}

```

Output:

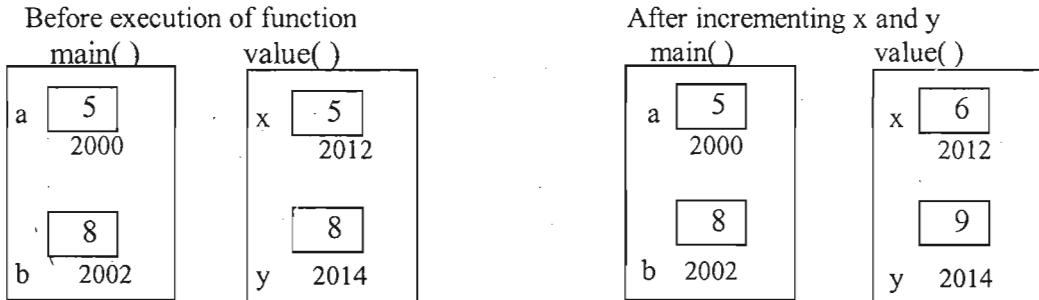
Before calling the function, a and b are 5, 8

In function changes are 6 , 9

After calling the function, a and b are 5, 8

Here a and b are variables declared in the function main() while x and y are declared in the function value().These variables reside at different addresses in memory. Whenever the function value() is called, two variables are created named x and y and are initialized with the values of variables a and b. This type of parameter passing is called call by value since we are only supplying the values of actual arguments to the calling function. Any operation performed on variables x and y in the function value(), will not affect variables a and b.

Before calling the function value() , the value of a = 5 and b = 8. The values of a and b are copied into x and y. Since the memory locations of x, y and a, b are different, so when the values of x and y are incremented, there will be no effect on the values of a and b. Therefore after calling the function, a and b are same as before calling the function and have the value 5 and 8.



Although C does not use call by reference, but we can simulate it by passing addresses of variables as arguments to the function. To accept the addresses inside the function, we'll need pointer variables. Here is a program that simulates call by reference by passing addresses of variables a and b.

```

/* P8.17 program to explain call by reference*/
#include<stdio.h>
main()
{
    int a=5;
    int b=8;
    printf("Before calling the function, a and b are %d,%d\n",a,b);
    ref(&a,&b);
}

```

```

    printf("After calling the function, a and b are %d,%d\n",a,b);
}
ref(int *p,int *q)
{
    (*p)++;
    (*q)++;
    printf("In function changes are %d,%d\n",*p,*q);
}

```

Output:

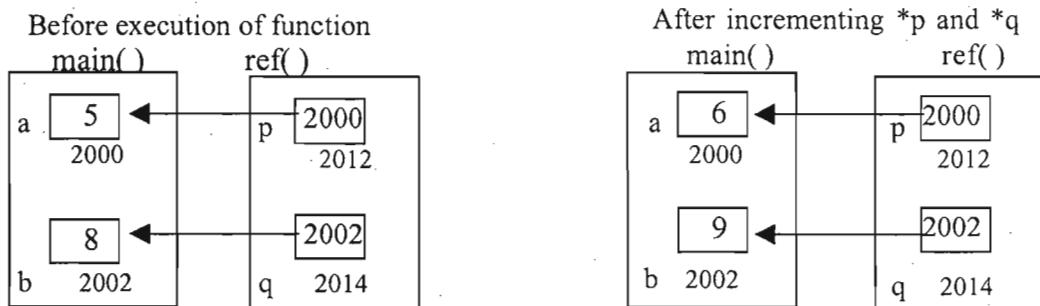
Before calling the function, a and b are 5, 8

In function changes are 6 , 9

After calling the function, a and b are 6, 9

Here we are passing addresses of variables a and b in the function call. So the receiving formal arguments in the function declaration should be declared of pointer type. Whenever function ref() is called, two pointer to int variables, named p and q are created and they are initialized with the addresses of a and b. Now if we dereference pointers p and q, we will be able to access variables a and b from function ref().

The main() accesses the memory locations occupied by variables a and b by writing their names, while ref() accesses the same memory locations indirectly by writing *p, *q.



Before calling the function ref(), the value of a = 5 and b = 8. The value of actual arguments are copied into pointer variables p and q, and here the actual arguments are addresses of variables a and b. Since p contains address of variable a, so we can access variable a inside ref() by writing *p, similarly variable b can be accessed by writing *q.

Now (*p)++ means value at address 2000 (which is 5) is incremented. Similarly (*q)++ means value at address 2002 (which is 8) is incremented. Now the value of *p = 6 and *q = 9. When we come back to main(), we see that the values of variable a and b have changed. This is because the function ref() knew the addresses of a and b, so it was able to access them indirectly.

So in this way we could simulate call by reference by passing addresses of arguments. This method is mainly useful when the called function has to return more than one values to the function.

8.15 Returning More Than One Value From A Function

We have studied that we can return only one value from a function through return statement. This limitation can be overcome by using call by reference. Let us take a simple example to understand this concept. Suppose we want a function to return the sum, difference and product of two numbers passed

to it. If we use return statement then we will have to make three different functions with one return statement in each. The following program shows how we can return all these values from a single function.

```
/*P8.18 Program to show how to return more than one value from a function
using call by reference*/
#include<stdio.h>
main()
{
    int a,b,sum,diff,prod;
    a=6;
    b=4;
    func(a,b,&sum,&diff,&prod);
    printf("Sum = %d, Difference = %d, Product = %d\n",sum,diff,prod);
}
func(int x,int y,int *ps,int *pd,int *pp)
{
    *ps=x+y;
    *pd=x-y;
    *pp=x*y;
}
```

Output:

Sum = 10, Difference = 2 , Product = 24

In func() variables a and b are passed by value while variables sum, diff, prod are passed by reference. The function func() knows the addresses of variables sum, diff and prod, so it accesses these variables indirectly using pointers and changes their values.

8.16 Function Returning Pointer

We can have a function that returns a pointer. The syntax of declaration of such type of function is-

type *func(type1, type 2, ...);

For example-

```
float *fun(int ; char ); /* This function returns a pointer to float. */
int *func(int , int ); /* This function returns a pointer to int. */
```

While returning a pointer, make sure that the memory address returned by the pointer will exist even after the termination of function. For example a function of this form is invalid-

```
main()
{
    int *ptr;
    ptr=func();
    .....
}

int *func()
{
    int x=5;
    int *p=&x;
    .....
```

```
    return p;
}
```

Here we are returning a pointer which points to a local variable. We know that a local variable exists only inside the function. Suppose the variable x is stored at address 2500, so the value of p will be 2500 and this value will be returned by the function func(). As soon as func() terminates, the local variables x will cease to exist.

The address returned by func() is assigned to pointer variable ptr inside main(), so now ptr will contain address 2500. When we dereference ptr, we are trying to access the value of a variable that no longer exists. So never return a pointer that points to a local variable. Now we'll take a program that uses a function returning pointer.

```
/*P8.19 Program to show the use of a function that returns pointer*/
#include<stdio.h>
int *fun(int *p,int n);
main()
{
    int arr[10]={1,2,3,4,5,6,7,8,9,10},n,*ptr;
    n=5;
    ptr=fun(arr,n);
    printf("Value of arr = %u, Value of ptr = %u, value of *ptr = %d\n",
    arr,ptr,*ptr);
}
int *fun(int *p,int n)
{
    p=p+n;
    return p;
}
```

Output:

Value of arr = 65104, Value of ptr = 65114, Value of *ptr = 6

8.17 Passing a 1-D Array to a Function

In the previous chapter we had studied that when an array is passed to a function, the changes made inside the function affect the original array. This is because the function gets access to the original array. Here is a simple program that verifies this fact.

```
/*P8.20 Program to show that changes to the array made inside the function
affect the original array*/
#include<stdio.h>
main()
{
    int i,arr[5]={3,6,2,7,1};

    func(arr);
    printf("Inside main() : ");
    for(i=0;i<5;i++)
        printf("%d ",arr[i]);
    printf("\n");
```

```

func(int a[])
{
    int i;
    printf("Inside func() : ");
    for(i=0;i<5;i++)
    {
        a[i]=a[i]+2;
        printf("%d",a[i]);
    }
    printf("\n");
}

```

Output:

Inside func() : 5 8 4 9 3

Inside main() : 5 8 4 9 3

Now after studying about pointers we are in a position to understand what actually happens when an array is passed to a function. There are three ways of declaring the formal parameter, which has to receive the array. We can declare it as an unsized or sized array or we can declare it as a pointer.

```

func(int a[])
{
    .....
}

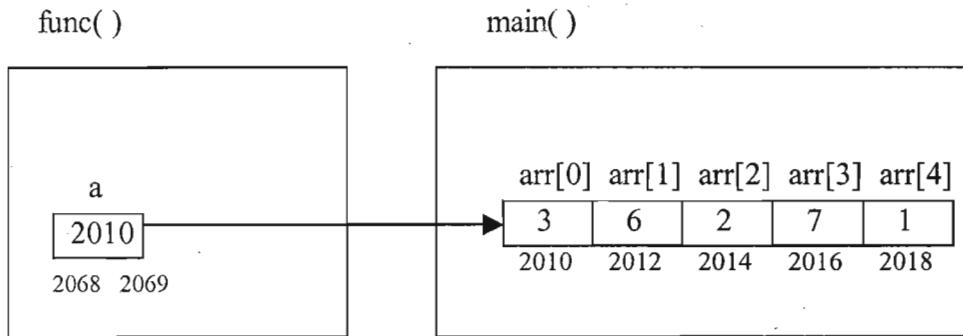
func(int a[5]);
{
    .....

}

func(int *a);
{
    .....
}

```

In all the three cases the compiler reserves space only for a pointer variable inside the function. In the function call, the array name is passed without any subscript or address operator. Since array name represents the address of first element of array, hence this address is assigned to the pointer variable in the function. So inside the function we have a pointer that contains the base address of the array. In the above program, the argument **a** is declared as a pointer variable whose base type is **int**, and it is initialized with the base address of array arr. We have studied that if we have a pointer variable containing the base address of an array, then we can access any array element either by pointer notation or subscript notation. So inside the function, we can access any *i*th element of arr by writing *(a+i) or a[i]. Since we are directly accessing the original array hence all the changes made to the array in the called function are reflected in the calling function.



The following program will illustrate the point that we have discussed.

```
/*P8.21 Program to verify the fact that when an array is passed to a
function, the receiving argument is declared as a pointer */
#include<stdio.h>
main()
{
    float f_arr[5]={1.4,2.5,3.7,4.1,5.9};
    int i_arr[5]={1,2,3,4,5};
    char c_arr[5]={'a','b','c','d','e'};
    printf("Inside main() : ");
    printf("Size of arr = %u\t",sizeof(f_arr));
    printf("Size of arr = %u\t",sizeof(i_arr));
    printf("Size of arr = %u\n",sizeof(c_arr));
    func(f_arr,i_arr,c_arr);
}
func(float f[],int *i,char c[])
{
    printf("Inside func() : ");
    printf("Size of f = %d\t",sizeof(f));
    printf("Size of i = %d\t",sizeof(i));
    printf("Size of c = %d\n",sizeof(c));
}
```

Output:

Inside main() : Size of f_arr = 20	Size of i_arr = 10	Size of c_arr = 5
Inside func() : Size of f = 2	Size of i = 2	Size of c = 2

Inside the function func(), f, i, c are declared as pointers , and this is evident by the fact that the size of each one of them is 2 bytes.

8.18 Passing a 2-D Array to a Function

We have studied in the previous chapter that whenever a multidimensional array is passed to a function, then it is optional to specify the leftmost dimension but all other dimensions must be specified. So if we have a 2-D array with 3 row and 4 columns, then the definition of a function that accepts it can be written in these two ways-

```
func(int a[3][4])
{
```

```
.....
}
func(int a[][4])
{
.....
}
```

Any changes made to the array in the function will be reflected in the calling function.

Whenever a 2-D array is passed to a function, the function actually receives a pointer to a 1-D array, where the size of 1-D array is equal to the number of columns. For example in the above case the function receives a pointer to an array of 4 integers. So we may write the function definition in this form also-

```
func(int (*a)[4])
{
.....
}
```

Here **a** is declared as a pointer to an array of 4 integers, and it is initialized with the base address of the original 2-D array. Now inside the function we can use either pointer notation or subscript notation to access the elements. The following program will make these points clear.

```
/*P8.22 Program to pass a 2-D array to a function*/
#include<stdio.h>
main()
{
    int i,j,arr[3][4]={
        {11,12,13,14},
        {15,16,17,18},
        {19,20,21,22},
    };
    printf("Inside main() : sizeof(arr) = %u\n",sizeof(arr));
    func(arr);
    printf("Contents of array after calling func() are :\n");
    for(i=0;i<3;i++)
    {
        for(j=0;j<4;j++)
            printf("%d ",arr[i][j]);
        printf("\n");
    }
}
func(int (*a)[4])
{
    int i,j;
    printf("Inside func() : sizeof(a) = %u\n",sizeof(a));
    printf("Inside func() : sizeof(*a) = %u\n",sizeof(*a));
    for(i=0;i<3;i++)
        for(j=0;j<4;j++)
            a[i][j]=a[i][j]+2;
}
```

Output:

Inside main() : sizeof(arr) = 24

Inside func() : sizeof(a) = 2

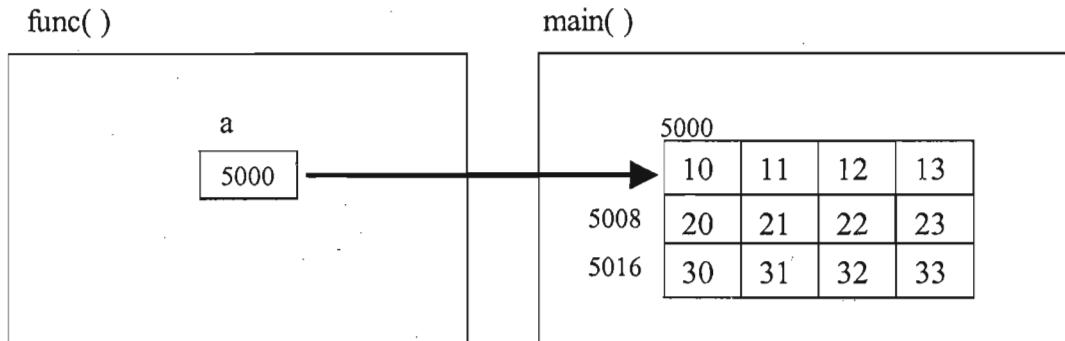
Inside func() : sizeof(*a) = 8

Contents of array after calling func() are -

13 14 15 16

17 18 19 20

21 22 23 24



8.19 Array Of Pointers

We can declare an array that contains pointers as its elements. Every element of this array is a pointer variable that can hold address of any variable of appropriate type. The syntax of declaring an array of pointers is similar to that of declaring arrays except that an asterisk is placed before the array name.

```
datatype *arrayname[size];
```

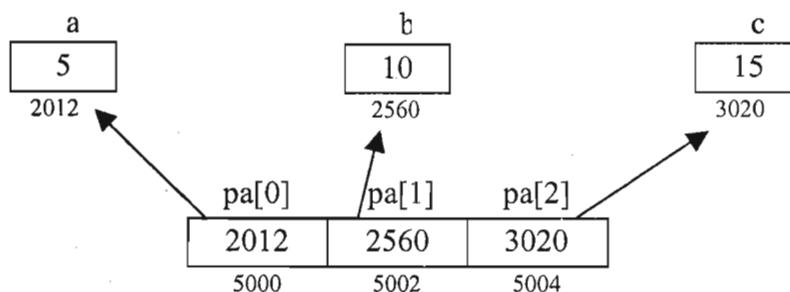
For example to declare an array of size 10 that contains integer pointers we can write-

```
int *arrp[10];
```

```
/*P8.23 Program for understanding the concept of array of pointers*/
#include<stdio.h>
main()
{
    int *pa[3];
    int i, a=5, b=10, c=15;
    pa[0]=&a;
    pa[1]=&b;
    pa[2]=&c;
    for(i=0; i<3; i++)
    {
        printf("pa[%d] = %u\t", i, pa[i]);
        printf("*pa[%d] = %d\n", i, *pa[i]);
    }
}
```

Output:

```
pa[0] = 2012      *pa[0] = 5
pa[1] = 2560      *pa[1] = 10
pa[2] = 3020      *pa[2] = 15
```



Here `pa` is declared as an array of pointers. Every element of this array is a pointer to an integer. `pa[i]` gives the value of the i^{th} element of '`pa`' which is an address of any int variable and `*pa[i]` gives the value of that int variable.

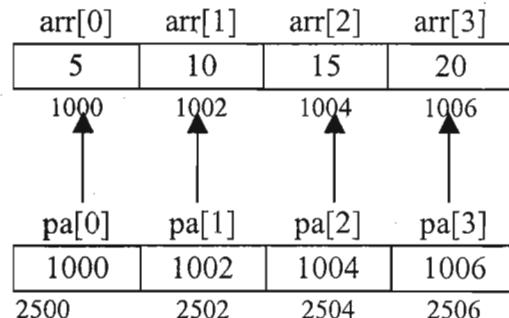
The array of pointers can also contain addresses of elements of another array.

```
/*P8.24 Program for understanding array of pointers*/
#include<stdio.h>
main()
{
    int i,arr[4]={5,10,15,20};
    int *pa[4];
    for(i=0;i<4;i++)
        pa[i] = &arr[i];
    for(i=0;i<4;i++)
    {
        printf("pa[%d] = %u\t",i,pa[i]);
        printf("*pa[%d] = %d\n",i,*pa[i]);
    }
}
```

Output:

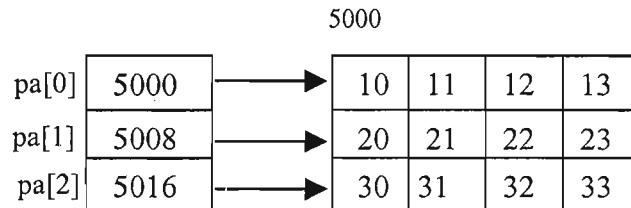
<code>pa[0] = 1000</code>	<code>*pa[0] = 5</code>
<code>pa[1] = 1002</code>	<code>*pa[1] = 10</code>
<code>pa[2] = 1004</code>	<code>*pa[2] = 15</code>
<code>pa[3] = 1006</code>	<code>*pa[3] = 20</code>

Here '`pa`' is declared as array of pointers. Each element of this array contains the address of each element of array '`arr`'.



Now we'll take a 2-D array arr with 3 rows and 4 columns. An array of pointers of size 3 is declared and each pointer in this array is assigned the address of 0th element of each row of the 2-D array, i.e. ith element of pa is a pointer to 0th element of ith row of a 2-D array. This can be done as-

```
int arr[3][4]={{10,11,12,13},{20,21,22,23},{30,31,32,33}};
int *pa[3];
for(i=0;i<3;i++)
    pa[i]=arr[i];
```



Now let us see how we can access the elements of the 2-D array arr using the array of pointers pa. pa[0] is a pointer to the 0th element of 0th 1-D array of the array arr, similarly pa[i] will be a pointer to 0th element of ith 1-D array of arr.

Since base type of pointer pa[i] is int, so if we want the address of jth element of ith 1-D array then we can add j to the expression pa[i]. Hence the expression pa[i]+j will give the address of jth element of ith 1-D array. So the expression *(pa[i]+j) will give the value of the jth element of ith 1-D array.

We know that pa[i] is equivalent to *(pa+i). So the above expression can be written as *(*(pa+i) +j), and we know that this expression is equivalent to pa[i][j]. So finally we can access the jth element of ith 1-D array by writing pa[i][j].

```
/*P8.25 program for understanding array of pointers*/
#include<stdio.h>
main()
{
    int i,j,arr[3][4]={{10,11,12,13},{20,21,22,23},{30,31,32,33}};
    int *pa[3];
    for(i=0;i<3;i++)
        pa[i]=arr[i];
    for(i=0;i<3;i++)
    {
        for(j=0;j<4;j++)
            printf("%d ",pa[i][j]);
        printf("\n");
    }
}
```

Output:

```
10 11 12 13
20 21 22 23
30 31 32 33
```

8.20 void Pointers

We have studied that a pointer should be assigned an address of the same type as mentioned in pointer

declaration. For example if we have a pointer to int, then it would be incorrect to assign the address of a float variable to it. But an exception to this rule is a pointer to void. A pointer to void is a generic pointer that can point to any data type. The syntax of declaration of a void pointer is-

```
void *vpt;
```

Here void is a keyword and vpt is declared as a pointer of void type. For example-

```
int i = 2, *ip = &i;
float f = 2.3, *fp = &f;
double d;
void *vp;
ip = fp;           /*Incorrect */
vp = ip;           /*Correct*/
vp = fp;           /*Correct */
vp = &d;           /*Correct */
```

We can assign address of any data type to a void pointer and a void pointer can be assigned to a pointer of any data type.

A void pointer can't be dereferenced simply by using indirection operator. Before dereferencing, it should be type cast to appropriate pointer data type. For example if vp is a void pointer and it holds the address of an integer variable then we can't dereference it just by writing *vp. We'll have to write *(int *)vp, where leftmost asterisk is the indirection operator and (int *) is used for typecasting. Similarly pointer arithmetic can't be performed on void pointers without typecasting.

```
/*P8.26 Program to understand the dereferencing of a void pointer*/
#include<stdio.h>
main( )
{
    int a=3;
    float b=3.4, *fp=&b;
    void *vp;
    vp=&a;
    printf("Value of a = %d\n",*(int *)vp);
    *(int *)vp=12;
    printf("Value of a = %d\n",*(int *)vp);
    vp=fp;
    printf("Value of b = %f\n",*(float *)vp);
}
```

Output:

```
Value of a = 3
Value of a = 12
Value of b = 3.400000
```

```
/*P8.27 Program to understand pointer arithmetic in void pointers */
#include<stdio.h>
main()
{
    int i;
```

```

float a[4]={1.2,2.5,3.6,4.6};
void *vp;
vp=a;
for(i=0;i<4;i++)
{
    printf("%.1f\t",*(float *)vp);
    (float *)vp=(float *)vp+1; /*Can't write vp=vp+1*/
}
printf("\n");
}

```

Output:

1.2 2.5 3.6 4.6

void pointers are generally used to pass pointers to functions which have to perform same operations on different data types.

8.21 Dynamic Memory Allocation

The memory allocation that we have done till now was static memory allocation. The memory that could be used by the program was fixed i.e. we could not increase or decrease the size of memory during the execution of program. In many applications it is not possible to predict how much memory would be needed by the program at run time. For example if we declare an array of integers-

```
int emp_no[200];
```

In an array, it is must to specify the size of array while declaring, so the size of this array will be fixed during runtime. Now two types of problems may occur. The first case is that the number of values to be stored is less than the size of array and hence there is wastage of memory. For example if we have to store only 50 values in the above array, then space for 150 values(300 bytes) is wasted. In second case our program fails if we want to store more values than the size of array, for example if there is need to store 205 values in the above array.

To overcome these problems we should be able to allocate memory at run time. The process of allocating memory at the time of execution is called dynamic memory allocation. The allocation and release of this memory space can be done with the help of some built-in-functions whose prototypes are found in alloc.h and stdlib.h header files. These functions take memory from a memory area called heap and release this memory whenever not required, so that it can be used again for some other purpose.

Pointers play an important role in dynamic memory allocation because we can access the dynamically allocated memory only through pointers.

8.21.1 malloc()

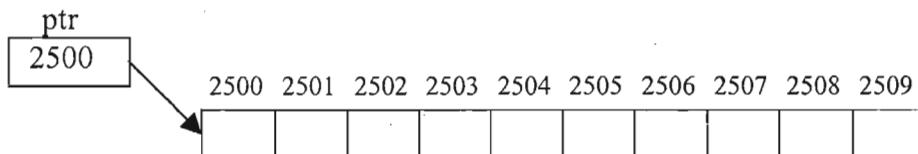
Declaration: void *malloc(size_t size);

This function is used to allocate memory dynamically. The argument *size* specifies the number of bytes to be allocated. The type *size_t* is defined in stdlib.h as unsigned int. On success, malloc() returns a pointer to the first byte of allocated memory. The returned pointer is of type void, which can be type cast to appropriate type of pointer. It is generally used as-

```
ptr = (datatype *) malloc ( specified size );
```

Here *ptr* is a pointer of type *datatype*, and *specified size* is the size in bytes required to be reserved memory. The expression `(datatype *)` is used to typecast the pointer returned by malloc(). For example-

```
int *ptr;
ptr = ( int * ) malloc ( 10 );
```



This allocates 10 contiguous bytes of memory space and the address of first byte is stored in the pointer variable ptr. This space can hold 5 integers. The allocated memory contains garbage value. We can use sizeof operator to make the program portable and more readable.

```
ptr = ( int * ) malloc ( 5 * sizeof ( int ) );
```

This allocates the memory space to hold five integer values.

If there is not sufficient memory available in heap then malloc() returns NULL. So we should always check the value returned by malloc().

```
ptr = ( float * ) malloc(10*sizeof(float) );
if ( ptr == NULL )
    printf("Sufficient memory not available");
```

Unlike memory allocated for variables and arrays, dynamically allocated memory has no name associated with it. So it can be accessed only through pointers. We have a pointer which points to the first byte of the allocated memory and we can access the subsequent bytes using pointer arithmetic.

```
/*P8.28 Program to understand dynamic allocation of memory*/
#include<stdio.h>
#include<alloc.h>
main()
{
    int *p,n,i;
    printf("Enter the number of integers to be entered : ");
    scanf("%d",&n);
    p=(int *)malloc(n*sizeof(int));
    if(p==NULL)
    {
        printf("Memory not available\n");
        exit(1);
    }
    for(i=0;i<n;i++)
    {
        printf("Enter an integer : ");
        scanf("%d",p+i);
    }
    for(i=0;i<n;i++)
        printf("%d\t",*(p+i));
}
```

The function malloc() returns a void pointer and we have studied that a void pointer can be assigned to any type of pointer without typecasting. But we have used typecasting because it is a good practice to do so and moreover it also ensures compatibility with C++.

8.21.2 calloc()

Declaration: void *calloc(size_t n, size_t size);

The calloc() function is used to allocate multiple blocks of memory. It is somewhat similar to malloc() function except for two differences. The first one is that it takes two arguments. The first argument specifies the number of blocks and the second one specifies the size of each block. For example-

```
ptr = ( int * ) calloc ( 5 , sizeof(int) );
```

This allocates 5 blocks of memory, each block contains 2 bytes and the starting address is stored in the pointer variable ptr, which is of type int. An equivalent malloc() call would be-

```
ptr = ( int * ) malloc ( 5 * sizeof(int) );
```

Here we have to do the calculation ourselves by multiplying, but calloc() function does the calculation for us.

The other difference between calloc() and malloc() is that the memory allocated by malloc() contains garbage value while the memory allocated by calloc() is initialized to zero. But this initialization by calloc() is not very reliable, so it is better to explicitly initialize the elements whenever there is need to do so.

Like malloc(), calloc() also returns NULL if there is not sufficient memory available in the heap.

8.21.3 realloc()

Declaration: void *realloc(void *ptr, size_t newsize)

We may want to increase or decrease the memory allocated by malloc() or calloc(). The function realloc() is used to change the size of the memory block. It alters the size of the memory block without losing the old data. This is known as reallocation of memory.

This function takes two arguments, first is a pointer to the block of memory that was previously allocated by malloc() or calloc() and second one is the new size for that block. For example-

```
ptr = (int *) malloc ( size );
```

This statement allocates the memory of the specified size and the starting address of this memory block is stored in the pointer variable ptr. If we want to change the size of this memory block, then we can use realloc() as-

```
ptr = (int *) realloc ( ptr , newsize );
```

This statement allocates the memory space of newsize bytes, and the starting address of this memory block is stored in the pointer variable ptr. The newsize may be smaller or larger than the old size. If the newsize is larger, then the old data is not lost and the newly allocated bytes are uninitialized. The starting address contained in ptr may change if there is not sufficient memory at the old address to store all the bytes consecutively. This function moves the contents of old block into the new block and the data of the old block is not lost. On failure, realloc() returns NULL.

```
/*P8.29 program to understand the use of realloc() function*/
#include<stdio.h>
#include<alloc.h>
main()
{
    int i,*ptr;
    ptr=(int *)malloc(5*sizeof(int));
}
```

```

if(ptr==NULL)
{
    printf("Memory not available\n");
    exit(1);
}
printf("Enter 5 integers : ");
for(i=0;i<5;i++)
    scanf("%d ",ptr+i);
ptr=(int *)realloc(ptr,9*sizeof(int)); /*Allocate memory for 4 more
integers*/
if(ptr==NULL)
{
    printf("Memory not available\n");
    exit(1);
}
printf("Enter 4 more integers : ");
for(i=5;i<9;i++)
    scanf("%d",ptr+i);
for(i=0;i<9;i++)
    printf("%d ",*(ptr+i));
}

```

8.21.4 free()

Declaration: void free(void *p)

The dynamically allocated memory is not automatically released; it will exist till the end of program. If we have finished working with the memory allocated dynamically, it is our responsibility to release that memory so that it can be reused. The function free() is used to release the memory space allocated dynamically. The memory released by free() is made available to the heap again and can be used for some other purpose. For example-

```
free ( ptr );
```

Here ptr is a pointer variable that contains the base address of a memory block created by malloc() or calloc(). Once a memory location is freed it should not be used. We should not try to free any memory location that was not allocated by malloc(), calloc() or realloc().

When the program terminates all the memory is released automatically by the operating system but it is a good practice to free whatever has been allocated dynamically. We won't get any errors if we don't free the dynamically allocated memory, but this would lead to memory leak i.e. memory is slowly leaking away and can be reused only after the termination of program. For example consider this function-

```

func()
{
    int *ptr;
    ptr=(int *)malloc(10*sizeof(int));
    .....
}
```

Here we have allocated memory for 10 integers through malloc(), so each time this function is called, space for 10 integers would be reserved. We know that the local variables vanish when the function terminates, and since ptr is a local pointer variable so it will be deallocated automatically at the termination of function. But the space allocated dynamically is not deallocated automatically, so that space remains

there and can't be used, leading to memory leaks. We should free the memory space by putting a call to `free()` at the end of the function.

Since the memory space allocated dynamically is not released after the termination of function, so it is valid to return a pointer to dynamically allocated memory. For example-

```
int *func()
{
    int *ptr;
    ptr=(int*)malloc(10*sizeof(int));
    .....
    return ptr;
}
```

Here we have allocated memory through `malloc()` in `func()`, and returned a pointer to this memory. Now the calling function receives the starting address of this memory, so it can use this memory. Note that now the call to function `free()` should be placed in the calling function when it has finished working with this memory. Here `func()` is declared as a function returning pointer. Recall that it is not valid to return address of a local variable since it vanishes after the termination of function.

8.21.5 Dynamic Arrays

The memory allocated by `malloc()`, `calloc()` and `realloc()` is always made up of contiguous bytes. Moreover in C there is an equivalence between pointer notation and subscript notation i.e. we can apply subscripts to a pointer variable. So we can access the dynamically allocated memory through subscript notation also. We can utilize these features to create dynamic arrays whose size can vary during run time.

Now we'll rewrite the program P8.28 using subscript notation.

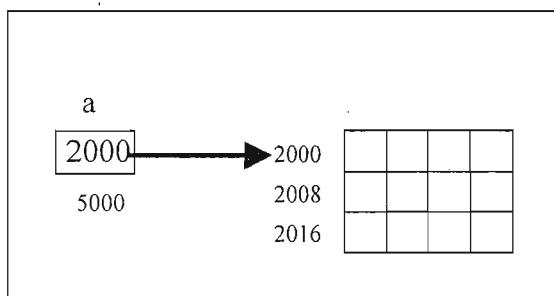
```
/*P8.30 Program to access dynamically allocated memory as a 1d array*/
#include<stdio.h>
#include<alloc.h>
main()
{
    int *p,n,i;
    printf("Enter the number of integers to be entered : ");
    scanf("%d",&n);
    p=(int *)malloc(n*sizeof(int));
    if ( p == NULL)
    {
        printf("Memory not available\n");
        exit(1);
    }
    for(i=0;i<n;i++)
    {
        printf("Enter an integer : ");
        scanf("%d",&p[i]);
    }
    for(i=0;i<n;i++)
        printf("%d\t",p[i]);
}
```

In this way we can simulate a 1-D array for which size is entered at execution time.

Now we'll see how to create a dynamically allocated 2-D array. In the next program we have used a pointer to an array to dynamically allocate a 2-D array. Here the number of columns is fixed while the number of rows can be entered at run time.

```
/*P8.31 Program to dynamically allocate a 2-D array using pointer to an
array*/
#include<stdio.h>
#include<alloc.h>
main()
{
    int i,j,rows;
    int (*a)[4];
    printf("Enter number of rows : ");
    scanf("%d", &rows);
    a=(int (*)[4])malloc(rows*4*sizeof(int));
    for(i=0;i<rows;i++)
        for(j=0;j<4;j++)
    {
        printf("Enter a[%d] [%d] : ",i,j);
        scanf("%d",&a[i][j]);
    }
    printf("The matrix is :\n");
    for(i=0;i<rows;i++)
    {
        for(j=0;j<4;j++)
            printf("%5d",a[i][j]);
        printf("\n");
    }
    free(a);
}
```

Suppose the number of rows entered is 3. The following figure shows how the dynamically allocated memory is accessed using pointer to an array. Since there are 3 rows and 4 columns so we'll allocate 24 bytes through malloc(), and the address returned by malloc() is assigned to **a**. The return value of malloc() is cast appropriately.



Now we'll allocate a 2-D array using array of pointers. Here the number of rows is fixed while the number of columns can be entered at run time.

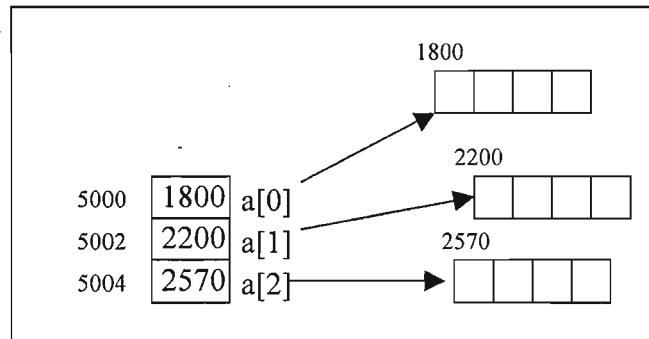
```
/*P8.32 Program to dynamically allocate a 2-D array using array of pointers*/
```

```

#include<stdio.h>
#include<alloc.h>
main()
{
    int *a[3], i, j, cols;
    printf("Enter number of columns : ");
    scanf("%d", &cols);
    /*Initialize each pointer in array by address of dynamically allocated
    memory*/
    for(i=0; i<3; i++)
        a[i]=(int *)malloc(cols*sizeof(int));
    for(i=0; i<3; i++)
        for(j=0; j<cols; j++)
    {
        printf("Enter value for a[%d][%d] : ", i, j);
        scanf("%d", &a[i][j]);
    }
    printf("The matrix is :\n");
    for(i=0; i<3; i++)
    {
        for(j=0; j<cols; j++)
            printf("%5d", a[i][j]);
        printf("\n");
    }
    for(i=0; i<3; i++)
        free(a[i]);
}

```

Suppose the number of columns entered is 4. This figure shows how to dynamically allocate a 2-D array using array of pointers. In this case the rows may not be allocated consecutively.



If we want to enter both the number of rows and number of columns at run time, then we can dynamically allocate the array of pointers also.

```

/*P8.33 Program to dynamically allocate a 2-D array*/
#include<stdio.h>
#include<alloc.h>
main( )
{

```

```

int **a,i,j,rows,cols;

printf("Enter number of rows and columns : ");
scanf("%d%d",&rows,&cols);

/*Allocate a one dimensional array of int pointers*/
a=(int **)malloc(rows*sizeof(int*));

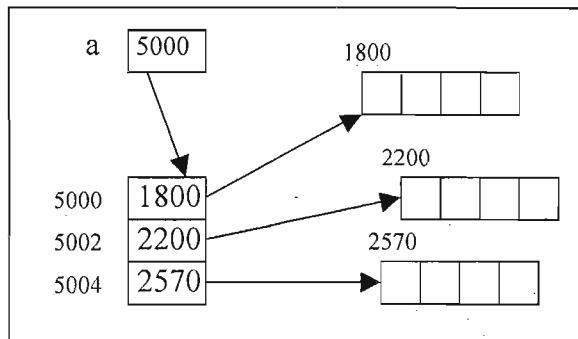
/*Allocate a one dimensional array of integers for each row pointer*/
for(i=0;i<rows;i++)
    a[i]=(int *)malloc(cols*sizeof(int));

for(i=0;i<rows;i++)
    for(j=0;j<cols;j++)
    {
        printf("Enter a[%d] [%d] : ",i,j);
        scanf("%d",&a[i][j]);
    }
printf("The matrix is :\n");
for(i=0;i<rows;i++)
{
    for(j=0;j<cols;j++)
        printf("%5d",a[i][j]);
    printf("\n");
}

for(i=0;i<rows;i++)
    free(a[i]);
free(a);
}

```

Suppose the number of rows entered is 3 and the number of columns entered is 4, then the figure for above program would be like this-



Here the rows are not contiguous, but we can access each element with subscripts. The size of any row can be easily increased or decreased by realloc().

8.22 Pointers To Functions

The code of a function resides in memory hence every function has an address like all other variables

in the program. We can get the address of a function by just writing the function's name without parentheses.

```
/*P8.34 Program to illustrate that every function has an address and how
to access that address*/
#include<stdio.h>
main()
{
    int func1();
    printf("Address of function main() is : %u \n",main);
    printf("Address of function func1() is : %u \n",func1);
    func1(); /*Function call */
}
func1()
{
    printf("India is great\n");
}
```

Output :

Address of function main() is : 657

Address of function func1() is : 691

India is great

8.22.1 Declaring A Pointer To A Function

We have seen that functions have addresses, so we can have pointers that can contain these addresses and hence point to them. The syntax for declaration of a pointer to a function is as-

```
return type (*ptr_name )(type1, type2, ..... );
```

For example-

```
float (*fp)( int );
char (*func_p)(float, char);
```

Here fp is a pointer that can point to any function that returns a float value and accepts an int value as argument. Similarly func_p is a pointer that can point to functions returning char and accepting float and char as arguments.

We can see that this declaration is somewhat similar to the declaration of a function, except that the pointer name is preceded by a * and is enclosed in parentheses. Use of * is obvious since we did this while declaring pointers to variables also, but why is the pointer name enclosed in parentheses. Let us remove the parentheses and see.

```
float *fp(int );
```

How would you declare a function returning a pointer to float and taking an int value? Well exactly in the same manner as above. So in this declaration fp is declared to be a function rather than a pointer and this happened because parentheses have higher precedence than * operator. This is the reason for enclosing the pointer name inside parentheses.

Now we have learnt how to declare a pointer to a function, the next step is to assign a function's address to it.

```
float (*fp)(int , int );           /*Declaring a function pointer */
float func( int , int );          /*Declaring a function*/
```

```
fp = func;           /*Assign address of function func( ) to pointer fp*/
```

After the above assignment fp contains the address of function func(). Declaring a function is necessary before using its address anywhere in the program because without declaration the compiler will not know about this function and will generate an error.

8.22.2 Calling A Function Through Function Pointer

Now let us see how to invoke a function using a function pointer.

```
r = func( a, b);      /*Calling function in usual way*/
r = (*fp)(a, b);      /*Calling function via function pointer */
```

The effect and result of calling a function by its name or by a function pointer is exactly the same.

```
/*P8.35 Program to invoke a function using function pointer*/
#include<stdio.h>
main()
{
    float (*fp)(int, float);
    float add(int, float), result;

    fp=add;      /*Assign address of function add() to pointer fp*/

    /*Invoking a function directly using function's name*/
    result=add(5, 6.6);
    printf("%f\n", result);

    /*Invoking a function indirectly by dereferencing function pointer*/
    result=(*fp)(5, 6.6);
    printf("%f\n", result);
}
float add(int a, float b)
{
    return(a+b);
}
```

Output:

```
11.600000
11.600000
```

8.22.3 Passing a Function's Address as an Argument to Other Function

We can send the function's address as an argument to other function in the same way as we send other arguments. To receive the function's address as an argument, a formal parameter of the appropriate type should be declared. We can then invoke the function sent as an argument by dereferencing the formal pointer variable. The following program will make this point clear.

```
/* P8.36 Program to send a function's address as an argument to other
function */
#include<stdio.h>
main()
{
```

```

void func(char, void (*fp)(float));
void fun1(float);
printf("Function main() called\n");
func('a', fun1);

}

void func(char b, void (*fp)(float))/*Address of fun1 stored in fp*/
{
    printf("Function func() called\n");
    (*fp)(8.5); /*Calling fun1 indirectly using pointer*/
}
void fun1(float f)
{
    printf("Function fun1() called \n");
}

```

Output:

Function main() called
 Function func() called
 Function fun1() called

Here func() is a function which accepts two arguments, a char and a function pointer. This function pointer can point to any function that accepts a float and returns nothing.

fun1() is a function that accepts a float and returns nothing hence we can send its address as second argument to the function func(). The function main() calls function func() while the function func() calls function fun1() indirectly using function pointer. Now we'll write the same program and this time we'll send a pointer that contains the address of function fun1().

```

/* P8.37 Program to pass a pointer containing function's address as an
argument*/
#include<stdio.h>
main()
{
    void func(char, void (*fp)(float));
    void fun1(float);
    void (*p)(float);
    p=fun1;
    printf("Function main() called\n");
    func('a',p);
}
void func(char b, void (*fp)(float)) /*Value of p stored in fp*/
{
    printf("Function func() called\n");
    (*fp)(8.5); /*Calling fun1 indirectly using pointer*/
}
void fun1(float f)
{
    printf("Function fun1() called\n");
}

```

Output:

Function main() called

Function func() called
 Function fun1() called

8.22.4 Using Arrays Of Function Pointers

All this stuff may look a bit confusing and you may think why call a function using function pointer when it can be easily called using its name. Well in many applications we don't know in advance which function will be called. In that case we can take the addresses of different functions in an array and then call the appropriate function depending on some index number.

Let us take a program and understand this concept. In this program we'll add, subtract, multiply or divide two numbers depending on user's choice. For this of course we'll make four different functions. The addresses of these functions will be stored in an array of function pointers.

```
float add(float, int);           /*Declaration of functions */
float sub(float, int);
float mul(float, int);
float div(float, int);

float (*fp[4])( float, int );    /*Declare an array of function pointers*/

fp[0] = add;                    /*Assigning address to elements of the array of function pointer*/
fp[1] = sub;
fp[2] = mul;
fp[3] = div;
```

Instead of the above assignment statements, we could have initialized the array as-

```
float (*fp[])(float , int ) ={ add, sub, mul, div };
```

Now we can see that-

(*fp)[0](a, b);	is equivalent to add(a, b);
(*fp)[1](a, b);	is equivalent to sub(a, b);
(*fp)[2](a, b);	is equivalent to mul(a, b);
(*fp)[3](a, b);	is equivalent to div(a, b);

In the following program, the function select() is used to display menu options and to input user's choice. Depending on the choice of user, corresponding function is called.

```
/*P8.38 Program to understand the use of array of function pointers*/
#include<stdio.h>
int select(void);
float add(float,int);
float sub(float,int);
float mul(float,int);
float div(float,int);
main()
{
    int i,b;
    float a,r;
```

```

float (*fp[ ])(float, int)={add, sub, mul, div};
while(1)
{
    i=select();
    if(i==5)
        exit(1);
    printf("Enter a float and a integer : ");
    scanf("%f %d",&a,&b);
    r=(*fp[i-1])(a,b);
    printf("Result is %f\n",r);
}
int select(void)
{
    int choice;

    printf("1.Add\n");
    printf("2.Subtract\n");
    printf("3.Multiply\n");
    printf("4.Divide\n");
    printf("5.Exit\n");
    printf("Enter your choice : ");
    scanf("%d",&choice);
    return choice;
}

float add(float a,int b)
{
    return a+b;
}

float sub(float a,int b)
{
    return a-b;
}

float mul(float a,int b)
{
    return a*b;
}

float div(float a,int b)
{
    return a/b;
}

```

This program could be written using a switch but writing it using function pointers increases efficiency.
Function pointers are generally used in compilers, interpreters and database programs.

Here is a review of the pointer declarations used in this chapter-

int *p;	/*Pointer to int */
int **p;	/*Pointer to pointer to int */
int *p[20];	/* Array of 20 int pointers*/
int (*p)[20];	/* Pointer to an array of 20 integers */
int *f(void);	/*Function that returns an int pointer*/

```

int (*fp)(void);           /*Pointer to a function, function returns int */
int (*fp[4])(void);       /*An array of 4 pointers to functions, each function returns
                           int */

int *(*fp)(void);          /*Pointer to a function, function returns an int pointer */
float *(*fp)(int, float);  /*Pointer to a function, function takes two arguments of int
                           and float type and returns a float pointer*/
float *(*fp[4])(int, float); /*An array of 4 pointers to functions, each function takes two
                           arguments of int and float type and returns a float pointer*/

```

Exercise

Assume stdio.h is included in all programs.

- (1) main()


```

      {
          int a=5, *ptr;
          ptr=&a;
          printf("Input a number : ");
          scanf("%d",ptr);           /*Suppose the input number is 16*/
          printf("%d %d\n",a,*ptr);
      }
  
```
- (2) main()


```

      {
          int *ptr;
          printf("Enter a number : ");
          scanf("%d",ptr);
          printf("%d\n",*ptr);
      }
  
```
- (3) main()


```

      {
          int arr[5],i;
          for(i=0;i<5;i++)
              printf("%u ",arr+i);    /*Suppose base address of arr is 5000*/
          printf("\nEnter 5 numbers\n");
          for(i=0;i<5;i++)
              scanf("%d",arr+i);
          for(i=0;i<5;i++)
              printf("%d ",*(arr+i));
      }
  
```
- (4) main()


```

      {
          int i,arr[5]={25,30,35,40,45},*p;
          p=&arr;
          for(i=0;i<5;i++)
              printf("%d\t%d\t",*(p+i),p[i]);
      }
  
```
- (5) main()

```
'h
ns
nt
/o
*/  

{  

    int i, arr[5]={25,30,35,40,45}, *p;  

    p=&arr[4];  

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

        printf("%d\t%d\t", *(p-i), p[-i]);  

}  

(6) main()  

{  

    int i, arr[5]={25,30,35,40,55}, *p;  

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

    {  

        printf("%d ", *arr);  

        arr++;  

    }
}  

(7) main()  

{  

    int i, arr[5]={25,30,35,40,45}, *p=arr;  

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

    {  

        (*p)++;  

        printf("%d ", *p);  

        p++;
    }
}  

(8) main()  

{  

    int i, arr[5]={25,40,55,70,85}, *p=arr;  

    for(i=0;i<5;i++)
        printf("%d ", *p++);
    printf("\n");
  

    for(i=0;i<5;i++)
        printf("%d ", *--p);
    printf("\n");
}
}  

(9) main()  

{  

    int i, arr[5]={25,40,55,70,85}, *p=arr;  

    for(i=0;i<8;i++)
        printf("%d ", ++*p);
    printf("\n");
    for(i=0;i<7;i++)
        printf("%d ", (*p)++);
    printf("\n");
}
```

```

(10)main()
{
    int arr[10]={25,30,35,40,55,60,65,70,85,90},*p;
    for(p=&arr[0];p<arr+10;p++)
        printf("%d ",*p);
}

(11)main()
{
    int arr[10]={25,30,35,40,55,60,65,70,85,90},*p;
    for(p=arr+2;p<arr+8;p=p+2)
        printf("%d ",*p);
}

(12)main()
{
    int i,arr[10]={25,30,35,40,55,60,65,70,85,90};
    int *p=arr+9;
    for(i=0;i<10;i++)
        printf("%d ",*p-_);
}

(13)main()
{
    int arr[10]={25,30,35,40,55,60,65,70,85,90},*p;
    for(p=arr+9;p>=arr;p--)
        printf("%d ",*p );
}

(14)main()
{
    int arr[4]={10,20,30,40};/*Assume base address of arr is 5000*/
    int x=100,*ptr=arr;
    printf("%u%d      %d\n",ptr,*ptr,x);
    x=*ptr++;
    printf("%u%d      %d\n",ptr,*ptr,x);
    x=*&ptr;
    printf("%u%d      %d\n",ptr,*ptr,x);
    x=++*ptr;
    printf("%u%d      %d\n",ptr,*ptr,x);
    x=(*ptr)++;
    printf("%u%d      %d\n",ptr,*ptr,x);
}

(15)main( )
{
    int x, arr[8]={11, 22, 33, 44, 55, 66, 77, 88};
    x = (arr+2)[3];
    printf("%d\n", x);
}

```

```
(16)main( )
{
    int arr[8]={11,22,33,44,55,66,77,88};
    int *p,*q;
    q=arr/2;
    p=q*2;
    printf(" %d %d",*p,*q);
}

(17)main( )
{
    int arr[6]={1,2,3,4,5,6};
    int *p=arr;
    printf("Size of p = %u, Size of arr = %u\n",sizeof(p),sizeof(arr));
}

(18)main( )
{
    float a=5,*p,**pp;
    p=&a; /*Assume address of a is 5000*/
    pp=&p; /*Assume address of p is 5520*/
    printf("a = %f, p= %u, pp = %u\n",a,p,pp);
    a=a+1;
    p=p+1;
    pp=pp+1;
    printf("a = %f, p= %u, pp = %u\n",a,p,pp);
}

(19)int a=5,b=10;
main()
{
    int x=20,*ptr=&x;
    printf("%d ",*ptr);
    change1(ptr);
    printf("%d ",*ptr);
    change2(&ptr);
    printf("%d\n",*ptr);
}
change1(int *p)
{
    p=&a;
}
change2(int **pp)
{
    *pp=&b;
}

(20)main( )
{
    int a=2,b=6;
```

```

func(a,&b);
printf("a = %d, b = %d\n",a,b);
}
func(int x,int *y)
{
    int temp;
    temp=x;
    x=*y;
    *y=temp;
}

(21)int *ptr;
main()
{
    func();
    printf("%d\n",*ptr);
}
func()
{
    int num=10;
    ptr=&num;
}

(22)main()
{
    int a=5,b=8;
    func(a,b);
    printf("a = %d , b = %d\n",a,b);
}
func(int x, int y)
{
    int temp;
    temp=*(&x),*(&x)=*(&y),*(&y)=temp;
}

(23)main()
{
    int arr[5]={1,2,3,4,5};      /*Assume base address of arr is 2000*/
    int *p=&arr;
    printf("p = %u,\t",p);
    func1( p );
    printf("p = %u,\t",p);
    func2(&p);
    printf("p = %u\n",p);
}
void func1(int *ptr)
{
    ptr++;
}
void func2(int **pptr )
{

```

```
(*pptr)++;  
}  
  
(24)main()  
{  
    int arr[10];  
    func(arr);  
}  
func(int a[10])  
{  
    int b[10];  
    int x=5,y=4;  
    a=&x;  
    b=&y;  
}  
  
(25)main()  
{  
    int arr[3][4]; /*Assume base address of arr is 5000*/  
    printf("%u\t",arr);  
    printf("%u\t",arr[0]);  
    printf("%u\n",&arr[0][0]);  
    printf("%d\t",sizeof(arr));  
    printf("%d\t",sizeof(arr[0]));  
    printf("%d\n",sizeof(arr[0][0]));  
}  
  
(26)main()  
{  
    int arr[3][4][5]; /*Assume base address of arr is 2000*/  
    printf("%u\t",arr);  
    printf("%u\t",arr[0]);  
    printf("%u\t",arr[0][0]);  
    printf("%u\n",&arr[0][0][0]);  
    printf("%d\t",sizeof(arr));  
    printf("%d\t",sizeof(arr[0]));  
    printf("%d\t",sizeof(arr[0][0]));  
    printf("%d\n",sizeof(arr[0][0][0]));  
}  
  
(27)main()  
{  
    int arr[10] = {1,2,3,4,5,6,7,8,9,10};  
    func(arr+3);  
}  
func(int a[  ])  
{  
    int i;  
    for(i=0;a[i]!=8;i++)  
        printf("%d ",a[i]);  
}
```

```

(28)main()
{
    int i,j;
    int arr[10]={3,2,4,1,5,9,8,10,7,6};

    for(i=0;i<10;i++)
        for(j=0;j<10-i-1;j++)
            if(*(arr+j)>*(arr+j+1))
                swap(arr+j,arr+j+1);
    for(i=0;i<10;i++)
        printf("%d\t",arr[i]);
    printf("\n");
}

swap(int *b,int *c)
{
    int temp;
    temp=*b,*b=*c,*c=temp;
}

(29)main()
{
    int i,arr[3][4] = {{10,11,12,13},{20,21,22,23},{30,31,32,33}};
    int *pa[3];
    int (*p)[4];
    p=arr;
    for(i=0;i<3;i++)
        pa[i] = arr[i];
    printf("%d %d %d\n",pa[0][0],pa[0][1],pa[2][3]);
    printf("%d %d %d\n",p[0][0],p[0][1],p[2][3]);
}

(30)int *func1(void);
int *func2(void);
main()
{
    int *ptr1,*ptr2;
    ptr1=func1();
    ptr2=func2();
    printf("%d %d\n",*ptr1,*ptr2);
}
int *func1(void)
{
    int a=8, *p=&a;
    return p;
}
int *func2(void)
{
    int *p;
    p=(int *)malloc(sizeof(int));
    *p=9;
    return p;
}

```

```

    }

(31)main( )
{
    int i, arr[3][4] = {{10,11,12,13},{20,21,22,23},{30,31,32,33}};
    int *p=arr;
    for(i=0;i<12;i++)
        printf("%d ",p[i]);
    printf("\n");
}

```

Answers

- (1) 16 16
- (2) ptr is not initialized, it contains garbage value and may be pointing anywhere in memory, ptr should be initialized before being used.
- (3) 5000 5002 5004 5006 5008
 Enter 5 numbers : 1 2 3 4 5
 1 2 3 4 5
- The first for loop prints the addresses of the array elements, second for loop inputs the numbers in the array and the third for loop prints the array elements.
- (4) 25 25 30 30 35 35 40 40 45 45
- (5) 45 45 40 40 35 35 30 30 25 25
- (6) Error, since arr is a constant pointer and it can't be changed.
- (7) 26 31 36 41 46
 By (*p)++ we are incrementing the value pointed to by p, and by p++ we are incrementing the pointer.
- (8) 25 40 55 70 85
 85 70 55 40 25
- (9) 26 27 28 29 30 31 32 33
 33 34 35 36 37 38 39
- (10) 25 30 35 40 55 60 65 70 85 90
- (11) 35 55 65
- (12) 90 85 70 65 60 55 40 35 30 25
- (13) 90 85 70 65 60 55 40 35 30 25
- (14) 5000 10 100
 5002 20 10
 5004 30 30
 5004 31 31
 5004 32 31
- (15) 66
 (arr+2)[3] will be interpreted as *(arr+2+3) or as *(arr+5), which is same as arr[5].
- (16) Error, since multiplication and division operations are not valid with pointers.
- (17) Size of p = 2, Size of arr = 12

- (18) a = 5.000000, p= 5000, pp = 5520
 a = 6.000000, p= 5004, pp = 5522

(19) 20 20 10

Since we need to change the value of ptr we have to send its address, and in the function we have to receive address of a pointer so we need a pointer to pointer, hence the function change2() will be able to change the value of ptr and change1() can't change the value of ptr .

(20) a = 2, b=2

Since the address of b is passed so its value changes, while the value of a does not change because its value is passed.

(21) In this program we are assigning the address of a local variable num. When the function func() terminates, the variable num expires and the memory used by it may be used for some other purpose. This program may give 10 as the output, but the memory may be overwritten anytime.

(22) a = 5 b = 8

We are passing the values of variables a and b, so their values will not change. Writing *(&x) is same as writing x.

(23) p = 2000, p = 2000, p = 2002

The reasoning is same as in question 19.

(24) Error, a = &x is OK since a is declared as a pointer variable, but b = &y not Ok because b is name of an array, hence a constant pointer.

(25) 5000 5000 5000

 24 8 2

(26) 2000 2000 2000 2000

 120 40 10 2

(27) 4 5 6 7

The address of arr[3] is passed to the function. Inside the function, a is declared as a pointer variable and it gets initialized with the address of arr[3].

(28) 1 2 3 4 5 6 7 8 9 10

The numbers are sorted through bubble sort

(29) 10 11 33

 10 11 33

pa is an array of 3 pointers, each of base type int, while p is a pointer to an array of 4 integers.

(30) -20 9

It is wrong to return the address of a local variable, but the address of dynamically allocated memory can be returned.

(31) 10 11 12 13 20 21 22 23 30 31 32 33

Chapter 9

Strings

There is no separate data type for strings in C. They are treated as arrays of type char. A character array is a string if it ends with a null character ('0'). This null character is an escape sequence with ASCII value 0. Strings are generally used to store and manipulate data in text form like words or sentences.

9.1 String Constant or String Literal

A string constant is a sequence of characters enclosed in double quotes. It is sometimes called a literal. The double quotes are not a part of the string. Some examples of string constants are-

"V"

"Taj Mahal"

"2345"

"Subhash Chandra Bose was a great leader"

"" (Null string, contains only '0')

"My age is %d and height is %f\n" (Control string used in printf)

Whenever a string constant is written anywhere in a program, it is stored somewhere in memory as an array of characters terminated by a null character('0'). The string constant itself becomes a pointer to the first character in the array. For example the string "Taj Mahal" will be stored in memory as-

1000	1001	1002	1003	1004	1005	1006	1007	1008	1009
T	a	j		M	a	h	a	l	\0

Each character occupies one byte and compiler automatically inserts the null character at the end. The string constant "Taj Mahal" is actually a pointer to the character 'T'. So whenever a string constant is used in the program it is replaced by a pointer pointing to the string.

If we have a pointer variable of type char *, then we can assign the address of this string constant to it as-

```
char *p = "Taj mahal";
```

Similarly when we write-

```
printf("Subhash Chandra Bose\n");
```

then actually a pointer to character(char *) is passed to the printf() function.

If identical string constants are used in a program, they will be stored separately at different memory

LOCATIONS. FOR THE IT A STRING CONSTANT "India" appears three times in a program then it will be stored twice in memory. Consider this program

```
#include<stdio.h>
main()
{
    printf("%u\n", "good");
    printf("%u\n", "good");
    if("bad" == "bad")
        printf("Same\n");
    else
        printf("Not same\n");
}
```

Output

174

183

Not same

From the above output we can see that the two strings "good" are identical but they are stored at different places. When we compare two identical strings using equality operator then we are actually comparing the addresses and not the strings. (Later we'll study a library function strcmp() used to compare strings).

Can you guess what will the expression "software"[4] represent? According to the equivalence of pointer and subscript notations, this expression is equivalent to *("software"+4). We know that "software" represents the address of first character i.e. 's'. On adding 4 to this address we get the address of character 'w', and on dereferencing it we get the character 'w'.

In some compilers we can change the string constant by storing its address in a pointer, but it is not a good practice to do so and the behaviour of such a program is undefined. So we should never attempt to alter the string constants.

We have studied that a string constant gives the address of first character in it, but there is an exception to this rule; when the string constant is used as an initializer for a character array then it does not represent any address and it is not stored anywhere in memory. For example-

```
char arr[5] = "Deep";
```

Here the string constant "Deep" is not stored in memory and hence does not represent any address.

Note that the 'b' and "b" are different. 'b' is a character constant which represents the ASCII value of character 'b' while "b" is a string constant which consists of character 'b' and null character '\0'.

Inside a string constant, the backslash is considered as an escape character, so if there is a need to include backslash character within a string constant then it should be preceded by another backslash. If we want to include double quotes inside string constant, then it should also be preceded by a backslash. For example consider these two printf statements-

```
printf("good\b\bad");
printf("I love \"C\" programming");
```

The output of these 2 statements would be-

good\bad

I love "C" programming

The length of a string is not limited to a line only, it can be continued by adding a backslash at the end of the line.

If the string constants are placed adjacent to each other then the compiler concatenates them and places a single null character at the end of the concatenated string. For example these two string constants will be concatenated by the compiler as-

"Red " "Fort" → "Red Fort"

9.2 String Variables

To create a string variable we need to declare a character array with sufficient size to hold all the characters of the string including null character.

```
char str[ ] = { 'N', 'e', 'w', ' ', 'Y', 'o', 'r', 'k', '\0' };
```

We may also initialize it as-

```
char str[ ] = "New York";
```

This initialization is same as the previous one and in this case the compiler automatically inserts the null character at the end. Note that here the string constant does not represent an address. The array str will be stored in memory as-

1000	1001	1002	1003	1004	1005	1006	1007	1008
N	e	w		Y	o	r	k	\0

str[0] str[1] str[2] str[3] str[4] str[5] str[6] str[7] str[8]

Here we have not specified the size of the array, but if we specify it then we should take care that array should be large enough to hold all the characters including the null character.

```
/*P9.2 Program to print characters of a string and address of each
character. */
#include<stdio.h>
main()
{
    char str[ ]="India";
    int i;
    for(i=0;str[i]!='\0';i++)
    {
        printf("Character = %c\t",str[i]);
        printf("Address = %u\n",&str[i]);
    }
}
```

Output

Character = I	Address = 1000
Character = n	Address = 1001
Character = d	Address = 1002
Character = i	Address = 1003

Character = a Address = 1004

	arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	arr[5]
I	n	d	i	a	\0	
1000	1001	1002	1003	1004	1005	

```
/*P9.3 Program to print the address and characters of the string using
pointer*/
#include<stdio.h>
main()
{
    char str[]="India";
    char *p;
    p=str;
    while(*p!='\0')
    {
        printf("Character = %c\t",*p);
        printf("Address=%u\n",p);
        p++;
    }
}
```

The output of this program is same as that of previous program.

Here 'p' is pointer variable which holds the base address of array str[]. Incrementing this pointer by 1 gives the address of next element of character array str[], so on incrementing and dereferencing this pointer we can print all the elements of the string. This procedure is similar to that applied in other arrays, except that here the loop terminates when the character '\0' is encountered which signifies the end of string.

There is a shortcut way for entering and printing strings, using %s specification in the control string of printf() and scanf().

```
/*P9.4 Program to input and output a string variable using scanf( )*/
#include<stdio.h>
main()
{
    char name[20];
    printf("Enter name :");
    scanf("%s",name);
    printf(" %s ",name);
    printf(" %s\n","Srivastava");
}
```

1st run:

Enter a name: Deepali

Deepali Srivastav

2nd run:

Enter a name : Suresh Kumar

Suresh Srivastava

The printf() takes the base address of string and continues to display the characters until it encounters the character '\0'.

When we enter the string using %s, the null character is automatically stored at the end of array. We haven't used & sign in the scanf() since the name of the array is itself address of the array. In the 2nd run when we entered a string with space we could not get the required result. This is because scanf() stops reading as soon as it encounters a whitespace. So for entering strings with whitespaces we can use the function gets(). It stops reading only when it encounters a newline and replaces this newline by the null character. We have another function puts() which can output a string and replaces the null character by a newline.

```
/*P9.5 Program to understand the use of gets() and puts()*/
#include<stdio.h>
main()
{
    char name[20];
    printf("Enter name : ");
    gets(name);
    printf("Entered name is : ");
    puts(name);
}
```

Output:

```
Enter name : Suresh Kumar
Entered name is : Suresh Kumar
```

9.3 String Library Functions

There are several library functions used to manipulate strings. The prototypes for these functions are in header file string.h. We'll discuss some of them below-

9.3.1 strlen()

This function returns the length of the string i.e. the number of characters in the string excluding the terminating null character. It accepts a single argument, which is pointer to the first character of the string. For example strlen("suresh") returns the value 6. Similarly if s1 is an array that contains the name "deepali" then strlen(s1) returns the value 7.

```
/*P9.6 Program to understand the work of strlen() function*/
#include<stdio.h>
#include<string.h>
main()
{
    char str[20];
    int length;
    printf("Enter the string:\n");
    scanf("%s",str);
    length=strlen(str);
    printf("Length of the string is : %d\n",length);
```

}

Output:

```
Enter the string : CinDepth
Length of the string is : 8
```

Creation Of This Function

Array version-

```
int astrlen(char str[])
{
    int i=0;
    while(str[i]!='\0')
        i++;
    return i;
}
```

Pointer version-

```
int pstrlen(char *str)
{
    char *start=str;
    while(*str!='\0')
        str++;
    return (str-start);
}
```

We can also write the while condition as-

```
while(*str)
```

because the ASCII value of '\0' is 0 which is considered false in C.

9.3.2 strcmp()

This function is used for comparison of two strings. If the two strings match, strcmp() returns value 0, otherwise it returns a non-zero value. This function compares the strings character by character. The comparison stops when either the end of string is reached or the corresponding characters in the two strings are not same. The non-zero value returned on mismatch is the difference of the ASCII values of the non-matching characters of the two strings.

strcmp(s1, s2) returns a value-

- < 0 when s1 < s2
- = 0 when s1 == s2
- > 0 when s1 > s2

Generally we don't use the exact non-zero value returned in case of mismatch. We only need to know its sign to compare the alphabetical positions of the two strings. We can use this function to sort strings alphabetically.

```
/*P9.7 Program to understand the work of strcmp() function*/
#include<stdio.h>
#include<string.h>
main()
{
```

```
char str1[10],str2[10];
printf("Enter the first string : ");
scanf("%s",str1);
printf("Enter the second string : ");
scanf("%s",str2);
if((strcmp(str1,str2))==0)
    printf("Strings are same\n");
else
    printf("Strings are not same\n");
```

Output:

```
Enter the first string : Bangalore
Enter the second string : Mangalore
Strings are not same
Creation of this function
```

Array version-

```
int strcmp(char str1[],char str2[])
{
    int i=0;
    while(str1[i]!='\0'&&str2[i]!='\0'&&str1[i]==str2[i])
        i++;

    if(str1[i]==str2[i])
        return 0;
    else
        return(str1[i]-str2[i]);
}
```

Pointer version-

```
int pstrcmp(char *str1,char *str2)
{
    while(*str1!='\0'&&*str2!='\0'&&*str1==*str2)
    {
        str1++;
        str2++;
    }
    if(*str1==*str2)
        return 0;
    else
        return(*str1-*str2);
}
```

9.3.3 strcpy()

This function is used for copying one string to another string. strcpy(str1, str2) copies str2 to str1. Here str2 is the source string and str1 is destination string. If str2 = "suresh" then this function copies "suresh" into str1. This function takes pointers to two strings as arguments and returns the pointer to first string.

```
/* P9.8 Program to understand the work of strcpy() function*/
#include<stdio.h>
#include<string.h>
main()
{
    char str1[10],str2[10];
    printf("Enter the first string : ");
    scanf("%s",str1);
    printf("Enter the second string : ");
    scanf("%s",str2);
    strcpy(str1,str2);
    printf("First string :%s \t\t Second string : %s\n",str1,str2);
    strcpy(str1,"Delhi");
    strcpy(str2,"Calcutta");
    printf("First string :%s \t\t Second string : %s\n",str1,str2);
}
```

Output:

```
Enter the first string : Bombay
Enter the second string : Mumbai
First string : Mumbai           Second string : Mumbai
First string : Delhi            Second string : Calcutta
```

The programmer should take care that the first string has enough space to hold the second string. The function calls like strcpy("New", str1) or strcpy("New", "York") are invalid because "New" is a string constant which is stored in read only memory and so we can't overwrite it.

```
strcpy("New", str1);      /*Invalid*/
strcpy("New", "York");   /*Invalid*/
```

Creation of this Function

Array version-

```
char *astrcpy(char str1[],char str2[])
{
    int i=0;
    while(str2[i]!='\0')
    {
        str1[i]=str2[i];      /*Copy character by character*/
        i++;
    }
    str1[i]='\0';
    return str1;
}
```

Pointer version-

```
char *pstrcpy(char *str1,char *str2)
{
    while(*str2!='\0')
    {
        *str1=*str2;
```

```

        str1++;
        str2++;
    }
    *str1 = '\0';
    return str1;
}

```

We can write the above function concisely as-

```

char *pstrcpy(char *str1, char *str2)

    while (*str2++ = *str1++);
    return str1;
}

```

9.3.4 strcat()

This function is used for concatenation of two strings. If first string is "King" and second string is "size" then after using this function the first string becomes "Kingsize".

```
strcat(str1, str2); /*concatenates str2 at the end of str1*/
```

The null character from the first string is removed, and the second string is added at the end of first string. The second string remains unaffected.

This function takes pointer to two strings as arguments and returns a pointer to the first(concatenated) string.

```

ig. *P9.9 Program to understand the work of strcat() function.*/
sa #include<stdio.h>
main()
{
    char str1[20], str2[20];
    printf("Enter the first string : ");
    scanf("%s", str1);
    printf("Enter the second string : ");
    scanf("%s", str2);
    strcat(str1, str2);
    printf("First string : %s \tSecond string : %s\n", str1, str2);
    strcat(str1, "_one");
    printf("Now first string is : %s \n", str1);
}

```

Output:

Enter the first string : data

Enter the second string : base

First string : database Second string : base

Now first string is : database_one

Creation of this function

Array version-

```

char *astrcat(char str1[], char str2[])
{
    int i=0, j=0;

```

```

        while(str1[i]!='\0')           /*Check for the end of first string*/
            i++;
        while(str2[j]!='\0') /*Add second string at the end of first*/
        {
            str1[i]=str2[j];
            i++;
            j++;
        }
        str1[i]='\0';
        return str1;
    }
}

```

Pointer version-

```

char *pstrcat(char *str1,char *str2)
{
    while(*str1!='\0')
        str1++;
    while(*str2!='\0')
    {
        *str1=*str2;
        str1++;
        str2++;
    }
    *str1='\0';
    return str1;
}

```

The function strcat() returns a pointer to the first string, hence it can be nested. The following program illustrates this-

```

/*P9.10 Program to understand the work of strcat() function.*/
#include<stdio.h>
#include<string.h>
main()
{
    char str1[20]="Subhash ";
    char str2[10]="Chandra ";
    strcat(strcat(str1,str2),"Bose");
    printf("str1 - %s\n",str1);
}

```

Output:

str1 - Subhash Chandra Bose

Other string related functions are explained in the chapter on library functions.

9.4 String Pointers

We can take a char pointer and initialize it with a string constant. For example-

```
char *ptr = "Chennai";
```

Here ptr is a char pointer which points to the first character of the string constant “Chennai” i.e. contains the base address of this string constant.

Now we'll compare the strings defined as arrays and strings defined as pointers.

```
char str[] = "Mumbai";
```

```
char *ptr = "Chennai";
```

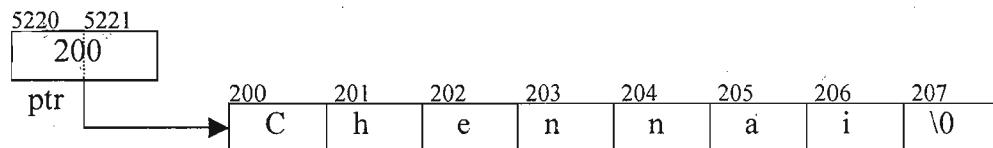
These two forms may look similar but there are some differences in them. The initialization itself has different meaning in both forms. In the array form, initialization is a short form for-

```
char str[] = {‘M’, ‘u’, ‘m’, ‘b’, ‘a’, ‘i’, ‘\0’};
```

while in pointer form, address of string constant is assigned to the pointer variable.

Now let us see how they are represented in memory.

1000	1001	1002	1003	1004	1005	1006
M	u	m	b	a	i	\0
str[0]	str[1]	str[2]	str[3]	str[4]	str[5]	str[6]



Here str is an array of characters and 7 bytes are reserved for it. Since str is the name of an array hence it is a constant pointer which will always point to the first element of array. The elements of array are initialized with the characters of the string. Note that we had mentioned before that when a string constant appears as array initializer, then it does not return an address.

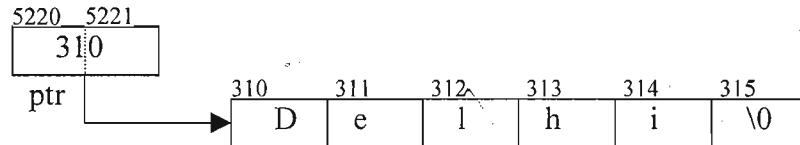
In the second case, the string constant "Chennai" is stored somewhere in memory with 8 consecutive bytes reserved for it. The string constant returns the address of the first character of the string that is assigned to the pointer variable ptr. So in this case total 10 bytes are reserved, 2 bytes for the pointer variable and 8 bytes for the string.

The main difference is that str is a constant pointer and will always contain address 1000 while ptr is a pointer variable and may contain any other address. So string assignments are valid for pointers while they are invalid for strings defined as arrays.

```
str = "Bombay"; /*Invalid*/
```

```
ptr = "Delhi"; /*Valid*/
```

We can assign string of any length to ptr. That string constant will be stored somewhere and its address will be assigned to ptr.



We can assign a different string to str by scanf(), strcpy or by assigning characters.

```
strcpy(str, "Bombay");
```

```
scanf("%s", str);
```

or

```
str[0] = 'B'; str[1] = 'o'; str[2] = 'm'; str[3] = 'b'; str[4] = 'a'; str[5] = 'y'; str[6] = '\0';
This will write the new string at the address 1000, but take care that string size does not exceed the size of array.
```

We have studied earlier that string constants are stored in read only area by some compilers and so they can't be changed. So these operations are invalid-

```
char *ptr = "Bareilly";
ptr[0] = 'D';           /*Invalid*/
scanf("%d", ptr);      /*Invalid*/
strcpy(ptr, "Bareilly"); /*Invalid*/
```

We can't use scanf() with uninitialized pointer since it contain garbage value-

```
char *ptr;
scanf("%d", ptr);      /*Invalid*/
```

For this first we should allocate memory through malloc(), and let this pointer point to that region of memory.

```
ptr = (char *)malloc(20);
scanf("%s", ptr); /*Valid*/
```

```
/*P9.11*/
#include<stdio.h>
main()
{
    char *str;
    str=(char *)malloc(10);
    printf("Enter a string : ");
    scanf("%s",str);
    printf("String is : %s\n",str);
}
```

Output:

```
Enter a string : Oxford
String is : Oxford
```

9.5 Array Of Strings Or Two Dimensional Array Of Characters

Strings are character arrays so array of strings means array of character type arrays i.e. a two dimensional array of characters.

Suppose we declare and initialize a two-dimensional array of characters as-

```
char arr[5][10] = {
    "white",
    "red",
    "green",
```

```

        "yellow",
        "blue"
    };
}

This initialization is equivalent to
char arr[5][10] = {
    {'w','h','i','t','e','\0'},
    {'r','e','d','\0'},
    {'g','r','e','e','n','\0'},
    {'y','e','l','l','o','w','\0'},
    {'b','l','u','e','\0'}
};

```

Here first subscript of array denotes number of strings in the array and second subscript denotes the maximum length that each string can have. The space reserved for this two-d array is 50 bytes.

Here the name of 2-D array is arr and it gives the base address of the array i.e. it gives the address of first string.

arr[0]	represents 0 th string , points to 0 th character of 0 th string
arr[1]	represents 1 st string , points to 0 th character of 1 st string
arr[i]	represents i th string , points to 0 th character of i th string
arr[i][j]	represents j th character in i th string

So we see that if we want to access individual characters in the string then we use two subscripts and if we want to access the strings we use a single subscript.

2000	w	h	i	t	e	\0					2009
2010	r	e	d	\0							2019
2020	g	r	e	e	n	\0					2029
2030	y	e	l	l	o	w	\0				2039
2040	b	l	u	e	\0						2049

This is the internal storage representation of array of strings. 2000 is the base address of the first string. Similarly, 2010 is the base address of the second string. Here 10 bytes are reserved in memory for each string. We can see that first string takes only 6 bytes, so 4 bytes are wasted. Similarly 2nd string takes only 4 bytes and 6 bytes are wasted. The total number of bytes occupied by the array is 50 while the strings use only 28 bytes so 22 bytes of memory is wasted. Now we'll see through a program how to print these strings.

```

/*P9.12 Program to print the strings of the two-dimensional character
array*/
#include<stdio.h>
#define N 5
#define LEN 10
main()
{
    char arr[N][LEN]={
        "white",

```

```
        "red",
        "green",
        "yellow",
        "blue"
    };
int i;
for(i=0;i<N;i++)
{
    printf ("String = %s\t",arr[i] );
    printf ("Address of string = %u\n", arr[ i]);
}
```

Output:

String = white	Address of string = 2000
String = red	Address of string = 2010
String = green	Address of string = 2020
String = yellow	Address of string = 2030
String = blue	Address of string = 2040

In the above program we have initialized the 2-D array with strings. Suppose we don't initialize it at the time of declaration and try to assign strings to it afterwards like this-

```
arr[0] = "white"; /*Invalid*/  
arr[1] = "red"; /*Invalid*/
```

This will give an error ‘lvalue required’. Similarly we cannot write

```
arr[0] = arr[1]; /*Invalid*/
```

If we want to assign values to strings we'll have to use `strcpy()` or `scanf()` function.

```
strcpy( arr[0], "white");           /*Valid*/  
scanf( "%s", arr[0]);             /*Valid*/  
strcpy(arr[0], arr[1]);           /*Valid*/
```

Now we'll take a program to sort the strings alphabetically using the selection sort technique.

```
/*P9.13 Program to sort the array of strings*/
#include<stdio.h>
#define N 5
#define LEN 10
main()
{
    char arr[N][LEN]={  
        "white",  
        "red",  
        "green",  
        "yellow",  
        "blue"  
    };  
    char temp[10];  
    int i,j;
```

```

printf("Before sorting :\n");
for(i=0;i<N;i++)
    printf ("%s      ", arr[i]);
printf("\n");
for(i=0;i<N;i++)
for(j=i+1;j<N;j++)
    if(strcmp(arr[i],arr[j])>0)
    {
        strcpy(temp,arr[i]);
        strcpy(arr[i],arr[j]);
        strcpy(arr[j],temp) ;
    }
printf("After sorting :\n");
for(i=0;i<N;i++)
    printf ("%s      ", arr[i]);
}

```

Output:

Before sorting :
white red green yellow blue

After sorting :
blue green red white yellow

The internal representation of strings after sorting is-

2000	b	l	u	e	\0					2009
2010	g	r	e	e	n	\0				2019
2020	r	e	d	\0						2029
2030	w	h	i	t	e	\0				2039
2040	y	e	l	l	o	w	\0			2049

Internal representation after sorting

Here the sorting is not considered very efficient since each time the whole string is being copied.

9.6 Array Of Pointers To Strings

We have already studied about array of pointers. Array of pointers to strings is an array of char pointers in which each pointer points to the first character of a string i.e. each element of this array contains the base address of a string. Let us take an example and see how this array can be declared and initialized.

```

char *arrp[ ]={

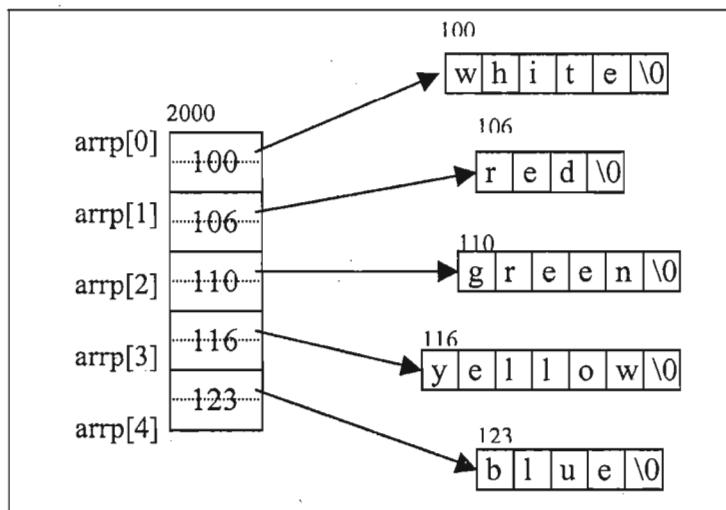
                "white",
                "red",
                "green",
                "yellow",
                "blue"
}

```

Here arrp is an array of pointers to strings. We have not specified the size of array, so the size is determined

by the number of initializers. The initializers are string constants. arrp[0] contains the base address of string "white", similarly arrp[1] contains the base address of string "red".

Now let us see how these strings are stored in memory-



String : white	Address of string : 100	Address of string is stored at : 2000
String : red	Address of string : 106	Address of string is stored at : 2002
String : green	Address of string : 110	Address of string is stored at : 2004
String : yellow	Address of string : 116	Address of string is stored at : 2006
String : blue	Address of string : 123	Address of string is stored at : 2008

Here all strings occupy 28 bytes and 10 bytes are occupied by the array of pointers. So the total bytes occupied are 38 bytes. In two-dimensional array, total bytes occupied were 50 bytes for these same strings. So we can see here saving of 12 bytes.

In the figure we see that strings are stored in consecutive memory locations. Well it is not necessary they may not be stored consecutively and pointers may point to strings located anywhere. Now let us take a program and see how to print these strings.

```
/*P9.14 Program to print the address and string using array of pointer
to strings*/
#include<stdio.h>
main()
{
    int i;
    char *arrp[ ]={ . . .
                    "white",
                    "red",
                    "green",
                    "yellow",
                    "blue"
    };
}
```

depth

String

```

for(i=0;i<5;i++)
{
    printf ("String : %s\t",arrp[i] );
    printf ("Address of string : %u\t",arrp[i] );
    printf("Address of string is stored at : %u\n",arrp+i );
}

```

Output:

String : white	Address of string : 100	Address of string is stored at : 2000
String : red	Address of string : 106	Address of string is stored at : 2002
String : green	Address of string : 110	Address of string is stored at : 2004
String : yellow	Address of string : 116	Address of string is stored at : 2006
String : blue	Address of string : 123	Address of string is stored at : 2008

Instead of initializing we could have assigned strings after declaration as-

```

arrp[0] = "white";      /*Valid*/
arrp[1] = "red";        /*Valid*/

```

Remember this type of assignment was invalid when we used 2-D array. Here it is valid because arrp[0] is a pointer variable and it can be assigned address of any string.

Now let us compare array of strings and array of pointers to strings. Earlier we had studied the differences between strings declared as pointers and strings declared as arrays. Here also we have same differences that are illustrated by this program.

```

/*P9.15 Program to show the differences between array of strings and
array of pointers to strings*/
#include<stdio.h>
main()
{
    char arr[5][10];
    char *arrp[5];

    arr[0] = "January";           /*Invalid*/
    arrp[0] = "January";          /*Valid*/

    strcpy(arr[1], "February");   /*Valid*/
    strcpy(arrp[1], "February");  /*Invalid, arrp[1] not initialized*/

    scanf("%s", arr[2]);          /*Valid*/
    scanf("%s", arrp[2]);         /*Invalid, arrp[2] not initialized*/

    arrp[3] = (char *)malloc(10);
    strcpy(arrp[3], "March");     /*Valid*/

    arrp[4] = (char *)malloc(10);
    scanf("%s", arrp[4]);         /*Valid*/
}

```

We have seen that we can't input strings directly in an array of pointers using scanf(). We have to

first allocate memory through malloc(). Here is a program in which we first enter the string and then allocate memory required for that string. This allocated memory is pointed by an element of array of pointers.

```
/*P9.16*/
#include<stdio.h>
#include<string.h>
main()
{
    char *arrp[10], str[20];
    int i;
    for(i=0;i<10;i++)
    {
        printf("Enter string %d : ",i+1);
        gets(str);
        /* Now allocate memory sufficient to hold the string*/
        arrp[i]=(char *)malloc(strlen(str)+1);
        strcpy(arrp[i],str);
    }
    for(i=0;i<10;i++)
        printf("%s\t",*(arrp+i));
    printf("\n");
    for(i=0;i<10;i++)
        free(arrp[i]);
}
```

If we want to enter the number of strings at run time, then we can allocate the array of pointers dynamically as we had done in previous chapter program P8.33.

Array of pointers is also useful in grouping together logically related data items. For example we can take the names of months in an array and then access them by appropriate subscript.

```
/*P9.17 Program to input a date and print the month*/
#include<stdio.h>
main()
{
    int d,m,y;
    char *months[]={"January", "February", "March", "April", "May",
                    "June", "July", "August", "September", "October",
                    "November", "December"};
    printf("Enter date (dd/mm/yy) : ");
    scanf("%d/%d/%d",&d,&m,&y);
    printf("Month : %s\n",months[m-1]);
}
```

Output:

Enter date : 2/5/2002

Month : May

We could do this by switch statement also but using array of pointers makes the stuff concise.

The next program sorts the strings represented by array of pointers using selection sort technique

```

/*P9.18 Program to sort strings represented by array of pointers*/
#include<stdio.h>
#define N 5
main()
{
    char *arrp[N]={ "white", "red", "green", "yellow", "blue" };
    int i,j;
    char *temp;
    printf("Before sorting : \n");
    for(i=0;i<N;i++)
        printf("%s\t",arrp[i]);
    printf("\n");
    for(i=0;i<N;i++)
        for(j=i+1;j<N;j++)
            if(strcmp(arrp[i],arrp[j])>0)
            {
                temp=arrp[i];
                arrp[i]=arrp[j];
                arrp[j]=temp;
            }
    printf("After sorting :\n");
    for(i=0;i<N;i++)
        printf("%s\t",arrp[i]);
    printf("\n");
}

```

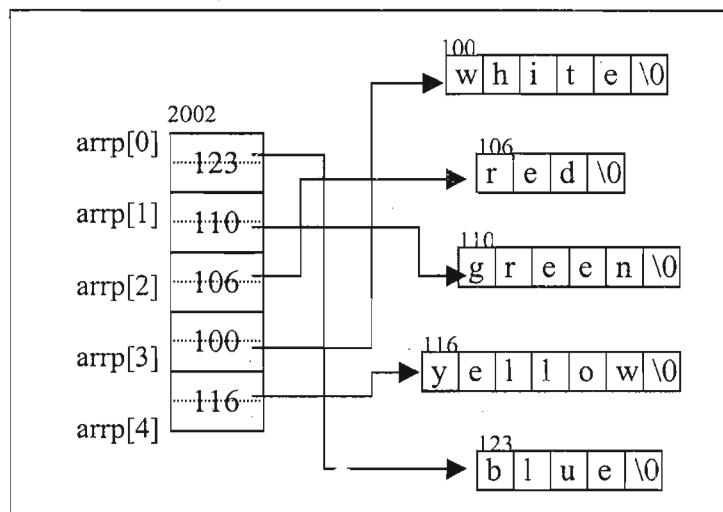
Output:

Before sorting -

white red green yellow blue

After sorting -

blue green red white yellow.



Here sorting is efficient since only pointers are being exchanged.

9.7 sprintf()

Declaration:

```
int sprintf (char *str, const char *controlstring [, argument1, argument2, .....] );
```

This function is same as printf() function except that instead of sending the formatted output to the screen, it stores the formatted output in a string. So with the help of this function we can convert variables of any data type to string. A null character is appended at the end automatically. This function returns the number of characters output to the string excluding the null character. It is the responsibility of the programmer to take the size of string large enough to store all the arguments so that no overflow occurs.

In the following program we convert integer and float values to strings using sprintf.

```
/*P9.19 Program to convert integer and float values to strings using
sprintf() function*/
#include<stdio.h>
main()
{
    char str1[10];
    char str2[10];
    int x=1348;
    float y=234.56;
    sprintf(str1,"%d",x);
    sprintf(str2,"%.2f",y);
    printf("str1 = %s,str2 = %s\n",str1,str2);
}
```

Output:

str1 = 1348, str2 = 234.56

```
/*P9.20 Program to understand the use of sprintf() function*/
#include<stdio.h>
main()
{
    char str[30];
    char name[10] = "Suresh";
    int m1=89,m2=78,m3=80;
    float per=(m1+m2+m3)/3.0;
    char gr='A';
    sprintf(str,"Result - %s %d %d %d %f %c\n",name,m1,m2,m3,per,g
    printf("The string is : \n");
    puts(str);
}
```

Output:

The string is-

Result - Suresh 89 78 80 82.333336 A

9.8 sscanf()

```
int sscanf (const char *str, const char *controlstring [, address1, address2, .....]);
```

This function is same as the scanf() function except that data is read from a string rather than the standard input. We can read the formatted text from a string and convert it into variables of different data types. In the following program, we have two strings that are converted to integer and float values using sscanf().

```
/*P9.21 Program to convert strings to integer and float values*/
#include<stdio.h>
main()
{
    char str1[10] = "1348";
    char str2[10] = "234.56";
    int x;
    float y;
    sscanf(str1, "%d", &x);
    sscanf(str2, "%f", &y);
    printf("Value of x = %d, Value of y = %.2f\n", x, y);
```

Output:

Value of x = 1348, Value of y = 234.56

```
/*P9.22 Program to understand the use of sscanf() function*/
#include<stdio.h>
main()
{
    struct{
        char name[10];
        int age;
        float sal;
    }emp;
    char str[30] = "Anita 23 8000.0";
    sscanf(str, "%s%d%f", &emp.name, &emp.age, &emp.sal);
    printf("Name      : %s\n", emp.name);
    printf("Age       : %d\n", emp.age);
    printf("Salary   : %.2f\n", emp.sal);
```

Output:

Name	:	Anita
Age	:	23
Salary	:	8000.00

sscanf() can be used for validating input data. We know how to put simple validity checks on data such as whether it is in specified range or not, but what if the user starts entering a string where an integer is required. This type of error is difficult to detect and correct. Here we can use sscanf() to check whether a valid integer value is entered or not. For this we take the input in a string instead of an integer variable and then check each character of that string , and if any character in the string

is other than a digit , ‘+’ or ‘-’, then we can print the message that input is not a valid integer. If all the characters input in the string are valid then we can convert that string to an integer variable with the help of `sscanf()` and then use that integer variable in our program.

9.9 Some Additional Problems

Problem 1

Write a program to test whether a word is palindrome or not. A palindrome is a word that remains the same when reversed. For example – radar, madam

```
/*P9.23 Program to test whether a word is palindrome or not*/
#include<stdio.h>
#include<string.h>
main()
{
    char str[10];
    int i=0,j,flag;
    printf("Enter the word : ");
    scanf("%s",str);
    j=strlen(str)-1;
    while(i<=j)
    {
        if(str[i]==str[j])
            flag=1;
        else
        {
            flag=0;
            break;
        }
        i++;
        j--;
    }
    if(flag==1)
        printf("Word is palindrome\n");
    else
        printf("Word is not palindrome\n");
}
```

Problem 2

Write a program to convert a lowercase string into uppercase.

```
/*P9.24*/
#include<stdio.h>
main()
{
    char str[10];
    int i=0;
    printf("Enter a string in lowercase : ");
    scanf("%s",str);
    while(str[i]!='\0')
    {
```

Stringr. If
with

```
    str[i]=str[i]-32;
    i++;
}
printf("The uppercase string is : %s\n",str);
}
```

the Problem 3

Write a program to enter any string and print it in reverse order.

```
/*P9.25 Program to enter any string and print it in reverse order.*/
#include<stdio.h>
main()
{
    char str[20];
    int len;
    printf("Enter any string : ");
    scanf("%s",str);
    len=strlen(str)-1;
    while(len>=0)
    {
        printf("%c",str[len]);
        len--;
    }
    printf("\n");
}
```

Problem 4

Write a program to accept any line and count the number of words in it.

```
/*P9.26*/
#include<stdio.h>
main()
{
    char line[100];
    int count=0,i=0;
    printf("Enter the line of text : ");
    gets(line);
    while(line[i]!='\0')
    {
        if(line[i]==32)
            count++;
        i++;
    }
    if(line[i]=='\0')
        count++;
    printf("The number of words in line = %d\n", count);
}
```

Problem 5

Write a program to input two strings consisting of maximum 80 characters. Examine both the strings

and remove all the common characters from both of these strings. Display the resultant string.

```
/*P9.27*/
#include<stdio.h>
main( )
{
    char str1[80],str2[80],str3[80],str4[80];
    int i,j,k,flag;

    printf("Enter the first string : ");
    scanf("%s",str1);
    printf("Enter the second string : ");
    scanf("%s",str2);
    k=0;
    for(i=0;i<strlen(str1);i++)
    {
        flag=0;
        for(j=0;j<strlen(str2);j++)
        {
            if(str1[i]==str2[j])
            {
                flag=1;
                break;
            }
        }
        if(flag!=1)
        {
            str3[k]=str1[i];
            k++;
        }
    }
    str3[k]='\0';

    k=0;
    for(i=0;i<strlen(str2);i++)
    {
        flag=0;
        for(j=0;j<strlen(str1);j++)
        {
            if(str2[i]==str1[j])
            {
                flag=1;
                break;
            }
        }
        if(flag!=1)
        {
            str4[k]=str2[i];
            k++;
        }
    }
    str4[k]='\0';
```

```
printf("The first string is : %s\n",str3);
printf("The second string is : %s\n",str4);
```

Problem 6

Write a program to read in a string and output the frequency, of each character, in that string.

```
/* P9.28 */
#include<stdio.h>
main()
{
    char str[10],ch;
    int i,j,n,count;
    printf("Enter a string : ");
    scanf("%s",str);
    n=strlen(str);

    for(i=0;i<n;i++)
    {
        ch=str[i];
        if(ch!=' `')
        {
            count=0;
            for(j=0;j<n;j++)
            {
                if(ch==str[j])
                {
                    count++;
                    str[j]=' `';
                }
            }
            printf("%c occurs %d times\n",ch,count);
        }
    }
}
```

Problem 7

Write a program to enter a 4 digit number and display it in words.

```
/*P9.29*/
#include<stdio.h>
main()
{
    int n,num,d=0,dig[4];
    char *ones[ ]={ "", "One", "Two", "Three", "Four", "Five", "Six",
                    "Seven", "Eight", "Nine", "Ten" };
    char *el[ ]={ "Ten", "Eleven", "Twelve", "Thirteen", "Fourteen",
                  "Fifteen", "Sixteen", "Seventeen", "Eighteen",
                  "Nineteen" };
}
```

```

char *tens[ ]={ "", "", "Twenty", "Thirty", "Forty", "Fifty", "Sixty",
                "Seventy", "Eighty", "Ninety" };
printf("Enter a 4 digit number : ");
scanf("%d",&num);
n=num;
do
{
    dig[d]=n%10;
    n/=10;
    d++;
}while(n>0);

if(d==4)
    printf("%s Thousand",ones[dig[3]]);
if(d>=3&&dig[2]!=0)
    printf(" %s Hundred ",ones[dig[2]]);
if(d>=2)
{
    if (dig[1]==0)
        printf("%s\n",ones[dig[0]]);
    else if(dig[1]==1)
        printf("%s\n",el[dig[0]]);
    else
        printf("%s %s\n",tens[dig[1]],ones[dig[0]]));
}
if(d==1 && num!=0)
    printf("%s\n",ones[dig[0]]);
if(num==0)
    printf("Zero\n");
}

```

Problem 8

Write a function that converts a string to an integer(string contains integer in decimal number syst

```

/*P9.30*/
#include<stdio.h>
#include<string.h>
main()
{
    char str[20];
    printf("Enter a number : ");
    scanf("%s",str);
    printf("%d\n",str_to_i(str));
}
int str_to_i(char str[])
{
    int i,num=0;
    if(str[0]=='-')
        i=1;
    else
        i=0;

```

```
    while(i<strlen(str))
        num=num*10+(str[i++]-48);

    if(str[0]=='-')
        return -num;
    else
        return num;
```

Problem 9

Write functions to convert integer and float values to string.

```
/*P9.31*/
#include<stdio.h>
#include<string.h>
void i_to_str(int num,char str[],int base);
void f_to_str(float num,char str[]);
main()
{
    char str1[10],str2[10],str3[10],str4[10],str5[10];
    int x=45;
    float y=58.5;
    i_to_str(x,str1,16);puts(str1);
    i_to_str(x,str2,10);puts(str2);
    i_to_str(x,str3,8);puts(str3);
    i_to_str(x,str4,2);puts(str4);
    f_to_str(y,str5);puts(str5);
}

void i_to_str(int num,char str[],int b)
{
    int i=0,temp,rem,j;
    while(num>0)
    {
        rem=num%b;
        num/=b;
        if(rem>9 && rem<16)
            str[i++]=rem-10+'A';
        else
            str[i++]=rem+'0';
    }
    str[i]='\0';
    for(i=0,j=strlen(str)-1;i<j;i++,j--)
        /*Reverse the string*/
    {
        temp=str[i];
        str[i]=str[j];
        str[j]=temp;
    }
}

void f_to_str(float num,char str[])
{
    int i,k;
```

```

float f;
i=num;
i_to_str(i,str,10);
str[strlen(str)+1]='\0';
str[strlen(str)]='.';
f=num-i;
k=f*10000; /*we'll get the float value upto 4 decimal places*/
i_to_str(k, str+strlen(str),10);
}

```

Output:

2D
45
55
101101
58.5000

Exercise

Assume that stdio.h is included in all programs.

- (1) main()


```

{
    int i=0;
    char name[10]={‘M’, ‘o’, ‘h’, ‘i’, ‘n’, ‘i’, ‘\0’};
    while(name[i])
    {
        printf("%c",name[i]);
        i++;
    }
}
```
- (2) main()


```

{
    char *str;
    printf("Enter a string : ");
    gets(str);
    printf("String is %s\n",str);
}
```
- (3) #include<string.h>


```

main()
{
    char *str1="Good",*str2="Morning";
    strcat(str1,str2);
    printf("%s\n",str1);
}
```
- (4) #include<string.h>


```

main()
{
```

```
char str[10] = "How";
strcat(str, '?');
printf("%s\n", str);
}

(5) main()
{
    char str[] = "Vijaynagar";
    str = str + 5;
    printf("%s\n", str);
}

(6) main()
{
    char str[] = "Vijaynagar";
    func(str + 5);
}
func(char *str)
{
    printf("%s\n", str);
}

(7) main()
{
    char str[] = {70, 97, 105, 116, 104, 0};
    printf("%s\n", str);
}

(8) main()
{
    char str[] = "painstaking";
    char *p = str + 5;
    printf("%c\t", *p);
    printf("%s\n", p);
}

(9) main()
{
    printf("%c\t", "Determination"[2]);
    printf("%c\t", *("Determination"+2));
    printf("%s\t", "Determination"+2);
    printf("Determination"+2);
    printf("\t");
    printf("Determination"+strlen("Deepali"));
    printf("\t");
    printf("Determination"+sizeof("Deepali"));
    printf("\n");
}

(10) main()
{
    char str[] = "Lucknow";
```

```

        char *p=str;
        p++;
        p=p+2;
        p[3]='t';
        printf("%s      %s\n",str,p);
    }

(11) #include<string.h>
main()
{
    char *p[]={ "Orange", "Yellow", "Sky", "Blue", "Black"};
    char arr[10];
    printf("%s      %s      %s\n",p[1],p[2],p[3]);
    strcpy(arr,"Luck" "now");
    printf("%s\n",arr);
}

(12) #include<string.h>
main()
{
    char str1[15] = "Good ";
    char str2[] = "Evening";
    strcpy(str1+strlen(str1),str2);
    printf("%s\n",str1);
}

(13) main()
{
    char name[15] = "Vikramaditya";
    int i=0;
    while(name[i])
    {
        printf("%c ",name[i]);
        i=i+3;
    }
}
(14) main()
{
    char str[10][20];
    int i;
    for(i=0;i<10;i++)
        scanf("%s",str[i]);
    for(i=0;i<10;i++)
        printf("%s",str[i]);
}

(15) main()
{
    char *str[10];
    int i;
    for(i=0;i<10;i++)

```

String
Depth

```

        scanf("%s",str[i]);
    for(i=0;i<10;i++)
        printf("%s",str[i]);
}

(16) #include<string.h>
char *combine( char *arr1, char *arr2);
main()
{
    char *str=combine("Suresh", "Kumar");
    puts(str);
}
char *combine(char *arr1,char *arr2)
{
    char str[80];
    int x,y,i,j;
    x=strlen(arr1);
    y=strlen(arr2);
    strcpy(str,arr1);
    for(i=x,j=0;j<x+y;i++,j++)
        str[i]=arr2[j];
    str[i]='\0';
    return(str);
}

(17) main()
{
    char *str="Deepali Srivastava";
    int i=0;
    while(str[++i]);
    printf("%d\n",i);
}

(18) main()
{
    int d1,m1,y1;
    char date[11]="24/05/1973";
    date[2]=date[5]='\0';
    sscanf(date,"%d",&d1);
    sscanf(date+3,"%d",&m1);
    sscanf(date+6,"%d",&y1);
    date[2]=date[5]('/');
    printf("d1=%d,m1=%d,y1=%d\n",d1,m1,y1);
    printf("date = %s\n",date);
}

(19) main()
{
    char *str="doubtful";
    func(str);
}

```

```

func(char *p)
{
    if(*p != 'f')
    {
        printf("%c", *p);
        func(++p);
    }
}

(20)main()
{
    char *ptr;
    ptr="My name is %s and age is %d\n";
    printf(ptr,"Ranju",30);
}

(21)void func1(char arr[ ]);
void func2(char a[ ]);
main()
{
    int i;
    char arr[5];
    puts(arr);
    func1(arr);
    puts(arr);
    func2(arr);
    puts(arr);
}
void func1(char x[])
{
    x="Jack";
    puts(x);
}
void func2(char x[])
{
    x[0]='J',x[1]='i',x[2]='l',x[3]='l',x[4]='\0';
    puts(x);
}

(22)main()
{
    char *ptr;
    ptr="Every saint has a past,\n"
    Every sinner has a future.\n";
    printf("Giving ""is ""living.""\n");
    printf(ptr);
}

```

Programming Exercise

1. Write a function for performing case insensitive string comparison.

2. Write a program to accept a line of text and display the number of consonants and spaces in that line of text.
3. Write a function that searches for a character in the string and returns the number of occurrences of that character in the string. It should take two arguments, first a string and then a character.
4. Write a function which replaces all the occurrences of a character from a string with another character. It should take three arguments, a string and two characters.
5. Write a function which deletes all the occurrences of a character from a string. It should take two arguments, a string and a character.
6. Write a program to accept a line of text and a word. Display the number of occurrences of that word in the text.
7. Write a function to remove all the leading and trailing blanks from a string.
8. Write a program to input text and replace all the occurrences of word "Calcutta" by "Kolkata" in that text.
9. Write a program to accept a line of text and print that text after removing all spaces and delimiters.
10. Write a program to accept any 10 names and display those names after sorting them alphabetically in descending order.
11. Write a program to encode text and to decode the encoded text.
 - (i) Perform the encoding so that every character is replaced by its next character. For example replace a by b, b by c and so on. Replace z by a.

Plain text : program

Encoded text : qsphsbn

Decoded text : program

- (ii) Perform the encoding according to these replacements

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
m	n	k	g	h	d	t	a	b	w	v	u	p	r	q	c	z	j	x	i	e	y	f	l	o	s

Plain text : program

Encoded text :cjqtjmp

Decoded text :program

12. Input a string and change it so that the characters are placed in alphabetical order. For example the string "motivate" should be changed to "aeimotv"
13. Write a program to abbreviate input text. For example if the input is "World Health Organization", then the output should be WHO.
14. Write a function to extract a substring from a string. Assume that the substring starts at the ith character and is n characters long.
15. Write a program to input a number from 0 to 6 and print the corresponding day using array of pointers. For example if 0 is entered then print Sunday, if 1 is entered print Monday.
16. Consider this list of names of persons and cities

Reeta - Kanpur	Alok - Rampur	Manish - Lucknow
----------------	---------------	------------------

Reena - Rampur	Suresh - Lucknow	Divya -Kanpur
----------------	------------------	---------------

Deepali - Lucknow	Saumya - Rampur	Kriti -Renusagar
-------------------	-----------------	------------------

Write a program such that if a name of a person is entered then the city is displayed and if a city is entered then names of all people living in that city are displayed.

Hint: store the names of cities and persons in separate array of pointers to strings.

17. Write a program to input 5 lines of text and then store them as separate strings using array of pointers to strings.

Answers

- (1) Mohini
- (2) Pointer str contains a garbage value, it should be initialized before reading any string through gets().
- (3) The memory to which str1 points is not writable, since string constant "Good" is stored in read only memory by some compilers. We should use a character array str[15].
- (4) '?' represents a character, and strcat concatenates only strings, so we should write "?".
- (5) Error : since name of array is constant pointer and can't be altered.
- (6) nagar
- (7) Faith

Here the integer values are assumed to be ASCII equivalents of characters and these characters get stored in the array.

- (8) t taking
- (9) t t termination ; termination nation action
- (10) Lucknot knot
- (11) Yellow SkyBlue Black
Lucknow
- (12) Good Evening
- (13) V r a t
- (14) The first loop will input 10 strings and the next for loop will display them.
- (15) The pointers in the array of pointers are not initialized; so it is not valid to input strings.
- (16) str is a local array declared inside combine() and hence it exists only inside this function, so it is wrong to return its address to any other function. Sometimes this program may give desired output but the memory occupied by array str can be overwritten anytime. If we want this program to work properly then we can declare the array str in main(), and send it to combine().
- (17) 18
- (18) d1 = 24, m1 = 5, y1 = 1 3
date = 24/05/1973
- (19) doubt
- (20) My name is Ranju and age is 30
- (21) _____
Jack

Jill

Jill

The two dotted lines represent garbage value. If an array is passed as an argument, then inside the function we have a pointer to array and of course this is a local pointer variable. In func1(), x is declared as pointer to char and initialized with the address of array arr. Inside this function, the address of string constant "Jack" is assigned to pointer x. So now x has lost the address of array arr and it has nothing to do with this array.

So after call to func1(), the array arr still contains garbage value. Now func2() is called and array arr is sent to it. Here also x is declared as pointer to char and initialized with the address of arr, but inside this function we have not changed the address of x.

■ Giving is living.

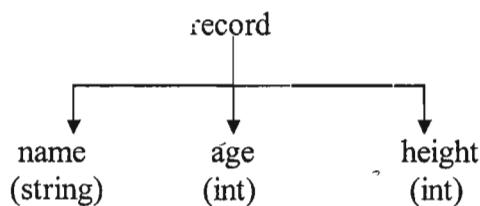
Every saint has a past, Every sinner has a future.

Adjacent string constants are concatenated. Any string constant can be continued on next line by putting a '\'.

Chapter 10

Structure And Union

Array is a collection of same type of elements but in many real life applications we may need to group different types of logically related data. For example if we want to create a record of a person which contains name, age and height of that person, then we can't use array because all the three data elements are of different types.



To store these related fields of different data types we can use a structure, which is capable of storing heterogeneous data. Data of different types can be grouped together under a single name using structure. The data elements of a structure are referred to as members.

10.1 Defining a Structure

Definition of a structure creates a template or format that describes the characteristics of its members. All the variables that would be declared of this structure type, will take the form of this template. The general syntax of a structure definition is-

```
struct tagname{  
    datatype member1;  
    datatype member2;  
    .....  
    .....  
    datatype memberN;  
};
```

Here **struct** is a keyword, which tells the compiler that a structure is being defined. member1, member2, ..., memberN are known as members of the structure and are declared inside curly braces. There should be a semicolon at the end of the curly braces. These members can be of any data type like int, char, float, array, pointers or another structure type. tagname is the name of the structure which is used further in the program to declare variables of this structure type.

Definition of a structure provides one more data type in addition to the built in data types. We can declare variables of this new data type that will have the format of the defined structure. It is important

note that definition of a structure template does not reserve any space in memory for the members; space is reserved only when actual variables of this structure type are declared. Although the syntax of declaration of members inside the template is identical to the syntax we use in declaring variables but these members are not variables, they don't have any existence until they are attached with a structure variable. The member names inside a structure should be different from one another but these names can be similar to any other variable name declared outside the structure. The member names of two different structures may also be same.

Let us take an example of defining a structure template.

```
struct student{
    char name[20];
    int rollno;
    float marks;
};
```

Here **student** is the structure tag and there are three members of this structure viz name, rollno and marks. Structure template can be defined globally or locally i.e. it can be placed before all functions in the program or it can be locally present in a function. If the template is global then it can be used by all functions while if it is local then only the function containing it can use it.

10.2 Declaring Structure Variables

By defining a structure we have only created a format, the actual use of structures will be when we declare variables based on this format. We can declare structure variables in two ways-

1. With structure definition
2. Using the structure tag

10.2.1 With Structure Definition

```
struct student{
    char name[20];
    int rollno;
    float marks;
}stu1,stu2,stu3;
```

Here stu1, stu2 and stu3 are variables of type **struct student**. When we declare a variable while defining the structure template, the tagname is optional. So we can also declare them as-

```
struct{
    char name[20];
    int rollno;
    float marks;
}stu1,stu2,stu3;
```

If we declare variables in this way, then we'll not be able to declare other variables of this structure type anywhere else in the program nor can we send these structure variables to functions. If a need arises to declare a variable of this type in the program then we'll have to write the whole template again. So although the tagname is optional it is always better to specify a tagname for the structure.

10.2.2 Using Structure Tag

We can also declare structure variables using structure tag. This can be written as-

```
struct student{
```

```

        char name[20];
        int rollno;
        float marks;
    };
struct student stu1,stu2;
struct student stu3;

```

Here stu1, stu2 and stu3 are structure variables that are declared using the structure tag student. Declaring a structure variable reserves space in memory. Each structure variable declared to be of type struct student has three members viz. name, rollno and marks. The compiler will reserve space for each variable sufficient to hold all the members. For example each variable of type struct student will occupy 26 (20+2+4) bytes.

10.3 Initialization Of Structure Variables

The syntax of initializing structure variables is similar to that of arrays. All the values are given in curly braces and the number, order and type of these values should be same as in the structure template definition. The initializing values can only be constant expressions.

```

struct student{
    char name[20];
    int rollno;
    float marks;
}stu1={"Mary", 25, 98};
struct student stu2={"John", 24, 67.5};

```

Here value of members of stu1 will be "Mary" for name, 25 for rollno, 98 for marks. The value of members of stu2 will be "John" for name , 24 for rollno, 67.5 for marks.

We cannot initialize members while defining the structure.

```

struct student {
    char name[20];
    int rollno;
    float marks=99; /*Invalid*/
}stu;

```

This is invalid because there is no variable called marks, and no memory is allocated for structure definition.

If the number of initializers is less than the number of members then the remaining members are initialized with zero. For example if we have this initialization-

```
struct student stu1 = {"Mary"};
```

Here the members rollno and marks of stu1 will be initialized to zero. This is equivalent to the initialization

```
struct student stu1 = {"Mary", 0, 0};
```

Some old compilers permit initialization of only global and static structures, but there is no such restriction in ANSI standard compilers.

10.4 Accessing Members of a Structure

For accessing any member of a structure variable, we use the dot (.) operator which is also known as the period or membership operator. The format for accessing a structure member is-

```
structvariable.member
```

Here on the left side of the dot there should be a variable of structure type and on right hand side there should be the name of a member of that structure. For example consider the following structure-

```
struct student{
    char name[20];
    int rollno;
    float marks;
};

struct student stu1,stu2;
```

name of stu1 is given by - stu1.name
 rollno of stu1 is given by - stu1.rollno
 marks of stu1 is given by - stu1.marks
 name of stu2 is given by - stu2.name
 rollno of stu2 is given by - stu2.rollno
 marks of stu2 is given by - stu2.marks

We can use stu1.name , stu1.marks, stu2.marks etc like any other ordinary variables in the program. They can be read, displayed, processed, assigned values or can be send to functions as arguments. We can't use student.name or student.rollno because student is not a structure variable, it is a structure tag.

```
/*P10.1 Program to display the values of structure members*/
#include<stdio.h>
#include<string.h>
struct student{
    char name[20];
    int rollno;
    float marks;
};

main()
{
    struct student stu1={"Mary",25,68};
    struct student stu2,stu3;
    strcpy(stu2.name,"John");
    stu2.rollno=26;
    stu2.marks=98;
    printf("Enter name, rollno and marks for stu3 : ");
    scanf("%s %d %f",stu3.name,&stu3.rollno,&stu3.marks);
    printf("stu1 : %s %d %.2f\n",stu1.name,stu1.rollno,stu1.marks);
    printf("stu2 : %s %d %.2f\n",stu2.name,stu2.rollno,stu2.marks);
    printf("stu3 : %s %d %.2f\n",stu3.name,stu3.rollno,stu3.marks);
}
```

Output:

Enter name, rollno and marks for stu3 : Tom 27 79.5
 stu1 : Mary 25 68.00
 stu2 : John 26 98.00
 stu3 : Tom 27 79.50

ach
essed
the dot

In this program we have declared three variables of type struct student. The first variable stu1 has been initialized, the members of second variable stu2 are given values using separate statements and the values for third variable stu3 are input by the user. Note that since stu2.name is an array so we can't assign a string to it using assignment operator, hence we have used the strcpy() function.

The dot operator is one of the highest precedence operators, its associativity is from left to right. Hence it will take precedence over all other unary, relational, logical, arithmetic and assignment operators. In an expression like ++stu.marks, first stu.marks will be accessed and then its value will be increased by 1.

10.5 Assignment of Structure Variables

We can assign values of a structure variable to another structure variable, if both variables are defined of the same structure type. For example-

```
/*P10.2 Program to assign a structure variables to another structure variable*/
struct student{
    char name[20];
    int rollno;
    float marks;
};

main()
{
    struct student stu1={"Oliver",12,98};
    struct student stu2;
    stu2=stu1;
    printf("stu1 : %s %d %.2f\n",stu1.name,stu1.rollno,stu1.marks);
    printf("stu2 : %s %d %.2f\n",stu2.name,stu2.rollno,stu2.marks);
}
```

Output:

```
stu1 : Oliver 12 98.00
stu2 : Oliver 12 98.00
```

Unary, relational, arithmetic, bitwise operators are not allowed with structure variables. We can use these operators with the members provided the member is not a structure itself.

10.6 Storage of Structures in Memory

The members of structures are stored in consecutive memory locations.

```
/*P10.3 Program to show that members of structure are stored in consecutive
memory locations*/
#include<stdio.h>
main()
{
    struct student{
        char name[5];
        int rollno;
        float marks;
    }stu;
```

```
been
values
assign
    printf("Address of name = %u\n",stu.name);
    printf("Address of rollno = %u\n",&stu.rollno);
    printf("Address of marks = %u\n",&stu.marks);
```

Output:

```
Address of stu.name = 65514
Address of stu.rollno = 65519
Address of stu.marks = 65521
```

The output may be different on different machines, and the number of bytes occupied may also vary because of the reasons explained in next section, but the main point to be noted here is that structure members are stored in consecutive memory locations.

10.7 Size of Structure

We may need to find out the size of structure in some situations like reading or writing to files. To find out the size of a structure by sizeof operator, we can either use the structure variable name or the tagname with the struct keyword. For example-

```
sizeof( struct student )
sizeof( stu1 )
sizeof( stu2 )
```

Here if stu1 and stu2 are variables of type struct student, then all the three expressions will give the same result.

Size of structures may be different on different machines. This is because of certain memory alignment restrictions on some computers. For example some machines store integers only at even addresses and long ints only at addresses which are multiple of 4. This is called aligning of data. Consider this structure-

```
struct
{
    char ch;
    int num;
}var;
```

Now here suppose var.ch is stored at an even address, then the next byte will be left unused since int can't be stored at an odd address. So instead of occupying 3 bytes this structure variable will occupy 4 bytes with a hole of unused byte in between. Due to these reasons, size of whole structure may not be equal to the sum of sizes of its members. So it is always better to find the size of structure variable by using sizeof operator rather than using the sum of sizes of its members.

10.8 Array of Structures

We know that array is a collection of elements of same datatype. We can declare array of structures where each element of array is of structure type. Array of structures can be declared as-

```
struct student stu[10];
```

Here stu is an array of 10 elements, each of which is a structure of type struct student, means each element of stu has 3 members, which are name, rollno and marks. These structures can be accessed through subscript notation. To access the individual members of these structures we'll use the dot operator as usual.

stu[0].name	stu[0].rollno	stu[0].marks
stu[1].name	stu[1].rollno	stu[1].marks
stu[2].name	stu[2].rollno	stu[2].marks
.....
.....
stu[9].name	stu[9].rollno	stu[9].marks

All the structures of an array are stored in consecutive memory locations.

```
/*P10.4 Program to understand array of structures*/
#include<stdio.h>
struct student{
    char name[20];
    int rollno;
    float marks;
};

main()
{
    int i;
    struct student stuarr[10];
    for(i=0;i<10;i++)
    {
        printf("Enter name, rollno and marks : ");
        scanf("%s%d%f",stuarr[i].name,&stuarr[i].rollno,
&stuarr[i].marks);
    }
    for(i=0;i<10;i++)
        printf("%s %d %f \n",stuarr[i].name,stuarr[i].rollno,
stuarr[i].marks);
}
```

In some compilers the above program may not work correctly and will give the message "floating point formats not linked" at run time. This problem occurs because the floating point formats (for scanf and other related functions) are not always linked, to reduce the size of executable file. The solution to this problem will be given in the manual of your compiler. For example, Borland C suggests inclusion of these two lines to solve this problem.

```
extern unsigned _floatconvert;
#pragma extref _floatconvert
```

Another way to avoid the above problem is to insert a definition of a function like this.

```
void link()
{ float x, *ptr = &x; }
```

The array of structures may be initialized using the same syntax as in arrays. For example-

```
struct student stuarr[3]={
    {"Mary",12,98.5},
    {"John",11,97},
    {"Tom",12,89.5}
};
```

The inner pairs of braces are optional if all the initializers are present in the list.

10.9 Arrays Within Structures

We can have an array as a member of structure. In structure student, we have taken the member name as an array of characters. Now we'll declare another array inside the structure student.

```
struct student{  
    char name[20];  
    int rollno;  
    int submarks[4];  
};
```

The array submarks denotes the marks of students in 4 subjects.

If stu is a variable of type struct student then-

- stu.submarks[0] - Denotes the marks of the student in first subject
- stu.submarks[1] - Denotes the marks in second subject.
- stu.name[0] - Denotes the first character of the name member.
- stu.name[4] - Denotes the fifth character of the name member.

If stuarr[10] is an array of type struct student then-

- stuarr[0].submarks[0] - Denotes the marks of first student in first subject
- stuarr[4].submarks[3] - Denotes the marks of fifth student in fourth subject.
- stuarr[0].name[0] - Denotes the first character of name member of first student
- stuarr[5].name[7] - Denotes the eighth character of name member of sixth student

```
/* P10.5 Program to understand arrays within structures */  
#include<stdio.h>  
struct student{  
    char name[20];  
    int rollno;  
    int submarks[4];  
};  
main()  
  
int i,j;  
struct student stuarr[3];  
for(i=0;i<3;i++)  
{  
    printf("Enter data for student %d\n",i+1);  
    printf("Enter name : ");  
    scanf("%s",stuarr[i].name);  
    printf("Enter roll number : ");  
    scanf("%d", &stuarr[i].rollno);  
    for(j=0;j<4;j++)  
    {  
        printf("Enter marks for subject %d : ",j+1);  
        scanf("%d", &stuarr[i].submarks[j]);  
    }  
}
```

```

for(i=0;i<3;i++)
{
    printf("Data of student %d\n",i+1);
    printf("Name : %s, Roll number : %d\nMarks : ",stuarr[i].name,
    stuarr[i].rollno);
    for(j=0;j<4;j++)
        printf("%d ",stuarr[i].submarks[j]);
    printf("\n");
}
}

```

10.10 Nested Structures (Structure Within Structure)

The members of a structure can be of any data type including another structure type i.e. we can include a structure within another structure. A structure variable can be a member of another structure. This is called nesting of structures.

```

struct tag1{
    member1;
    member2;
    .....
    struct tag2{
        member1;
        member2;
        .....
        member m;
    }var1;
    .....
    member n;
}var2;

```

For accessing member1 of inner structure we'll write-

```
var2.var1.member1
```

Here is an example of nested structures-

```

struct student{
    char name[20];
    int rollno;
    struct date{
        int day;
        int month;
        int year;
    } birthdate;
    float marks;
}stu1,stu2;

```

Here we have defined a structure date inside the structure student. This structure date has three members day, month, year and birthdate is a variable of type struct date. We can access the members of inner structure as-

stu1.birthdate.day	→	day of birthdate of stu1
stu1.birthdate.month	→	month of birthdate of stu1
stu1.birthdate.year	→	year of birthdate of stu1

stu2.birthdate.day → day of birthdate of stu2

Here we have defined the template of structure date inside the structure student, we could have defined it outside and declared its variables inside the structure student using the tag. But remember if we define the inner structure outside, then this definition should always be before the definition of outer structure. Here in this case the date structure should be defined before the student structure.

```
struct date{
    int day;
    int month;
    int year;
};

struct student{
    char name[20];
    int rollno;
    float marks;
    struct date birthdate;
}stu1,stu2;
```

The advantage of defining date structure outside is that we can declare variables of date type anywhere else also. Suppose we define a structure teacher, then we can declare variables of date structure inside it as-

```
struct teacher{
    char name[20];
    int age;
    float salary;
    struct date birthdate,joindate;
}t1,t2;
```

The nested structures may also be initialized at the time of declaration. For example-

```
struct teacher t1= { "Sam", 34, 9000, {8, 12, 1970}, { 1, 7, 1995 } } ;
```

Nesting of a structure within itself is not valid. For example the following structure definition is invalid-

```
struct person{
    char name[20];
    int age;
    float height;
    struct person father; /*Invalid*/
}emp;
```

The nesting of structures can be extended to any level. The following example shows nesting at level three i.e. first structure is nested inside a second structure and second structure is nested inside a third structure.

```
struct time
{
    int hr;
    int min;
    int sec;
};

struct date
{
    int day;
    int month;
```

```

    int year;
    struct time t;
};

struct student
{
    char name[20];
    struct date dob; /*Date of birth*/
}stu1,stu2;

```

To access hour of date of birth of student stu1 we can write-

stu1.dob.t.hr

10.11 Pointers to Structures

We have studied that pointer is a variable which holds the starting address of another variable of a data type like int, float or char. Similarly we can have pointer to structure, which can point to the start address of a structure variable. These pointers are called structure pointers and can be declared

```

struct student{
    char name[20];
    int rollno;
    int marks;
};

struct student stu,*ptr;

```

Here ptr is a pointer variable that can point to a variable of type struct student. We'll use the & operator to access the starting address of a structure variable, so ptr can point to stu by writing-

ptr = &stu;

There are two ways of accessing the members of structure through the structure pointer.

As we know ptr is a pointer to a structure, so by dereferencing it we can get the contents of struct variable. Hence *ptr will give the contents of stu. So to access members of a structure variable we can write-

```

(*ptr).name
(*ptr).rollno
(*ptr).marks

```

Here parentheses are necessary because dot operator has higher precedence than the * operator. This syntax is confusing so C has provided another facility of accessing structure members through pointers. We can use the arrow operator (→) which is formed by hyphen symbol and greater than symbol. we can access the members as-

```

ptr->name
ptr->rollno
ptr->marks

```

The arrow operator has same precedence as that of dot operator and it also associates from left to right.

```

/*P10.6 Program to understand pointers to structures*/
#include<stdio.h>
struct student{
    char name[20];
}

```

```

        int rollno;
        int marks;
    };

main()
{
    struct student stu={"Mary", 25, 68};
    struct student *ptr=&stu;
    printf("Name - %s\t", ptr->name);
    printf("Rollno - %d\t", ptr->rollno);
    printf("Marks - %d\n", ptr->marks);
}

```

We can also have pointers that point to individual members of a structure variable. For example-

```

int *p = &stu.rollno;
float *ptr = &stu.marks;

```

The expression &stu.rollno is equivalent to &(stu.rollno) because the precedence of dot operator is more than that of address operator.

10.12 Pointers Within Structures

A pointer can also be used as a member of structure. For example we can define a structure like this-

```

struct student{
    char name[20];
    int *ptrmem;
};

struct student stu, *stuptr=&stu;

```

Here ptrmem is pointer to int and is a member of the structure student.

To access the value of ptrmem, we'll write

stu.ptrmem or stuptr->ptrmem

To access the value pointed to by stu.ptrmem, we'll write

*stu.ptrmem or *stuptr->ptrmem

Since the priority of dot and arrow operators is more than that of dereference operator, hence the expression *stu.ptrmem is equivalent to *(stu.ptrmem), and the expression *stuptr->ptrmem is equivalent to *(stuptr->ptrmem).

10.13 Structures And Functions

Structures may be passed as arguments to function in different ways. We can pass individual members, whole structure variable or structure pointers to the function. Similarly a function can return either a structure member or whole structure variable or a pointer to structure.

10.13.1 Passing Structure Members As Arguments

We can pass individual structure members as arguments to functions like any other ordinary variable.

```

/*P10.7 Program to understand how structure members are sent to a function
*/
#include<stdio.h>
#include<string.h>

```

```

struct student {
    char name[20];
    int rollno;
    int marks;
};

display(char name[], int rollno, int marks);
main()
{
    struct student stu1={"John", 12, 87};
    struct student stu2;
    strcpy(stu2.name, "Mary");
    stu2.rollno=18;
    stu2.marks=90;
    display(stu1.name, stu1.rollno, stu1.marks);
    display(stu2.name, stu2.rollno, stu2.marks);
}
display(char name[], int rollno, int marks)
{
    printf("Name - %s\t", name);
    printf("Rollno - %d\t", rollno);
    printf("Marks - %d\n", marks);
}

```

Output:

Name - John Rollno - 12 Marks - 87

Name - Mary Rollno - 18 Marks - 90

Here we have passed members of the variables stu1 and stu2 to the function display(). The names of the formal arguments can be similar to the names of the members. We can pass the arguments using call by reference also so that the changes made in the called function will be reflected in the calling function. In that case we'll have to send the addresses of the members. It is also possible to return a single member from a function.

10.13.2 Passing Structure Variable As Argument

Passing individual members to function becomes cumbersome when there are many members and the relationship between the members is also lost. We can pass the whole structure as an argument.

```

/*P10.8 Program to understand how a structure variable is sent to a function*/
#include<stdio.h>
struct student{
    char name[20];
    int rollno;
    int marks;
};

display(struct student);
main()
{
    struct student stu1={"John", 12, 87};
    struct student stu2={"Mary", 18, 90};
}

```

```

display(stu1);
display(stu2);

display(struct student stu)

printf("Name - %s\t",stu.name);
printf("Rollno - %d\t",stu.rollno);
printf("Marks - %d\n",stu.marks);

```

Output:

Name - John Rollno - 12 Marks - 87

Name - Mary Rollno - 18 Marks - 90

Here it is necessary to define the structure template globally because it is used by both functions to declare variables.

The name of a structure variable is not a pointer unlike arrays, so when we send a structure variable as an argument to a function, a copy of the whole structure is made inside the called function and all the work is done on that copy. Any changes made inside the called function are not visible in the calling function since we are only working on a copy of the structure variable, not on the actual structure variable.

10.13.3 Passing Pointers To Structures As Arguments

If the size of a structure is very large, then it is not efficient to pass the whole structure to the function since a copy of it has to be made inside the called function. In this case it is better to send address of the structure, which will improve the execution speed.

We can access the members of the structure variable inside the calling function using arrow operator. In this case any changes made to the structure variable inside the called function, will be visible in the calling function since we are actually working on the original structure variable.

```

/*P10.9 Program to understand how a pointer to structure variable is
sent to a function*/
#include<stdio.h>
struct student{
    char name[20];
    int rollno;
    int marks;
};

display(struct student *);
inc_marks(struct student *);

main()
{
    struct student stu1={"John",12,87};
    struct student stu2={"Mary",18,90};
    inc_marks(&stu1);
    inc_marks(&stu2);
    display(&stu1);
    display(&stu2);
}

```

```

inc_marks(struct student *stuptr)
{
    (stuptr->marks)++;
}
display( struct student *stuptr)
{
    printf("Name - %s\t", stuptr->name);
    printf("Rollno - %d\t", stuptr->rollno);
    printf("Marks - %d\n", stuptr->marks);
}

```

Output :

Name - John Rollno - 12 Marks - 88

Name - Mary Rollno - 18 Marks - 91

10.13.4 Returning A Structure Variable From Function

Structure variables can be returned from functions as any other variable. The returned value can be assigned to a structure of the appropriate type.

```

/*P10.10 Program to understand how a structure variable is returned from a function*/
#include<stdio.h>
struct student{
    char name[20];
    int rollno;
    int marks;
};
void display(struct student);
struct student change(struct student stu);
main()
{
    struct student stu1={"John",12,87};
    struct student stu2={"Mary",18,90};
    stu1=change(stu1);
    stu2=change(stu2);
    display(stu1);
    display(stu2);
}
struct student change(struct student stu)
{
    stu.marks=stu.marks+5;
    stu.rollno=stu.rollno-10;
    return stu;
}

void display(struct student stu)
{
    printf("Name - %s\t", stu.name);
    printf("Rollno - %d\t", stu.rollno);
    printf("Marks - %d\n", stu.marks);
}

```

Output:

```
Name - John Rollno - 2 Marks - 92
Name - Mary Rollno - 8 Marks - 95
```

10.13.5 Returning A Pointer To Structure From A Function

Pointers to structures can also be returned from functions. In the following program, the function func() returns a pointer to structure.

```
*P10.11 Program to understand how a pointer to structure is returned
from a function*
#include<stdio.h>
#include<string.h>
struct student{
    char name[20];
    int rollno;
    int marks;
};

void display(struct student * );
struct student *func();
main()

    struct student *stuptr;
    stuptr=func();
    display(stuptr);

    struct student *func()

        struct student *ptr;
        ptr=(struct student *) malloc(sizeof(struct student));
        strcpy(ptr->name,"Joseph");
        ptr->rollno=15;
        ptr->marks=98;
        return ptr;

    void display(struct student *stuptr)

        printf("Name - %s\t",stuptr->name);
        printf("Rollno - %d\t",stuptr->rollno);
        printf("Marks - %d\n",stuptr->marks);
```

Output:

```
Joseph 15 98
```

10.13.6 Passing Array Of Structures As Argument

As we pass an array to a function, similarly we can pass the array of structure to function, where each element of array is of structure type. This can be written as-

```
*P10.12 Program to understand how an array of structures is sent to
a function */
#include<stdio.h>
```

```

struct student{
    char name[20];
    int rollno;
    int marks;
};

void display(struct student);
void dec_marks(struct student stuarr[ ]);

main()
{
    int i;
    struct student stuarr[3]={
        {"Mary",12,98},
        {"John",11,97},
        {"Tom",13,89}
    };
    dec_marks(stuarr);
    for(i=0;i<3;i++)
        display(stuarr[i]);
}

void dec_marks(struct student stuarr[])
{
    int i;
    for(i=0;i<3;i++)
        stuarr[i].marks=stuarr[i].marks-10;
}

void display(struct student stu)
{
    printf("Name - %s\t", stu.name);
    printf("Rollno - %d\t", stu.rollno);
    printf("Marks - %d\n", stu.marks);
}

```

Output:

Name - Mary Rollno - 12 Marks - 88

Name - John Rollno - 11 Marks - 87

Name - Tom Rollno - 13 Marks - 79

All the changes made in the array of structures inside the called function will be visible in the calling function.

The following program calculates the total marks and grade of students, and then sorts the names of students on the basis of marks and prints out the sorted records.

```

/*P10.13 Program to find out and print the grade of students*/
#include<stdio.h>
#define N 5
struct student
{
    char name[20];
    int rollno;
    int marks[6];
    int total;
}

```

```
char grade;
void display(struct student arr);
void calculate(struct student arr[ ]);
void sort( struct student arr[ ]);
main()
{
    struct student stu[N],temp;
    int i,j;
    for(i=0;i<N;i++)
    {
        printf("Enter name : ");
        scanf("%s",stu[i].name);
        printf("Enter rollno : ");
        scanf("%d",&stu[i].rollno);
        stu[i].total=0;
        printf("Enter marks in 6 subjects : ");
        for(j=0;j<6;j++)
            scanf("%d",&stu[i].marks[j]);
    }
    calculate(stu);
    sort(stu);
    for(i=0;i<N;i++)
        display(stu[i]);
}

void calculate(struct student stu[])
{
    int i,j;
    for(i=0;i<N;i++)
    {
        for(j=0;j<6;j++)
            stu[i].total+=stu[i].marks[j];
        if(stu[i].total>500)
            stu[i].grade='A';
        else if(stu[i].total>400)
            stu[i].grade='B';
        else if(stu[i].total>250)
            stu[i].grade='C';
        else
            stu[i].grade='D';
    }
}

void sort(struct student stu[])
{
    int i,j;
    struct student temp;
    for(i=0;i<N-1;i++)
        for(j=i+1;j<N;j++)
            if(stu[i].total<stu[j].total)
            {
                temp=stu[i];
```

```

        stu[i]=stu[j];
        stu[j]=temp;
    }
}

void display(struct student *stu)
{
    int i;
    printf("Rollno - %d\t", stu.rollno);
    printf("Name - %s\n", stu.name);
    printf("Total - %d\t", stu.total);
    printf("Grade - %c\n\n", stu.grade);
}

```

In the following program the records are sorted on different keys depending on the choice of the user.

```

/*P10.14 Program to sort the records on different keys*/
#include<stdio.h>
#define N 5
struct date{
    int day;
    int month;
    int year;
};

struct employee{
    char name[20];
    struct date dob;
    struct date doj;
    int salary;
};

void sort_name(struct employee emp[]);
void sort_dob(struct employee emp[]);
void sort_doj(struct employee emp[]);
void sort_salary(struct employee emp[]);
void display(struct employee emp[]);
int datecmp(struct date date1, struct date date2 );
main( )
{
    struct employee emp[N];
    int i,choice;
    for(i=0;i<N;i++)
    {
        printf("Enter name :");
        scanf("%s", emp[i].name);
        printf("Enter date of birth(dd/mm/yy) : ");
        scanf("%d/%d/%d", &emp[i].dob.day, &emp[i].dob.month,
        &emp[i].dob.year);
        printf("Enter date of joining(dd/mm/yy) : ");
        scanf("%d/%d/%d", &emp[i].doj.day, &emp[i].doj.month,
        &emp[i].doj.year);
        printf("Enter salary : ");
        scanf("%d", &emp[i].salary);
    }

    sort_name(emp);
    sort_dob(emp);
    sort_doj(emp);
    sort_salary(emp);
    display(emp);
}

```

```

    printf("\n");
}
while(1)
{
    printf("1.Sort by name\n");
    printf("2.Sort by date of birth\n");
    printf("3.Sort by date of joining\n");
    printf("4.Sort by salary\n");
    printf("5.Exit\n");
    printf("6.Enter your choice :");
    scanf("%d", &choice);
    switch(choice)
    {
        case 1:
            sort_name(emp);
            break;
        case 2:
            sort_dob(emp);
            break;
        case 3:
            sort_doj(emp);
            break;
        case 4:
            sort_salary(emp);
            break;
        case 5:
            exit(1);
        default:
            printf("Wrong choice\n");
    }
    printf("\n");
    display(emp);
}

```

```

void display(struct employee emp[])
{
    int i;
    printf("NAME\t\tDOB\t\tDOJ\t\tSALARY\n");
    for(i=0;i<N;i++)
    {
        printf("%s\t\t",emp[i].name);
        printf("%d/%d/%d\t\t",emp[i].dob.day,emp[i].dob.month,
               emp[i].dob.year);
        printf("%d/%d/%d\t\t",emp[i].doj.day,emp[i].doj.month,
               emp[i].doj.year);
        printf("%d\n",emp[i].salary);
    }
    printf("\n");
}

```

```
void sort_name(struct employee emp[])
```

```
{  
    struct employee temp;  
    int i,j;  
    for(i=0;i<N-1;i++)  
        for(j=i+1;j<N;j++)  
            if(strcmp(emp[i].name,emp[j].name)>0)  
            {  
                temp=emp[i];  
                emp[i]=emp[j];  
                emp[j]=temp;  
            }  
    }  
void sort_dob(struct employee emp[])  
{  
    struct employee temp;  
    int i,j;  
    for(i=0;i<N-1;i++)  
        for(j=i+1;j<N;j++)  
            if(datecmp(emp[i].dob,emp[j].dob)>0)  
            {  
                temp=emp[i];  
                emp[i]=emp[j];  
                emp[j]=temp;  
            }  
    }  
void sort_doj(struct employee emp[])  
{  
    struct employee temp;  
    int i,j;  
    for(i=0;i<N-1;i++)  
        for(j=i+1;j<N;j++)  
            if(datecmp(emp[i].doj,emp[j].doj)>0)  
            {  
                temp=emp[i];  
                emp[i]=emp[j];  
                emp[j]=temp;  
            }  
    }  
void sort_salary(struct employee emp[])  
{  
    struct employee temp;  
    int i,j;  
    for(i=0;i<N-1;i++)  
        for(j=i+1;j<N;j++)  
            if(emp[i].salary<emp[j].salary)  
            {  
                temp=emp[i];  
                emp[i]=emp[j];  
                emp[j]=temp;  
            }  
    }  
/*Returns 1 if date1 < date2, returns -1 if date1 > date2, return
```

```

    equal*/
int datecmp(struct date date1, struct date date2)

    if(date1.year<date2.year)
        return 1;
    if(date1.year>date2.year)
        return -1;
    if(date1.month<date2.month)
        return 1;
    if(date1.month>date2.month)
        return -1;
    if(date1.day<date2.day)
        return 1;
    if(date1.day>date2.day)
        return -1;
    return 0;
}

```

10.14 Self Referential Structures

A structure that contains pointers to structures of its own type is known as self referential structure.
For example-

```

struct tag{
    datatype member1;
    datatype member2;
    .....
    .....
    struct tag *ptr1;
    struct tag *ptr2;
};

```

Here ptr1 and ptr2 are structure pointers that can point to structure variables of type struct tag, so struct tag is a self referential structure. These types of structures are helpful in implementing data structures like linked lists and trees. We'll discuss the data structure linked list in this chapter.

10.15 Linked List

List is a collection of elements. There are two ways of maintaining a list in computer memory. The first way is to take an array for storing the elements of the list, but arrays have some disadvantages. Insertion and deletion of an element from an array requires more processing. If the number of elements in the list is less than the size of array then memory will be wasted and if the number of elements exceeds the size of array then also we'll have problems.

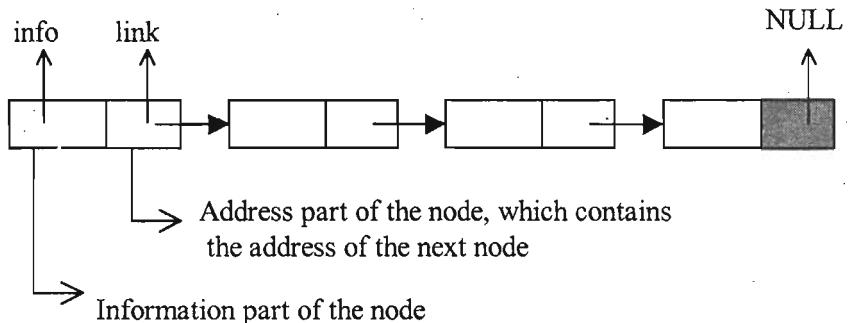
The second way of implementing a list in memory is by using a self referential structure. These types of lists are known as linked lists. A linked list is made up of nodes where each node has two parts, first part contains the information and second part contains the address of the next node. The address part of the last node of linked list will be NULL. The general form of a node of linked list is-

```

struct node {
    datatype member1;
    datatype member2;
}

```

```
.....  
.....  
    struct node *link; /* Pointer to next node of the list  
};
```



The nodes of a linked list are not stored contiguously in memory. In array list we could perform the operations using the array name. In the case of linked list, we'll perform all the operations w the help of a pointer that points to the first node the linked list. This pointer is generally named `start` and it is the only source through which we can access our linked list. The list will be considered em if the pointer `start` contains `NULL` value.

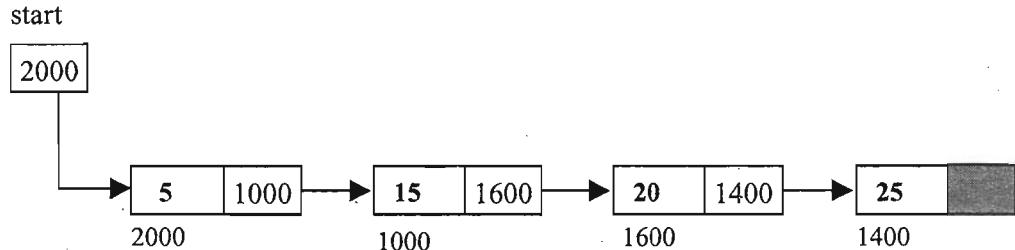
Now we'll take a linked list that contains an integer value in the information part. The structure the nodes of this list will be defined as-

```
struct node{  
    int info;  
    struct node *link;  
};
```

The pointer variable `start` will be declared as-

```
struct node *start;
```

Here is a linked list that has 4 nodes-



We can clearly see that the address of each node is stored in the link part of the previous node, the address of first node is stored in the pointer variable `start`. The link part of the last node cont `NULL`.

We'll discuss the following operations on this list.

- (i) Traversal of list
- (ii) Searching an element

- ④ Insertion of an element
- ⑤ Deletion of an element
- ⑥ Creation of list
- ⑦ Reversal of list

11.15.1 Traversing a Linked List

Traversal of a list means accessing each node exactly once. For this we'll take a structure pointer ptr. Initially ptr is assigned the value of start.

```
ptr = start;
```

So now ptr points to the first node of linked list. We can access the info part of first node by writing `ptr->info`.

Now we'll shift the pointer ptr forward so that it points to the next element. This can be easily done by assigning the address of the next element to ptr as-

```
ptr = ptr->link;
```

Now ptr has address of the next element. Similarly we can traverse each element of linked list through this assignment until ptr has NULL value, which is link part value of last element. So the linked list can be traversed as-

```
while(ptr!=NULL)
```

```
    printf("%d ",ptr->info);
    ptr=ptr->link;
```

11.15.2 Searching in a Linked List

For searching an element, we traverse the linked list and while traversing we compare the info part of each element with the given element. It can be written as-

```
while(ptr!=NULL)
```

```
    if(ptr->info==data) /*Search successful*/
        .....
    else
        ptr=ptr->link; /*Go to next element*/
```

Here data is the element, which we want to search.

11.15.3 Insertion into a Linked List

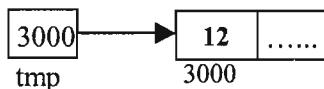
Insertion in a linked list may be possible in two ways-

1. Insertion in the beginning
2. Insertion in between or at the end.

To insert a node, initially we'll dynamically allocate space for that node using `malloc()`. Suppose `tmp` is a pointer that points to this dynamically allocated node. In the info part of the node we'll put the data value.

```
tmp = (struct node *)malloc( sizeof(struct node) );
tmp->info = data;
```

The link part of the node contains garbage value, we'll assign address to it separately in the two cases.



10.15.4 Insertion in the Beginning

After insertion the new node will become the first node, and the node which is currently the first node will come at second place. So we just have to adjust the links of the nodes.

Assign the value of start to the link part of inserted node as-

```
tmp->link = start;
```

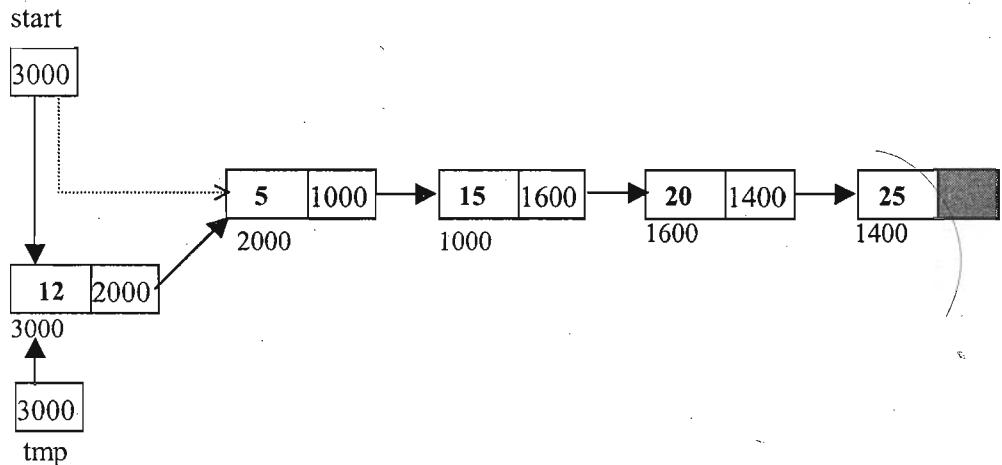
Now inserted node points to the next node, which was beginning node of the linked list.

Now inserted node is the first node of the linked list. So start should be reassigned as-

```
start = tmp
```

Now start will point to the inserted node, which is first node of the linked list.

The following figure shows this process, dotted lines represent the link before insertion.



10.15.5 Insertion in Between or at the end

First we traverse the linked list for obtaining the node after which we want to insert the element. We obtain pointer q which points to the element after which we have to insert new node. For inserting the element after the node, we give the link part of that node to the link part of inserted node and address of the inserted node is placed into the link part of the previous node.

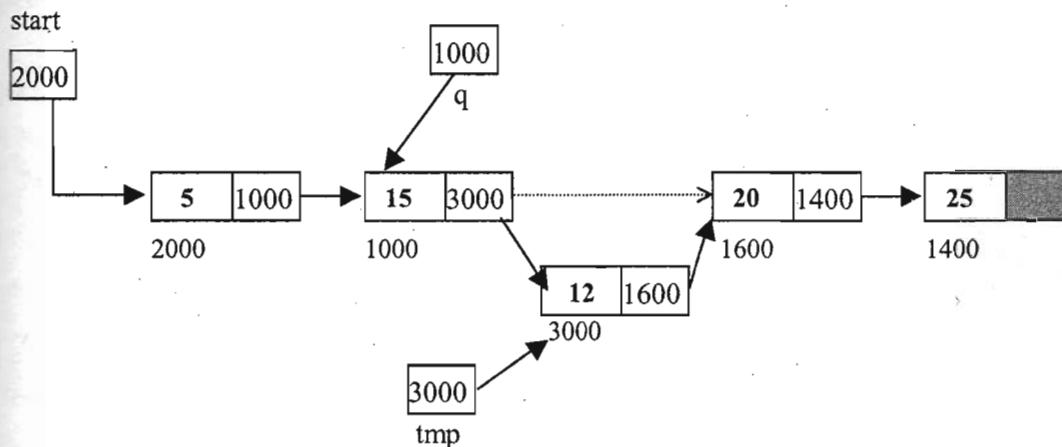
```
tmp->link = q->link;
```

```
q->link = tmp;
```

Here q is pointing to the previous node. After statement 1, link of inserted node will point to the next node and after statement 2 link of previous node will point to the inserted node.

If the node is to be inserted at the end then also the above two statements would work. In that case pointer q will point to the last node, hence q->link will be NULL. So after statement 1, link of inser-

ses. node will have NULL hence it will become the last node.



10.15.6 Deletion From A Linked List

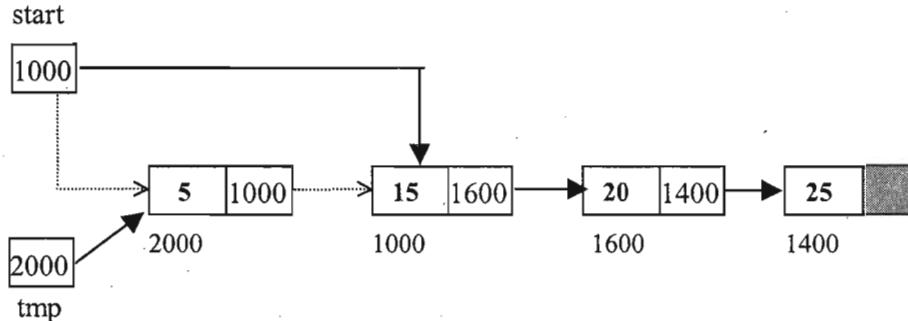
For deleting a node from the linked list, first we traverse the linked list and compare with each element. After finding the element there may be two cases for deletion-

1. Deletion of first node
2. Deletion of a node in between or at the end.

The deletion of nodes also requires adjustment of links. These link manipulations will delete the node from the linked list, but it will still occupy memory. Since the memory for each node was dynamically allocated by us, so it is our responsibility to release that memory after deletion of the node. To release the memory we'll use the function `free()`, and for that we'll need a pointer to the node that has to be deleted. We'll take a pointer `tmp` that will point to the node to be deleted and after deletion of the node we'll call `free` as-

```
free(tmp);
```

10.15.7 Deletion of First Node



Since the node to be deleted is the first node hence `tmp` will be assigned the address of first node.

```
tmp = start;
```

So now `tmp` points to the first node, which has to be deleted.

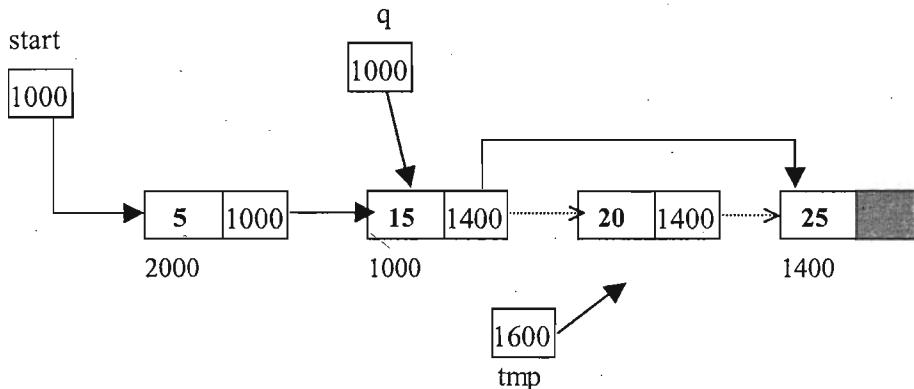
Since start points to the first node of linked list, so `start->link` will point to the second node of link list. After deletion of first node, the second node would become the first one, so start should be assigned the address of the second node as-

```
start = start->link;
```

Now we should free the element to be deleted, which is pointed to by `tmp`.

```
free( tmp );
```

10.15.8 Deletion of a Node in Between or at the End



We'll traverse the list and obtain a pointer which points to the node that is just before the node to be deleted. Suppose this pointer is `q`, so the node to be deleted will be pointed by `q->link`, hence we'll assign this value to `tmp` as-

```
tmp = q->link;
```

So now `tmp` will point to the node to be deleted.

Now we'll assign the link part of the node to be deleted to the link part of the previous node. This can be done as-

```
q->link = tmp->link;
```

Now link of previous node will point to node that is just after the node to be deleted.

Finally we'll free the deleted node as-

```
free(tmp);
```

If node to be deleted is last node of linked list then second statement can be written as-

```
q->link = NULL;
```

10.15.9 Creation Of List

Creation of list is very simple if you have understood the above operations. It is similar to insert of an element at the end of the list. Initially we'll allocate memory for a node as-

```
tmp = malloc(sizeof(struct node));
```

```
tmp->info = data;
```

```
tmp->link = NULL;
```

If the list is empty and we are inserting the first element, then we'll have to initialize the pointer as-

Structure and Union

```
if ( start == NULL ) /*If list is empty */  
    start = tmp;
```

After that we'll keep on inserting new nodes at the end of the list.

Now we are in a position to write a program that creates a linked list and performs operations on it. Each operation has been implemented inside a separate function. The pointer start is taken as global so that all functions can access it and it makes our program simple. If we declare start inside main() then we'll have to pass it to all other functions and use a pointer to pointer inside those functions.

```
/*P10.15 Program of single linked list*/  
#include <stdio.h>  
#include <malloc.h>  
struct node  
{  
    int info;  
    struct node *link;  
}*start;  
  
main()  
{  
    int choice,n,m,position,i;  
    while(1)  
    {  
        printf("1.Create List\n");  
        printf("2.Add at begining\n");  
        printf("3.Add after \n");  
        printf("4.Delete\n");  
        printf("5.Display\n");  
        printf("6.Search\n");  
        printf("7.Quit\n");  
        printf("Enter your choice : ");  
        scanf("%d",&choice);  
        switch(choice)  
        {  
            case 1:  
                start=NULL;  
                printf("How many nodes you want : ");  
                scanf("%d",&n);  
                for(i=0;i<n;i++)  
                {  
                    printf("Enter the element : ");  
                    scanf("%d",&m);  
                    create_list(m);  
                }  
                break;  
            case 2:  
                printf("Enter the element : ");  
                scanf("%d",&m);  
                addatbeg(m);  
                break;  
            case 3:
```

```

        printf("Enter the element : ");
        scanf("%d",&m);
        printf("Enter the position after which this element is
inserted : ");
        scanf("%d",&position);
        addafter(m,position);
        break;
    case 4:
        if(start==NULL)
        {
            printf("List is empty\n");
            continue;
        }
        printf("Enter the element for deletion : ");
        scanf("%d",&m);
        del(m);
        break;
    case 5:
        display();
        break;
    case 6:
        printf("Enter the element to be searched : ");
        scanf("%d",&m);
        search(m);
        break;
    case 7:
        exit();
    default:
        printf("Wrong choice\n");
    }/*End of switch*/
}/*End of while*/
}/*End of main()*/
}

create_list(int data)
{
    struct node *q,*tmp;
    tmp=malloc(sizeof(struct node));
    tmp->info=data;
    tmp->link=NULL;
    if(start==NULL) /*If list is empty*/
        start=tmp;
    else
    {
        /*Element inserted at the end*/
        q=start;
        while(q->link!=NULL)
            q=q->link;
        q->link=tmp;
    }
}/*End of create_list()*/

```

```
addatbeg(int data)
{
    struct node *tmp;
    tmp=malloc(sizeof(struct node));
    tmp->info=data;
    tmp->link=start;
    start = tmp;
/*End of addatbeg()*/
}

addafter(int data, int pos)

{
    struct node *tmp, *q;
    int i;
    q=start;
    for(i=0;i<pos-1;i++)
    {
        q=q->link;
        if(q==NULL)
        {
            printf("There are less than %d elements", pos);
            return;
        }
    /*End of for*/
    tmp=malloc(sizeof(struct node));
    tmp->link=q->link;
    tmp->info=data;
    q->link=tmp;
/*End of addafter()*/
}

del(int data)

{
    struct node *tmp,*q;
    if(start->info==data)
    {
        tmp=start;
        start=start->link; /*First element deleted*/
        free(tmp);
        return;
    }
    q=start;
    while(q->link->link!=NULL)
    {
        if(q->link->info==data)/*Element deleted in between*/
        {
            tmp=q->link;
            q->link=tmp->link;
            free(tmp);
            return;
        }
        q=q->link;
    }/*End of while */
}
```

```

if(q->link->info==data) /*Last element deleted*/
{
    tmp=q->link;
    free(tmp);
    q->link=NULL;
    return;
}
printf("Element %d not found\n",data);
}/*End of del()*/



display()
{
    struct node *q;
    if(start==NULL)
    {
        printf("List is empty\n");
        return;
    }
    q=start;
    printf("List is :\n");
    while(q!=NULL)
    {
        printf("%d ",q->info);
        q=q->link;
    }
    printf("\n");
}/*End of display() */



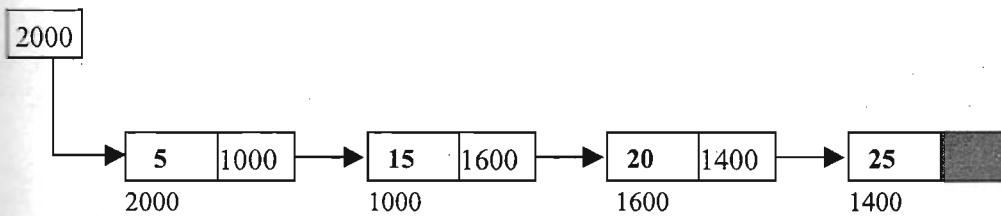
search(int data)
{
    struct node *ptr=start;
    int pos=1;
    while(ptr!=NULL)
    {
        if(ptr->info==data)
        {
            printf("Item %d found at position %d\n",data,pos);
            return;
        }
        ptr=ptr->link;
        pos++;
    }
    if(ptr==NULL)
        printf("Item %d not found in list\n",data);
}/*End of search()*/

```

10.15.10 Reversing A Linked List

Let us take a linked list-

start



Now we want to reverse this linked list. Reverse of this linked list will satisfy the following conditions-

First node will become the last node of linked list.

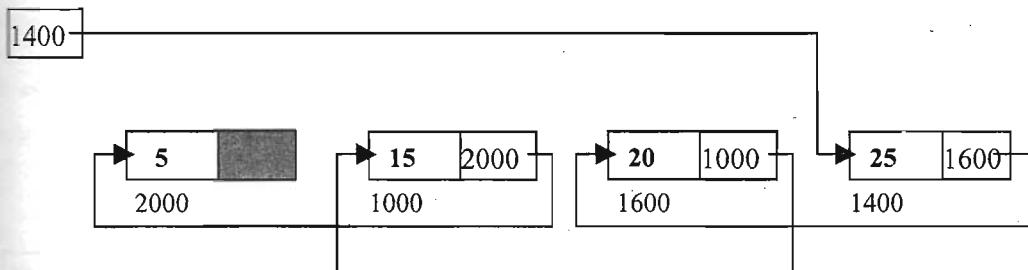
Last node will become the first node of linked list and now start will point to it.

Link of 2nd node will point to 1st node, link of 3rd node will point to second node and so on.

Link of last node will point to the previous node of last node in linked list.

Now reversed linked list will be as-

start



Creation Of reverse()

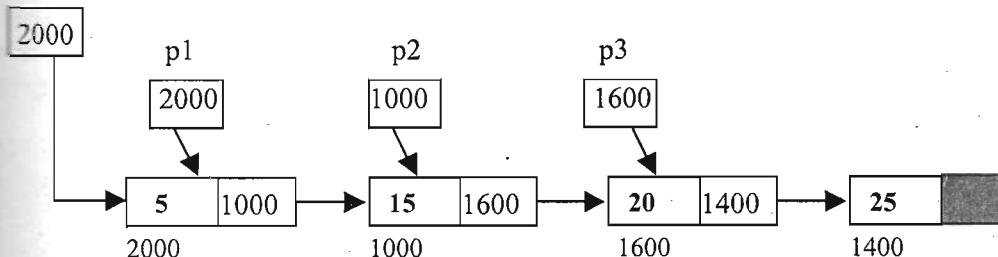
We will take three pointers p1, p2 and p3. Initially p1, p2 and p3 will point to first, second and third node of linked list.

p1 = start;

p2 = p1 -> link;

p3 = p2 -> link;

start



In reverse list first node will become the last node, so link part of first node should be NULL.

p1->link = NULL;

The link of second node should point to first node hence

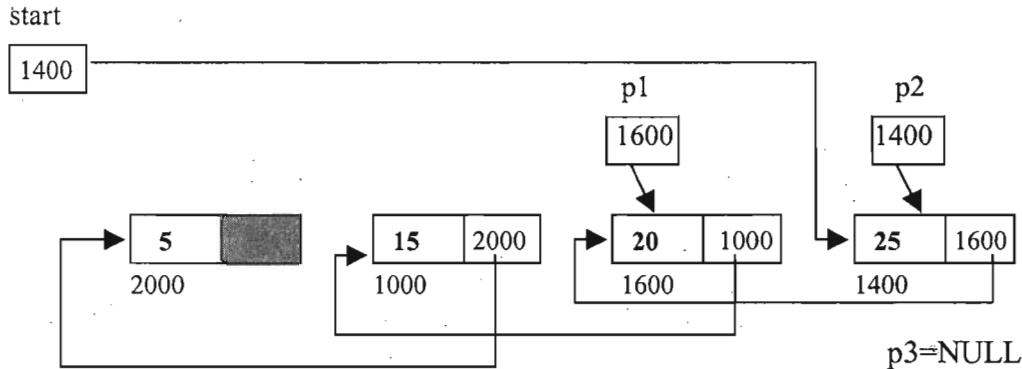
```
p2->link = p1;
```

Now we will traverse the linked list with p3 pointer and shift pointers p1 and p2 forward. We p1 to the link part of p2, so that link of each node will now point to its previous node.

```
p2->link=p1;
while(p3!=NULL)
{
    p1=p2;
    p2=p3;
    p3=p3->link;
    p2->link=p1;
}
```

When this loop will terminate p3 will be NULL, p2 will point to last node, and link of each node now point to its previous node. Now start should point to the last node of the linked list, which node of reversed linked list.

```
start = p2;
```



If the list contains only one element then there will be a problem in initializing p3, hence we this condition in the beginning-

```
if ( start->link == NULL )
    return;
```

Here is the function reverse that reverses a linked list.

```
reverse()
{
    struct node *p1,*p2,*p3;
    if(start->link==NULL) /*if only one element in the list*/
        return;
    p1=start;
    p2=p1->link;
    p3=p2->link;

    p1->link=NULL;
    p2->link=p1;
```

```

while(p3!=NULL)
{
    p1=p2;
    p2=p3;
    p3=p3->link;
    p2->link=p1;
}
start=p2;
/*End of reverse() */

```

10.16 union

Union is a derived data type like structure and it can also contain members of different data types. The syntax used for definition of a union, declaration of union variables and for accessing members is similar to that used in structures, but here keyword **union** is used instead of **struct**. The main difference between union and structure is in the way memory is allocated for the members. In a structure each member has its own memory location, whereas members of union share the same memory location. When a variable of type union is declared, compiler allocates sufficient memory to hold the largest member in the union. Since all members share the same memory location hence we can use only one member at a time. Thus union is used for saving memory. The concept of union is useful when it is not necessary to use all members of the union at a time.

The syntax of definition of a union is-

```

union union_name{
    datatype member1;
    datatype member2;
    .....
};

```

Like structure variables, the union variables can be declared along with the definition or separately. For example-

```

union union_name{
    datatype member1;
    datatype member2;
    .....
}variable_name;

```

This can also be declared as-

```
union union_name variable_name;
```

We can access the union members using the same syntax used for structures. If we have a union variable then the members can be accessed using dot(.) operator, and if we have a pointer to union then the members can be accessed using the arrow (->) operator.

```

/*P10.16 Program for accessing union members*/
#include<stdio.h>
main()
{
    union result{
        int marks;
        char grade;
        float per;
    }
}
```

```

        }res;
res.marks=90;
printf("Marks : %d\n",res.marks);
res.grade='A';
printf("Grade : %c\n",res.grade);
res.per=85.5;
printf("Percentage : %f\n",res.per);
}

```

Output :

Marks : 90
 Grade : A
 Percentage : 85.500000

Before the first printf, the value 90 is assigned to the union member marks, so other members grad and per contain garbage value. After first printf, the value 'A' is assigned to the union member grade. So now the other two members marks and per contain garbage value. Only one member of union can hold value at a time, don't try to use all the members simultaneously. So a union variable of type result can be treated as either an int variable or char variable or a float variable. It is the responsibility of the programmer to keep track of member that currently holds the value.

Union variables can also be initialized, but there is a limitation. We know that due to sharing of memory all the members can't hold values simultaneously. So during initialization also only one member can be given an initial value, and this privilege is given to the first member. Hence only the first member of a union can be given an initial value. The type of the initializer should match with the type of the first member. For example, we can initialize the above union variable as-

```
union result res ={78};
```

Now we'll take a program and compare the memory allocated for a union and structure variable

```

/* P10.17 Program to compare the memory allocated for a union and structure
variable*/
#include<stdio.h>
struct stag{
    char c;
    int i;
    float f;
};
union utag{
    char c;
    int i;
    float f;
};
main( )
{
    union utag uvar;
    struct stag svar;
    printf("Size of svar = %u\n",sizeof(svar));
    printf("Address of svar : %u\t",&svar);
    printf("Address of members : %u %u %u\n",&svar.c,&svar.i,&svar.f);
    printf("Size of uvar = %u\n",sizeof(uvar));
}

```

```
printf("Address of svar : %u\t",&svar);
printf("Address of members : %u %u %u\n",&svar.c,&svar.i,&svar.f);
```

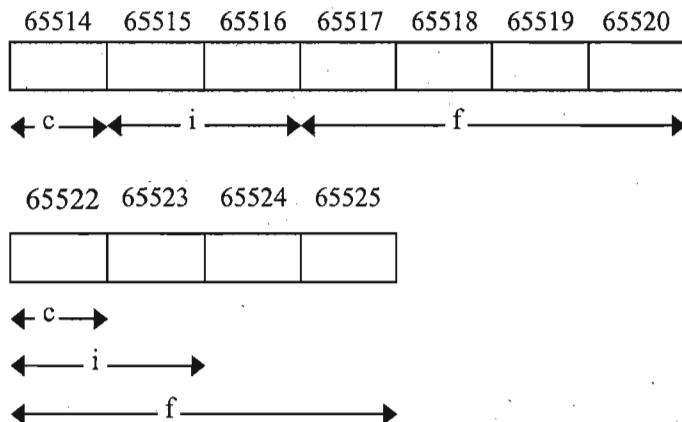
Output :

Size of svar = 7

Address of svar : 65514 Address of members : 65514 65515 65517

Size of uvar = 4

Address of uvar : 65522 Address of members : 65522 65522 65522



The addresses of members of a union are same while the addresses of members of a structure are different. The difference in the sizes of variables svar and uvar also indicates that union is very economical in the use of memory. Note that the sizes of both union and structure variables may be affected by padding as discussed previously.

A structure may be a member of union or a union may be a member of structure. For example-

union result

```
int marks;
char grade;

struct res

    char name[15];
    int age;
    union result performance;
} data;
```

Here data is the structure variable of type struct res. It has three members, an array of characters name, an integer age and a union member performance. Union will take only one value at a time, either an integer value marks or a character value grade. This can also be written as-

```
struct res

    char name[15];
    int age;
    union result
```

```

    {
        int marks;
        char grade;
    } performance;
} data;

```

If we want to use the member grade then we can write-

```
data.performance.grade
```

Similarly to use the member marks we can write-

```
data.performance.marks
```

Some other features of union are-

- (1) Arrays of unions can be declared.
- (2) Functions can take union variable as arguments and can return union variables.
- (3) Pointers to unions can be declared.
- (4) Unions can be nested.
- (5) Unions can be self referential like structures.

Besides saving memory space, unions can be helpful in other situations also. We can use unions declare an array that can hold values of different data type. The program below shows this-

```

/*P10.18 Program that uses an array of union to hold values of different
data types*/
#include<stdio.h>
union num{
    int i;
    long int l;
    float f;
    double d;
};
main( )
{
    union num arr[10]; /*Size of this array will be 10*sizeof(double)
arr[0].i=12;
arr[1].l=400000;
arr[2].f=11.12;
arr[3].d=23.34;
printf("%d\t",arr[0].i);
printf("%ld\t",arr[1].l);
printf("%f\t",arr[2].f);
printf("%f\n",arr[3].d);
}

```

Output :

12	400000	11.120000	23.340000
----	--------	-----------	-----------

Unions are also helpful in low level programming. We may manipulate the individual bytes in a data type using union. For example we can find whether a given machine's byte order is little-endian or big-endian. First we'll see what is a machine's byte order and then we'll write a program to determine the same.

The byte order of a machine specifies the order in which a multibyte data item is stored in memory.

on disk. There are two common byte orders.

Big-endian - Most significant byte is stored at the lowest address

Little-endian - Least significant byte is stored at the lowest address.

Intel family of processors use little-endian byte order, Motorola family of processors use big-endian byte order.

The binary representation of integer 5193 in 2 bytes is-

00010100 01001001

 M S Byte L S Byte

The following figure shows how this integer is stored in different byte orders.

	little-endian		big-endian
2000	0100 1001	LSB	2000 0001 0100
2001	0001 0100	MSB	2001 0100 1001

In little-endian, the least significant byte is stored at the lower address(2000), while the most significant byte is stored at the higher address(2001). In big-endian the MSB is stored at the lower address(2000), while the LSB is stored at the higher address(2001).

The following program determines the byte order of a machine-

```
*210.19 Program to determine the byte order of a machine*/
#include<stdio.h>
main()
{
    union{
        int x;
        char c[2];
    }var;
    var.x=1;
    if(var.c[0]==1)
        printf("Little endian\n");
    else
        printf("Big Endian\n");
}
```

To make our program portable we can take the size of character array to be equal to the size of int on the machine by specifying sizeof operator.

In the above example we had examined individual bytes of int data type, we may take any other data type also. For example in the next example we can examine the individual bytes of a variable of type struct stu-

```
struct student{
    char name[30];
    int age;
```

```

        int class;
    } ;
union{
    struct student stu;
    char c[sizeof(student)];
} var;

```

Now we can access the information stored in var.stu bytewise, this may be helpful in writing the struct variable var to any file byte by byte.

10.17 **typedef**

The type definition facility allows us to define a new name for an existing data type. The general syntax is-

```
typedef data_type new_name;
```

Here **typedef** is a keyword, **data_type** is any existing data type that may be a standard data type or a user defined type, **new_name** is an identifier, which is a new name for this data type. Note that we are not creating any new data type but we are only creating a new name for the existing data type. For example we can define a new name for int type by writing-

```
typedef int marks;
```

Now marks is a synonym for int and we can use marks instead of int anywhere in the program, for example-

```
marks sub1, sub2;
```

Here sub1, sub2 are actually int variables and are similar to any variable declared using int keyword. The above declaration is equivalent to-

```
int sub1, sub2;
```

Some more examples are-

```
typedef unsigned long int ulint;
```

```
typedef float real;
```

Here ulint is another name for type unsigned long int, and real is another name for float. The type declaration can be written wherever other declarations are allowed. We can give more than one name to a single data type using only one typedef statement. For example-

```
typedef int age, marks, units;
```

In the above typedef statement, we have defined three names for the data type int.

Since **typedef** is syntactically considered as a storage class, so we can't include a storage class in **typedef** statement. For example statements of these types are invalid-

```
typedef static char schar;
```

```
typedef extern int marks;
```

Now we'll see how **typedef** can be used to define new names for pointers, arrays, functions and structures.

1. Pointers

```
typedef float *fptr;
```

After this statement, fptr is a synonym for float * or pointer to float. Now consider this declaration-

```
fptr p, q, *r;
```

Here p and q are declared as pointer to float, while r is declared as a pointer to pointer to float.

2. Array

```
typedef int intarr[10];
```

After this statement intarr is another name for integer arrays of size 10. Now consider this declaration statement-

```
intarr a, b, c[15]; ( Equivalent to → int a[10], b[10], c[15][10]; )
```

Here a, b are declared as 1-D arrays of size 10, and c is declared as 2-D array of size 15x10.

3. Functions

```
typedef float funct(float, int);
```

Here funct is any function that takes two values, one float and one int and returns a float value.

Now consider this declaration statement-

```
funct add, sub, mul, div;
```

Here add, sub, mul, div are declared as functions that take a float and int value and return a float value.

The above statement is equivalent to the following declaration statements-

```
float add(float, int);
float sub(float, int);
float mul(float, int);
float div(float, int);
```

4. Structures

Similarly we can also use typedef for defining a new name for structures. Suppose we have this structure definition-

```
struct studentrec{
    char name[20];
    int marks;
};
```

Now whenever we want to use this structure we have to write struct studentrec. We can give a short and meaningful name to this structure by typedef.

```
typedef struct studentrec Student;
```

Now we can declare variable like this-

```
Student stu1, stu2; (Equivalent to → struct studentrec stu1, stu2; )
```

We can also combine typedef and structure definition. The syntax is as-

```
typedef struct tagname{
    datatype member1;
    .....
    .....
}newname;
```

Here tagname can be same as the newname. We can omit the tagname if the structure is not self referential.

```
typedef struct {
```

```

        char name[20];
        int age;
    }person;
person student,teacher,emp;

```

Here person is a new name for this structure and we have defined three structure variables, which have the format of the above definition.

In our linked list structure definition, we can write-

```

typedef struct node
{
    int item;
    struct node *link;
}node;

```

Now we can use node instead of struct node anywhere in our program-

```
node *start, *ptr;
```

Here new name is same as that of tagname. Here we can't omit the tagname since we have to define a pointer to the same structure inside the structure.

Advantages of using `typedef`

1. It makes our programs more readable and understandable since we can document our programs by giving meaningful and descriptive names for existing types.
2. In structures it is important since we can give a single name to the structure, we need not write `struct` keyword repeatedly.
3. It makes our programs more portable. When program is run on a different machine on which standard data types are represented by different number of bytes, only `typedef` statement has to be changed.

The `typedef` declarations may seem similar to `#define` directives, but they are different. The basic difference is that the `typedef` declarations are interpreted by the compiler while `#define` directives are processed by the preprocessor. In `#define` directive we can define an expansion for any text while in `typedef` we can only define new names for data types only. The following program shows the difference between `#define` directive and `typedef` declaration.

```

/*P10.20 Program to understand the difference between #define and typedef*
#include<stdio.h>
#define fp float *
main()
{
    fp p1,p2,p3;
    typedef float *fptr;
    fptr ptr1,ptr2,ptr3;
    printf("%u %u %u\n",sizeof(p1),sizeof(p2),sizeof(p3));
    printf("%u %u %u\n",sizeof(ptr1),sizeof(ptr2),sizeof(ptr3));
}

```

Output:

2 4 4

2 2 2

From the output we can see that p1 is declared as a pointer to float while p2 and p3 are declared as float variables. Actually the preprocessor expanded the declaration as-

```
float *p1, p2, p3;
```

So using #define we could not successfully define a new name for pointer to float type.

The variables ptr1, ptr2, ptr3 are all declared as pointers to float, so typedef successfully defined a new name for the type pointer to float.

Exercise

Assume that stdio.h is included in all programs.

```
(1) main()
{
    struct A{
        int marks;
        char grade;
    }A1;
    struct A B1;
    A1.marks=80;
    A1.grade='A';
    printf("Marks = %d\t",A1.marks );
    printf("Grade = %c\t",A1.grade );
    B1 = A1;
    printf("Marks = %d\t",B1.marks );
    printf("Grade = %c\n",B1.grade );
}

(2) main()
{
    struct rec{
        char *name;
        int age;
    }*ptr;
    char name1[10]="Somalika";
    ptr->name=name1;
    ptr->age=93;
    printf("%s\t",ptr->name);
    printf("%d\n",ptr->age);
}

(3) struct student { char name[20]; int age; };
main()
{
    struct student stu1={"Anita",10},stu2={"Anita",12};
    if(stu1==stu2)
        printf("Same\n");
    else
        printf("Not same\n");
}

(4) main()
```

```

{
    struct tag{
        int i;
        char c;
    };
    struct tag var={2,'s'};
    func(var);
}
func(struct tag v)
{
    printf("%d %c\n",v.i,v.c);
}

(5) main()
{
    struct tag{
        int i;
        char c;
    };
    struct tag var={2,'s'};
    func(var);
}
func(struct{int i; char c;} v )
{
    printf("%d %c\n",v.i,v.c);
}

(6) struct tag{ int i; char c;};
void func(struct tag);
main()
{
    struct tag var={12,'c'};
    func(var);
    printf("%d\n",var.i);
}
void func(struct tag var)
{
    var.i++;
}

(7) struct tag{ int i; char c;};
void func(struct tag *);
main()
{
    struct tag var={12,'c'};
    func(&var);
    printf("%d\n",var.i);
}
void func(struct tag *ptr)
{
    ptr->i++;
}

```

```
(8) #include<string.h>
main()
{
    union tag{
        char name[15];
        int age;
    }rec;
    strcpy(rec.name,"Somalika");
    rec.age=23;
    printf("Name = %s\n",rec.name);
}

(9) struct{
    char a[20];
    int b;
    union{
        double c;
        struct{
            char d[15];
            float e;
        }x;
    }y;
}z;
main()
{
    printf("%u %u %u\n",sizeof(z.y.x),sizeof(z.y),sizeof(z));
}

(10)main()
{
    typedef short int s_int;
    unsigned s_int var = 3;
    printf("%u",var);
}

(11)typedef struct tag{ int i; char c; }tag;
main()
{
    struct tag v1={1,'a'};
    tag v2={2,'b'};
    printf("%d %c %d %c\n",v1.i,v1.c,v2.i,v2.c);
}

(12)typedef struct { char name[20]; int age; }stu;
typedef struct { int data; node *link; }node;
main()
{
    stu *p=malloc(sizeof(stu));
    node *ptr=malloc(sizeof(node));
    p->age=30;
    ptr->data=3;
```

```

        printf("%d %d\n", p->age, ptr->data);
    }

```

Programming Exercise

1. Write a program to accept name, age and address of five persons and display the name of each person.

2. Write a program to accept name and arrival time of five trains and display the name with Rail time format.

(Note: For example 2PM is written as 14.00)

3. Write a program to accept 10 records with the structure-

```

struct{
    char *name;
    int *age;
    float salary;
}

```

Display the records before sorting and after sorting. Sorting is based on the primary key name and secondary key age.

4. Write a program to accept five records of employee. The structure is-

```

struct{
    char name[25];
    int age;
    int basic;
}

```

Calculate the total salary of the employees as-

$$\text{Total salary} = \text{Basic} + \text{DA} + \text{HRA}$$

$$\text{DA} = 10\% \text{ of basic}$$

$$\text{HRA} = 5\% \text{ of basic}$$

Display the name, age and total salary of the employees in descending order on the basis of total salary.

5. Write a program to concatenate one linked list at the end of another.

6. Write a program to remove first node of the list and insert it at the end.

7. Write a program to count the number of occurrences of an element in the list.

8. Find the largest and smallest element of a linked list, print total of all elements and find out average.

9. Write a program that maintains records of students using linked list. The structure of nodes in list would be-

```

struct node{
    char name[20];
    int rollno;
    int marks;
    struct node *link;
}

```

Now search a particular record based on roll number.

Answers

(1) Marks = 80 Grade = A Marks = 80 Grade = A

(2) Somalika 93

(3) Error, illegal structure operation. Since relational operators can't be used with structures. If we want to compare them then we'll have to make a function that compare all the members one by one.

(4) Error: undefined symbol i, and undefined symbol c. The structure definition is written inside main(), so it can't be accessed by function func(). It should be written before main() so that it is accessible to all functions.

(5) 2 s

(6) 12

(7) 13

(8) Garbage value will be printed, because currently the member age has been assigned a value.

(9) 19 19 41

Size of innermost structure is 19(15+4). The union contains two members of sizes 15 and 19, so the size of union is 19. The size of outermost structure is 41(20+2+19).

(10) This program will give errors, since it not valid to combine type specifiers(unsigned, long) with typedef names.

(11) 1 a 2 b

(12) The first typedef declaration is correct but the second one is not correct. Since in the second declaration the structure is self referential so we can't omit the structure tag.

Chapter 11

Files

The input and output operations that we have performed so far were done through screen and keyboard only. After termination of program, all the entered data is lost because primary memory is volatile. If the data has to be used later, then it becomes necessary to keep it in permanent storage device. C supports the concept of files through which data can be stored on the disk or secondary storage device. The stored data can be read whenever required. A file is a collection of related data placed on the disk.

The file handling in C can be broadly categorized in two types-

- High level(standard files or stream oriented files)
- Low level(system oriented files)

High level file handling is managed by library functions while low level file handling is managed by system calls. The high level file handling is commonly used since it is easier to manage and hides most of the details from the programmer. In this chapter we'll discuss about high level file handling only.

The header file stdio.h should be included in the program to make use of the I/O functions. We have already performed I/O with screen and keyboard using functions like scanf(), printf(), gets(), puts(), getchar() and putchar(). The advantage of using stream oriented file I/O is that the I/O in files is somewhat similar to the screen, keyboard I/O. For example we have functions like fscanf(), fprintf(), fgets(), fputs() which are equivalent to the functions that we have studied earlier.

11.1 Text And Binary Modes

There are two ways of storing data in files, binary format and text format. In text format, data is stored as lines of characters with each line terminated by a newline character('\n'). In binary format, data is stored on the disk in the same way as it is represented in the computer memory. Unix system does not make any distinction between text file and binary files.

Text files are in human readable form and they can be created and read using any text editor, while binary files are not in human readable form and they can be created and read only by specific programs written for them. The binary data stored in the file can't be read using a text editor.

The integer 1679 will take only 2 bytes in a binary file while it will occupy 4 bytes in a text file because in binary file it is stored as an integer while in text file it is stored as a sequence of 4 characters i.e. '1', '6', '7', '9'.

The hexadecimal value of 1679 is 0x068F, so in binary format it is represented by the two bytes 0x06 and 0x8F. In a text file, this number 1679 is represented by the bytes 0x31, 0x36, 0x37, 0x39(ASCII values).

Binary	-	0000	0110	1000	1111				
Text	-	0011	0001	0011	0110	0011	0111	0011	1001

Both text files and binary files keep record of the length of the file, and identify the end of file when this length is reached. In text files, there is one more way to detect the end of file. The character with ASCII value 26 is considered to be the end of file character in text files. All input functions stop reading from a text file when this character is encountered and return an end of file signal to the program. C does not insert this character in the file, it can be entered through the keyboard by Ctrl+Z(or Ctrl + D on some systems). In binary files no such character represents the end of file.

In text files newline is stored as a combination of carriage return '\r'(ASCII 13) and linefeed '\n'(ASCII 10) while in binary files newline is stored only as '\n'(ASCII 10).

In binary format, the data is stored in the same way as it is represented in memory so no conversions have to take place while transferring of data between memory and file. In text format some conversions have to take place while transferring data between memory and file. For example while writing to a text file newline("\n") has to be converted to a combination of '\r' and '\n' and while reading from a text file this combination ('\n' and '\r') is converted back to '\n'.

The input and output operations in binary files take less time as compared to that of text files because in binary files no conversions have to take place. However the data written using binary format is not very portable since the size of data types and byte order may be different on different machines. In text format, these problems do not arise so it is considered more portable.

11.2 Concept Of Buffer

Buffer is an area in memory where the data is temporarily stored before being written to the file. When we open a file, a buffer is automatically associated with its file pointer. Whatever data we send to the file is not immediately written to the file. First it is sent to the buffer and when the buffer is full then its contents are written to the file. When the file is closed, all the contents of buffer are automatically written to the file even if the buffer is not full. This is called flushing the buffer, we can also explicitly flush the buffer by a function fflush() described later.

The concept of buffer is used to increase efficiency. Had there been no buffer we would have to access the disk each time for writing even single byte of data. This would have taken lot of time because each time the disk is accessed, the read/write head has to be repositioned. When buffering is done, the data is collected in the buffer and data equal to the size of buffer is written to the file at a time, so the number of times disk is accessed decreases, which improves the efficiency.

The steps for file operations in C programming are as follows-

1. Open a file
2. Read the file or write data in the file
3. Close the file

The functions fopen() and fclose() are used for opening and closing the files respectively.

11.3 Opening a File

A file must be opened before any I/O operations can be performed on that file. The process of establishing a connection between the program and file is called opening the file.

A structure named FILE is defined in the file stdio.h that contains all information about the file like name, status, buffer size, current position, end of file status etc. All these details are hidden from the

programmer and the operating system takes care of all these things.

```
typedef struct{
    .....
    .....
    .....
} FILE;
```

A file pointer is a pointer to a structure of type FILE. Whenever a file is opened, a structure of type FILE is associated with it, and a file pointer that points to this structure identifies this file. The function fopen() is used to open a file.

Declaration:

```
FILE *fopen( const char *filename, const char *mode);
```

fopen() function takes two strings as arguments, the first one is the name of the file to be opened and the second one is the mode that decides which operations(read, write, append etc) are to be performed on the file. On success, fopen() returns a pointer of type FILE and on error it returns NULL. The return value of fopen() is assigned to a FILE pointer declared previously. For example-

```
FILE *fp1, *fp2;
fp1 = fopen ( "myfile.txt", "w" );
fp2 = fopen ( "yourfile.dat", "r" );
```

The name of a file is limited to FILENAME_MAX characters. After opening the file with fopen(), the name of file is not used in the program for any operation on it. Whenever we have to perform an operation on the file, we'll use the file pointer returned by fopen() function. So the name of file sometimes known as external name, while the file pointer associated with it is known as its internal name. The second argument represents the mode in which the file is to be opened. The possible values of mode are-

1. "w" (write)

If the file doesn't exist then this mode creates a new file for writing, and if the file already exists then the previous data is erased and the new data entered is written to the file.

2. "a" (append)

If the file doesn't exist then this mode creates a new file, and if the file already exists then the new data entered is appended at the end of existing data. In this mode, the data existing in the file is erased as in "w" mode.

3. "r" (read)

This mode is used for opening an existing file for reading purpose only. The file to be opened must exist and the previous data of file is not erased.

4. "w+" (write+read)

This mode is same as "w" mode but in this mode we can also read and modify the data. If the file doesn't exist then a new file is created and if the file exists then previous data is erased.

5. "r+" (read+write)

This mode is same as "r" mode but in this mode we can also write and modify existing data. The file to be opened must exist and the previous data of file is not erased. Since we can add new data and modify existing data so this mode is also called update mode.

"a+" (append+read)

This mode is same as the "a" mode but in this mode we can also read the data stored in the file. If the file doesn't exist, a new file is created and if the file already exists then new data is appended at the end of existing data. We cannot modify existing data in this mode.

To open a file in binary mode we can append 'b' to the mode, and to open the file in text mode 't' can be appended to the mode. But since text mode is the default mode, 't' is generally omitted while opening files in text mode. For example-

"wb"	Binary file opened in write mode
"ab+" or "a+b"	Binary file opened in append mode
"rt+" or "r+t"	Text file opened in update mode
"w"	Text file opened in write mode

III.3.1 Errors in Opening Files

If an error occurs in opening the file, then fopen() returns NULL. So we can check for any errors in opening by checking the return value of fopen().

```
FILE *fp;
fp=fopen("text.dat", "w");
if(fp==NULL)

printf("Error in opening file");
exit(1);
```

Errors in opening a file may occur due to various reasons, for example-

- 1. If we try to open a file in read or update mode, and the file doesn't exist or we do not have read permission on that file.
- 2. If we try to create a file but there is no space on the disk or we don't have write permission.
- 3. If we try to create a file that already exists and we don't have permission to delete that file.
- 4. Operating system limits the number of files that can be opened at a time and we are trying to open more files than that number.

We can give full pathname to open a file. Suppose we want to open a file in DOS whose path is "E:\booksdir\names.dat", then we'll have to write as-

```
fp = fopen("E:\\booksdir\\\\names.dat", "r");
```

Here we have used double backslash because single backslash inside string is considered as an escape character, '\b' and '\n' will be regarded as escape sequences if we use single backslash. In Unix, a single forward slash can be used.

Never give the mode in single quotes, since it is a string not a character constant.

```
fp = fopen("file.dat", 'w'); /*Error*/
```

III.4 Closing a File

The file that was opened using fopen() function must be closed when no more operations are to be performed on it. After closing the file, connection between file and program is broken.

Declaration: int fclose(FILE *fptr);

On closing the file, all the buffers associated with it are flushed i.e. all the data that is in the buffer

is written to the file. The buffers allocated by the system for the file are freed after the file is closed so that these buffers can be available for other files. Some systems have a limitation on the number of files that can be opened at a time, so we should close the files that are not currently in use so we can open other files.

Although all the files are closed automatically when the program terminates normally, but sometimes it may be necessary to close the file by `fclose()` e.g. when we have to reopen the file in some mode or when we exceed the number of opened files permitted by the system. Moreover it is a good practice to close files explicitly by `fclose()` when no more operations are to be performed on them because it becomes clear to the reader of the program that the file has no use now.

`fclose()` returns EOF on error and 0 on success(EOF is a constant defined in stdio.h and its value is -1). An error in `fclose()` may occur when there is not sufficient space on the disk or when the disk has been taken out of the drive.

If more than one files are opened, then we can close all the files by calling `fclose()` for each

```
fclose(fp1);
fclose(fp2);
```

```
....  
....
```

We can also close multiple files by calling a single function `fcloseall()`. It closes all the opened files.

Declaration: int `fcloseall(void);`

On error, `fcloseall()` returns EOF and on success it returns the number of files closed. We can use it as-

```
n=fcloseall();
if(n==EOF)
    printf("Could not close all open files\n");
else
    printf("%d files successfully closed\n",n);
```

11.5 End of File

The file reading functions need to know the end of file so that they can stop reading. When the end of file is reached, the operating system sends an end-of-file signal to the program. When the program receives this signal, the file reading functions return EOF, which is a constant defined in the file stdio.h and its value is -1. EOF is an integer value, so make sure that the return value of the function is assigned to an integer variable. Note that the value EOF is not present at the end of the file, it is returned by the file reading functions when end of file is reached.

11.6 Structure of a General File Program

```
main()
{
    FILE *fp;
    fp=fopen("filename", "mode");
    .....
    .....
    .....
    fclose ( fp );
}/*End of main */
```

7 Predefined File Pointers

predefined constant file pointers are opened automatically when the program is executed.

File pointer	Device
stdin	Standard input device(Keyboard))
stdout	Standard output device(Screen)
stderr	Standard error output device (Screen)

The functions used for file I/O are

Character I/O	- fgetc(), fputc(), getc(), putc()
String I/O	- fgets(), fputs()
Integer I/O	- getw(), putw()
Formatted I/O	- fscanf(), sprintf()
Record I/O	- fread(), fwrite()

We will discuss all these functions one by one in detail.

8 Character I/O

8.1 fputc ()

Declaration: int fputc(int c, FILE *fptr);

This function writes a character to the specified file at the current file position and then increments the file position pointer. On success it returns an integer representing the character written, and on error returns EOF.

```
/* Program to understand the use of fputc() function*/
#include<stdio.h>
main()
{
    FILE *fptr;
    int ch;
    if((fptr=fopen("myfile.txt", "w"))==NULL)
    {
        printf("File does not exist\n");
        exit();
    }
    else
    {
        printf("Enter text :\n");
        /*Press Ctrl+z in DOS and Ctrl+d in Unix to stop reading characters*/
        while((ch=getchar())!=EOF)
            fputc(ch,fptr);
    }
    fclose(fptr);
}
```

Output

Enter text :

The woods are lovely, dark and deep
 But I have miles to go before I sleep.
 ^Z

After the execution of this program, this text along with the ^Z character will be written to the myfile.txt.

11.8.2 fgetc()

Declaration: int fgetc(FILE *fptr);

This function reads a single character from a given file and increments the file pointer position. On success it returns the character after converting it to an int without sign extension. On end of file or error it returns EOF. In the next program we'll read the file myfile.txt that we have created in the previous program.

```
/*P11.2 Program to understand the use of fgetc()*/
#include<stdio.h>
main()
{
    FILE *p;
    char ch;
    if((p=fopen("myfile.txt", "r"))==NULL)
        printf("This file doesn't exist\n");
    else
    {
        while((ch=fgetc(p))!=EOF)
            printf("%c", ch);
    }
    fclose(p);
}
```

Output

The woods are lovely, dark and deep
 But I have miles to go before I sleep.

The while loop that we have written in the program is equivalent to this code-

```
ch=fgetc(p);
while(ch!=EOF)
{
    printf("%c", ch);
    fgetc(p);
}
```

The value returned by fputc() and fgetc() is not of type char but is of type int. This is because functions return an integer value EOF(-1) on end of file or error. The variable ch that is used to store the character read from the file, is also declared to be of int type for this reason only.

11.8.3 getc() and putc()

The operations of getc() and putc() are exactly similar to that of fgetc() and fputc(), the only difference is that the former two are defined as macros while the latter two are functions.

11.9 Integer I/O

11.9.1 putw()

Declaration: int putw(int value, FILE *fptr)

This function writes an integer value to the file pointed to by fptr. It returns the integer written to the file on success, and EOF on error.

```
/*P11.3 Program to understand the use of putw() function*/
#include<stdio.h>
main()
{
    FILE *fptr;
    int value;
    fptr=fopen("num.dat", "wb");
    for(value=1;value<=30;value++)
        putw(value,fptr);
    fclose(fptr);
}
```

This program will write integers from 1 to 30 into the file "num.dat".

11.9.2 getw()

Declaration: int getw(FILE *fptr);

This function returns the integer value from the file associated with fptr. It returns the next integer from the input file on success, and EOF on error or end of file.

```
/*P11.4 Program to understand the use of getw() function*/
#include<stdio.h>
main()
{
    FILE *fptr;
    int value;
    fptr=fopen("num.dat", "rb");

    while((value=getw(fptr))!=EOF)
        printf("%d\t",value);
    fclose(fptr);
}
```

This program will read and print integers from the file "num.dat" which was created earlier.

If getw() is used with text files then it will stop reading if integer 26 is present in the file because in text files end of file is denoted by ASCII 26 which is also a valid integer value. So getw() should not be used with text files.

The value of EOF is -1 which is a valid integer value, so this program will work correctly if -1 is not present in file, if -1 exists in the file then getw() will stop reading and all the values beyond -1 will be left unread. So we should use feof() to check end of file and ferror() to check error. These functions are discussed later in this chapter.

11.9 Integer I/O

11.9.1 putw()

Declaration: int putw(int value, FILE *fptr)

This function writes an integer value to the file pointed to by fptr. It returns the integer written to the file on success, and EOF on error.

```
/*P11.3 Program to understand the use of putw() function*/
#include<stdio.h>
main()
{
    FILE *fptr;
    int value;
    fptr=fopen("num.dat", "wb");
    for(value=1;value<=30;value++)
        putw(value,fptr);
    fclose(fptr);
}
```

This program will write integers from 1 to 30 into the file "num.dat".

11.9.2 getw()

Declaration: int getw(FILE *fptr);

This function returns the integer value from the file associated with fptr. It returns the next integer from the input file on success, and EOF on error or end of file.

```
/*P11.4 Program to understand the use of getw() function*/
#include<stdio.h>
main()
{
    FILE *fptr;
    int value;
    fptr=fopen("num.dat", "rb");

    while((value=getw(fptr))!=EOF)
        printf("%d\t",value);
    fclose(fptr);
}
```

This program will read and print integers from the file "num.dat" which was created earlier.

If getw() is used with text files then it will stop reading if integer 26 is present in the file because in text files end of file is denoted by ASCII 26 which is also a valid integer value. So getw() should not be used with text files.

The value of EOF is -1 which is a valid integer value, so this program will work correctly if -1 is not present in file, if -1 exists in the file then getw() will stop reading and all the values beyond -1 will be left unread. So we should use feof() to check end of file and ferror() to check error. These functions are discussed later in this chapter.

11.10 String I/O

11.10.1 fputs()

Declaration: int fputs(const char *str, FILE *fptr);

This function writes the null terminated string pointed to by str to a file. The null character that marks the end of string is not written to the file. On success it returns the last character written and on error it returns EOF.

```
/*P11.5 Program to understand the use of fputs()*/
#include<stdio.h>
main()
{
    FILE *fptr;
    char str[80];
    fptr=fopen("test.txt", "w");
    printf("Enter the text\n");
    printf("To stop entering, press Ctrl+d in Unix and Ctrl+z in Dos\n");
    while(gets(str)!=NULL)
        fputs(str,fptr);
    fclose(fptr);
}
```

Suppose we enter this text after running the program-

Yesterday is history

Tomorrow is mystery

Today is a gift

That's why it is called Present

^Z

When the first line of text is entered and Enter key is pressed, the function gets() converts the newline character to the null character and the array str contains "Yesterday is history"(20 characters + 1 null character). Now str is written to the file test.txt using fputs(). The null character is not written to the file, so only 20 characters are written.

In previous chapters we had studied about the function puts() that prints the string on the screen. The difference between these two functions is that puts() translates null character to a newline, but fputs() does not. fputs() will write a newline character to the file only if it is contained in the string. So after the first line of text newline character is not entered in the file. Immediately after the first line of text, the second one is written. In the next program we'll see how to read this file using fgets().

11.10.2 fgets()

Declaration: char *fgets(char *str, int n, FILE *fptr);

This function is used to read characters from a file and these characters are stored in the string pointed to by str. It reads at most n-1 characters from the file where n is the second argument. fptr is a file pointer which points to the file from which characters are read. This function returns the string pointed to by str on success, and on error or end of file it returns NULL.

This function reads characters from the file until either a newline or an end of file is encountered till n-1 characters have been read. It appends a null character ('\\0') after the last character read to terminate the string. The following program reads characters from the file test.txt that we had created.

```

FILE *fptr;
char str[80];
fptr=fopen("test.txt","r");
while(fgets(str,80,fptr)!=NULL)
    puts(str);
fclose(fptr);

```

Output:

Yesterday is history Tomorrow is mystery Today is a gift That's why it is called P

Suppose we call the fgets() function with second argument as 20, then the output would be-

Yesterday is histor

Tomorrow is Myster

Today is a gift Tha

why it is calle

Present

When fgets() was called with second argument as 80, then it read 79 characters from the file and stored them in array str and appended a null character. Now str was passed to puts(), so the string was displayed on the screen. The function puts() converts the null character of string to a newline, after displaying the string a newline is displayed. In the second iteration of the loop, the function encounters end of file after reading 6 characters only so it stops reading, stores these characters str, appends a null character and displays str on the screen using puts(). When fgets() was called with second argument as 20 , then it read 19 characters from the file.

gets() reads characters from the standard input while fgets() reads from a file. The difference between gets() and fgets() is that fgets() does not replace the newline character read by the null character, while gets() does. If fgets() reads a newline, then both newline and null character will be present in the final string.

gets() it may be possible that input is more than the size of array, since C does not check for array bounds, so an overflow may occur but in fgets() we can limit the size of input with the help of second argument.

11.1 Formatted I/O

We have studied about functions which can output and input characters, integers or strings from files. The next two functions that we are going to study now, can input and output a combination of all of these in a formatted way. Formatting in files is generally used when there is a need to display data on terminal or print data in some format.

11.1.1 fprintf()

Declaration: `fprintf (FILE *fptr, const char *format [, argument, ...]);`

This function is same as the printf() function but it writes formatted data into the file instead of standard output(screen). This function has same parameters as in printf() but it has one additional parameter which is a pointer of FILE type, that points to the file to which the output is to be written. It returns the number of characters output to the file on success, and EOF on error.

```
/*P11.7 Program to understand the use of fprintf()*/
#include<stdio.h>
main()
{
    FILE *fp;
    char name[10];
    int age;
    fp=fopen("rec.dat", "w");
    printf("Enter your name and age : ");
    scanf("%s%d", name, &age);
    fprintf(fp, "My name is %s and age is %d", name, age);
    fclose(fp);
}

/*P11.8 Program to understand the use of fprintf()*/
#include<stdio.h>
struct student
{
    char name[20];
    float marks;
}stu;
main()
{
    FILE *fp;
    int i,n;
    fp=fopen("students.dat", "w");
    printf("Enter number of records : ");
    scanf("%d", &n);
    for(i=1;i<=n;i++)
    {
        printf("Enter name and marks : ");
        scanf("%s%f", stu.name, &stu.marks);
        fprintf(fp, "%s %f", stu.name, stu.marks);
    }
}
```

11.11.2 fscanf()

Declaration: fscanf (FILE *fptr, const char *format [, address, ...]);

This function is similar to the scanf() function but it reads data from file instead of standard input so it has one more parameter which is a pointer of FILE type and it points to the file from which data will be read. It returns the number of arguments that were assigned some values on success, and EOF at the end of file.

In the next program we'll read the file students.dat that we have created using fprintf() in P11.8. Note that the format string used in fscanf() should be same as the format string used in fprintf() while writing to the file.

```
/* P11.9 Program to understand the use of fscanf() */
#include<stdio.h>
struct student
{
    char name[20];
    float marks;
}stu;
main()
{
    FILE *fopen(), *fp;
    fp=fopen("students.dat", "r");
    printf("NAME\tMARKS\n");
    while(fscanf(fp, "%s%f", stu.name, &stu.marks)!=EOF)
        printf("%s\t%f\n", stu.name, stu.marks);
    fclose(fp);
}
```

We had mentioned earlier that the file pointers stdout and stdin are automatically opened. If we use these file pointers in the functions fprintf() and fscanf(), then these function calls become equivalent to printf() and scanf().

If we replace the file pointer fp by stdout then-

fprintf(stdout, "My age is %s" , age); is equivalent to printf("My age is %d", age);

If we replace the file pointer fp by stdin then-

fscanf(stdin, "%s%d", name, &age); is equivalent to scanf("%s%d", name, &age);

11.12 Block Read / Write

It is useful to store blocks of data into file rather than individual elements. Each block has some fixed size, it may be a structure or an array. It is easy to read the entire block from file or write the entire block to the file. There are two useful functions for this purpose- fread() and fwrite(). Although we can read or write any type of data varying from a single character to arrays and structures through these functions, these are mainly used to read and write structures and arrays.

For using these functions, the file is generally opened in binary mode(e.g. "wb", "rb").

11.12.1 fwrite()

This function is used for writing an entire block to a given file.

Declaration: size_t fwrite(const void *ptr, size_t size, size_t n, FILE *fptr);

size_t is defined in stdio.h as-

```
typedef unsigned int size_t;
```

ptr is a pointer which points to the block of memory that contains the information to be written to the file, size denotes the length of each item in bytes, n is the number of items to be written to the file, and fptr is a FILE pointer which points to the file to which the data is written.

If successful, fwrite() will write n items or total (n * size) bytes to the file and will return n. On error or end of file it will return a number less than n.

To write a single float value contained in variable fval to the file

```
fwrite( &fval, sizeof(float), 1, fp);
```

To write an array of integers arr[10] to the file.

```
fwrite( arr, sizeof(arr), 1, fp);
```

To write only first 5 elements from the above array to the file

```
fwrite( arr, sizeof(int), 5, fp);
```

Note that here in third argument we'll send size of integer, because here the items that we are writing are integers not array.

To write a structure variable which is defined as

```
struct record{
```

```
    char name[20];
```

```
    int roll;
```

```
    float marks;
```

```
}student;
```

```
fwrite(&student,sizeof(student),1,fp);
```

This will write a single structure variable to the file.

To write an array of structures

Suppose stu_arr[200] is an array of structure defined above-

```
fwrite( stu_arr, sizeof(stu_arr), 1, fp);
```

Here it will write all the 200 elements of array stu_arr to the file.

To write only a part of an array of structure

Suppose in the above array stu[200] we have entered records of only 100 students then it is no use writing whole array to the file(garbage value will be written). We can write 100 records as-

```
fwrite(stu_arr, sizeof(struct record), 100, fp);
```

We have used sizeof operator instead of sending the size directly, so that our program becomes portable because the size of data types may vary on different computers. Moreover if new elements are added to our structure, we need not recalculate and change the size in our program.

```
/*P11.10 Program to understand the use of fwrite()*/
#include<stdio.h>
struct record
{
    char name[20];
    int roll;
    float marks;
}student;
main()
{
    int i,n;
    FILE *fp;
    fp=fopen("stu.dat", "wb");
    if(fp==NULL)
```

```

{
    printf("Error in opening file\n");
    exit(1);
}
printf("Enter number of records : ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
    printf("Enter name : ");
    scanf("%s",student.name);
    printf("Enter roll no : ");
    scanf("%d",&student.roll);
    printf("Enter marks : ");
    scanf("%f",&student.marks);
    fwrite(&student,sizeof(student),1,fp);
}
fclose(fp);
}

```

11.12.2 fread()

This function is used to read an entire block from a given file.

Declaration: size_t fread(void *ptr, size_t size, size_t n, FILE *fptr);

Here ptr is a pointer which points to the block of memory which receives the data read from the file, size is the length of each item in bytes, n is the number of items to be read from the file and fptr is a file pointer which points to the file from which data is read.

On success it reads n items from the file and returns n, if error or end of file occurs then it returns a value less than n. We can use feof() and ferror() to check these conditions.

To read a single float value from the file and store it in variable fval.

```
fread( &fval, sizeof(float), 1, fp);
```

To read array of integers from file and store them in array arr[10].

```
fread( arr, sizeof(arr), 1, fp);
```

To read 5 integers from file and store them in first five elements of array arr[10]

```
fread( arr, sizeof(int), 5, fp);
```

To read a structure variable that is defined as-

```
struct record{
    char name[20];
    int roll;
    float marks;
} student;
fread(&student,sizeof(student),1,fp);
```

This will read a single structure variable from the file and store it in variable student.

To read an array of structures

Suppose stu_arr[200] is an array of structure defined above.

```
    fread( stu_arr, sizeof(stu_arr), 1, fp);
```

Here it will read an array of structures from file and store it in stu_arr.

To read 100 records from file and store them in first 100 structures of stu_arr[200].

```
    fread( stu_arr, sizeof(struct record), 100, fp);
```

The following program reads the file stu.dat created in P11.10.

```
/*P11.11 Program to understand the use of fread()*/
#include<stdio.h>
struct record
{
    char name[20];
    int roll;
    float marks;
}student;
main()
{
    FILE *fp;
    fp=fopen("stu.dat", "rb");
    if(fp==NULL)
    {
        printf("Error in opening file\n");
        exit(1);
    }
    printf("\nNAME\tROLLNO\tMARKS\n");
    while(fread(&student,sizeof(student),1,fp)==1)
    {
        printf("%s\t",student.name);
        printf("%d\t",student.roll);
        printf("%f\n",student.marks);
    }
    fclose(fp);
}
```

The fread() function returns the number of records successfully read, so it will return 1 till there are records in the file and will return a number less than 1 when there will be no records in the file. So we have used this condition in our while loop to stop reading when end of file is reached.

11.13 Random Access To File

We can access the data stored in the file in two ways, sequentially or randomly. So far we have used only sequential access in our programs. For example if we want to access the forty-fourth record the first forty-three records should be read sequentially to reach the forty-fourth record. In random access data can be accessed and processed randomly i.e. in this case the forty-fourth record can be accessed directly. There is no need to read each record sequentially, if we want to access a particular record. Random access takes less time than the sequential access.

C supports these functions for random access file processing-

-fseek()

```
-fseek()
-rewind()
```

Before studying these functions it is necessary to understand the concept of file position pointer. File position pointer points to a particular byte in the file and all read and write operations on the file take place at this byte. This pointer automatically moves forward when a read or write operation takes place. To access the data randomly, we'll have to take control of this position pointer.

III.13.1 fseek()

This function is used for setting the file position pointer at the specified byte.

Declaration: int fseek(FILE *fp, long displacement, int origin);

Here fp is a file pointer, displacement is a long integer which can be positive or negative and it denotes the number of bytes which are skipped backward (if negative) or forward (if positive) from the position specified in the third argument. This argument is declared as long integer so that it is possible to move in large files.

The third argument named origin is the position relative to which the displacement takes place. It can take one of these three values.

Constant	Value	Position
SEEK_SET	0	Beginning of file
SEEK_CURRENT	1	Current position
SEEK_END	2	End of file

These three constants are defined in stdio.h so we can either use these names or their values.

Some examples of usage of fseek() function are-

1. fseek (p, 10L , 0);

Origin is 0, which means that displacement will be relative to beginning of file so position pointer is skipped 10 bytes forward from the beginning of the file. Since the second argument is a long integer, so L is attached with it.

2. fseek (p, 5L , 1);

Position pointer is skipped 5 bytes forward from the current position.

3. fseek (p, 8L , SEEK_SET);

Position pointer is skipped 8 bytes forward from the beginning of file.

4. fseek (p , -5L , 1);

Position pointer is skipped 5 bytes backward from the current position.

5. fseek (p, 0L , 2);

Origin is 2 which represents end of file and displacement is 0 which means that 0 bytes are skipped from end of file. So after this statement position pointer will point to the end of file.

6. fseek(p, -6L, SEEK_END);

Position pointer is skipped 6 bytes backward from the end of file.

7. fseek (p, 0L , 0);

This means 0 bytes are skipped from the beginning of file. After this statement position pointer points to the beginning of file.

On success fseek() returns the value 0, and on failure it returns a non-zero value. Consider the file stu.dat that we had created earlier in program P11.10. In the next program we'll be able to read a particular number of record randomly from the file with the help of fseek() function.

```
/*P11.12 Program to understand the use of fseek()*/
#include<stdio.h>
struct record
{
    char name[20];
    int roll;
    float marks;
}student;
main()
{
    int n;
    FILE *fp;
    fp=fopen("stu.dat", "rb");
    if(fp==NULL)
    {
        printf("Error in opening file\n");
        exit(1);
    }
    printf("Enter the record number to be read : ");
    scanf("%d",&n);
    fseek(fp, (n-1)*sizeof(student), 0); /*skip n-1 records*/
    fread(&student,sizeof(student),1,fp); /*Read the nth record*/
    printf("%s\t",student.name);
    printf("%d\t",student.roll);
    printf("%f\n",student.marks);
    fclose(fp);
}
```

Suppose we want to read the 5th record from the file, i.e. n=5. We'll skip first 4 records and place our position pointer at the beginning of the 5th record, so now fread() will read the 5th record. In general if we want to read the nth record we'll skip n-1 records with fseek() and place the position pointer at the beginning of the nth record.

11.13.2 ftell()

Declaration: long ftell(FILE *fptr);

This function returns the current position of the file position pointer. The value is counted from beginning of the file.

```
/* P11.13 Program to understand the use of ftell()*/
#include<stdio.h>
struct record
{
    char name[20];
    int roll;
    float marks;
}student;
```

```

main()

FILE *fp;
fp=fopen("stu.dat", "rb");
if(fp==NULL)
{
    printf("Error in opening file\n");
    exit(1);
}
printf("Position pointer in the beginning -> %ld\n", ftell(fp));
while(fread(&student, sizeof(student), 1, fp)==1)
{
    printf("Position pointer -> %ld\n", ftell(fp));
    printf("%s\t", student.name);
    printf("%d\t", student.roll);
    printf("%f\n", student.marks);
}
printf("Size of file in bytes is %ld\n", ftell(fp));
fclose(fp);

```

On error ftell() returns -1L and sets errno to a positive value.

11.13.3 rewind()

Declaration: rewind(FILE *fptr);

This function is used to move the file position pointer to the beginning of the file. Here fptr is a pointer of FILE type. The function rewind() is useful when we open a file for update.

Using rewind(fptr) is equivalent to fseek(fptr, 0L, 0).

```

/* P11.14 Program to understand the use of rewind()*/
#include<stdio.h>
main()
{
    FILE *fp;
    fp=fopen("stu.dat", "rb+");
    if(fp==NULL)
    {
        printf("Error in opening file\n");
        exit(1);
    }
    printf("Position pointer ->%ld\n", ftell(fp));
    fseek(fp, 0, 2);
    printf("Position pointer ->%ld\n", ftell(fp));
    rewind(fp);
    printf("Position pointer ->%ld\n", ftell(fp));
    fclose(fp);
}

```

Now we'll write some programs to perform various operations on the file stu.dat.

P11.15 Program to append records to a file/

```

#include<stdio.h>
main()
{
    struct record
    {
        char name[20];
        int roll;
        float marks;
    }student;
    FILE *fp;
    int choice=1;
    fp=fopen("stu.dat", "ab"); /*opened in append mode */
    if(fp==NULL)
    {
        printf("Error in opening file\n");
        exit(1);
    }
    while(choice==1)
    {
        printf("Enter name : ");
        scanf("%s",student.name);
        printf("Enter roll no : ");
        scanf("%d",&student.roll);
        printf("Enter marks : ");
        scanf("%f",&student.marks);
        fwrite(&student,sizeof(student),1,fp);
        printf("Want to enter more?(1 for yes/0 for no) : ");
        scanf("%d", &choice);
    }
    fclose(fp);
}

/*P11.16 Program to read records from a file and calculate grade of each student and display it

grade = A if marks >= 80
       = B if marks >= 60 and < 80
       = C if marks < 60
*/
#include<stdio.h>
main()
{
    struct record
    {
        char name[20];
        int roll;
        float marks;
    }student;
    FILE *fp;
    char grade;
    fp=fopen("stu.dat", "rb"); /*opened in read mode*/

```

```

if(fp==NULL)
{
    printf("Error in opening file\n");
    exit(1);
}
printf("\nNAME\t\tMARKS\t\tGRADE\n\n");
while(fread(&student,sizeof(student),1,fp)==1)
{
    printf("%s\t\t",student.name);
    printf("%2.2f\t\t",student.marks);
    if(student.marks>=80)
        printf("A\n");
    else if(student.marks>=60)
        printf("B\n");
    else
        printf("C\n");
}
fclose(fp);

```

```

*11.17 Program to modify records in a file*/
#include<stdio.h>
main()
{
    struct record
    {
        char name[20];
        int roll;
        float marks;
    }student;
    FILE *fp;
    char name[20];
    long size(sizeof(student));
    unsigned flag=0;
    fp=fopen("stu.dat","rb+");
    if(fp==NULL)
    {
        printf("Error in opening file\n");
        exit(1);
    }
    printf("Enter name of student whose record is to be modified : ");
    scanf("%s",name);

    while(fread(&student,sizeof(student),1,fp)==1)
        if(strcmp(student.name,name)==0)
        {
            printf("Enter new data -->\n");
            printf("Enter name : ");
            scanf("%s",student.name);
            printf("Enter roll no : ");
            scanf("%d",&student.roll);

```

```

        printf("Enter marks : ");
        scanf("%f",&student.marks);
        fseek(fp,-size,1);
        fwrite(&student,sizeof(student),1,fp);
        flag=1;
        break;
    }
    if(flag==0)
        printf("Name not found in file\n");
    else
        printf("Record Modified.....\n");
    fclose(fp);
}

```

For modifying records we have opened the file stu.dat in update(rb+) mode. We'll read all the records one by one with the help of fread(), and as soon as the name of the student matches with the name we've entered for modification, we'll enter new data for the student, and write that data to the file. Before fwrite() we have used fseek() to skip one record backwards because while reading we've reached the end of the record which is to be modified, i.e. position pointer is at the beginning of next record so if we write new data without using fseek() then it will be written over the next record. After modifying the data we'll stop reading and come out of the while loop with the help of break statement.

```

/*P11.18 Program to delete a record from the file*/
#include<stdio.h>
main()
{
    struct record
    {
        char name[20];
        int roll;
        float marks;
    }student;
    FILE *fp, *fptmp;
    char name[20];
    unsigned flag=0;
    fp=fopen("stu.dat","rb");
    if(fp==NULL)
    {
        printf("Error in opening file\n");
        exit(1);
    }
    printf("Enter the name to be deleted : ");
    scanf("%s",name);
    fptmp=fopen("tempfile.dat","wb");
    while(fread(&student,sizeof(student),1,fp)==1)
    {
        if(strcmp(name,student.name)!=0)
            fwrite(&student,sizeof(student),1,fptmp);
        else
            flag=1;
    }
}

```

```
fclose(fp);
fclose(fptmp);
remove("stu.dat");
rename("tempfile.dat", "stu.dat");
if(flag==0)
    printf("Name not found in file\n");
else
    printf("Record deleted.....\n");
}
```

For deleting a record from stu.dat we'll make a temporary file tempfile.dat and copy all the records to that file, except the record to be deleted, and then we'll delete the original stu.dat file and rename tempfile.dat to stu.dat file. The macro remove() is used for deleting a file and function rename() is used to rename a file.

```
/*P11.19 Program to display the records in sorted order, sorting is performed
in ascending order w.r.t. name*/
#include<stdio.h>
main()
{
    struct record
    {
        char name[20];
        int roll;
        float marks;
    }student,temp,stu[50];
    FILE *fp;
    int i,j,k=0;
    fp=fopen("stu.dat","rb"); /*opened in read mode*/
    if(fp==NULL)
    {
        printf("Error in opening file\n");
        exit(1);
    }
    while(fread(&student,sizeof(student),1,fp)==1)
        stu[k++]=student;
    /*Bubble sort*/
    for(i=0;i<k;i++)
    {
        for(j=0;j<k-1-i;j++)
        {
            if(strcmp(stu[j].name,stu[j+1].name)>0)
            {
                temp=stu[j];
                stu[j]=stu[j+1];
                stu[j+1]=temp;
            }
        }
    }
    printf("\nNAME\tROLLNO\tMARKS\n\n");
}
```

```

for(i=0;i<k;i++)
{
    printf("%s\t\t",stu[i].name);
    printf("%d\t\t",stu[i].roll);
    printf("%f\n",stu[i].marks);
}
fclose(fp);
}

```

Here we have read all the records from the file into an array and then we have sorted that array by bubble sort. After sorting we have displayed the sorted array on the screen. Note that the records in the file stu.dat are in their original unsorted form. If we want the records in the file to become sorted, then we should rewind the file and then write the sorted array to the file with the help of fwrite() like this-

```

rewind(fp);
fwrite( stu, sizeof(student), k, fp);

```

In this case file stu.dat should be opened in update mode. We may also write the sorted records to another file. Now we'll see how to sort on more than one key. Suppose name is primary key and marks is taken as secondary key, and sorting is to be done in ascending order w.r.t. name and in descending order w.r.t. marks.

```

for(i=0;i<k;i++)
{
    for(j=0;j<k-1-i;j++)
    {
        if(strcmp(stu[j].name,stu[j+1].name)>0)
        {
            temp=stu[j];
            stu[j]=stu[j+1];
            stu[j+1]=temp;
        }
        else if(strcmp(stu[j].name,stu[j+1].name)==0)
            if(stu[j].marks<stu[j+1].marks)
            {
                temp=stu[j];
                stu[j]=stu[j+1];
                stu[j+1]=temp;
            }
    }/*End of inner for loop*/
}/*End of outer for loop*/

```

Now we'll write a program to merge two files that are sorted on the same key, such that the merged file is also sorted on the same key. The logic is somewhat similar to the merging of arrays that we had done in chapter 7.

Create two files named sectionA.dat and sectionB.dat using program P11.10. The records in these two files should be in descending order w.r.t marks. So either you can enter the records in sorted order or you may sort the file after entering the records. The program given next merges these two sorted files into a third sorted file. The merged file is named merged.dat and it contains all the records of sectionA.dat and sectionB.dat in descending order w.r.t marks. This file can be read using program

P11.11.

```
/*P11.20 Program to merge two files*/
#include<stdio.h>
struct record
{
    char name[20];
    int roll;
    float marks;
}stu1,stu2;
main()
{
    FILE *fp1,*fp2,*fp3;
    int i,j;
    if((fp1=fopen("sectionA.dat","rb"))==NULL)
    {
        printf("Error in opening file\n");
        exit(1);
    }
    if((fp2=fopen("sectionB.dat","rb"))==NULL)
    {
        printf("Error in opening file\n");
        exit(1);
    }
    if((fp3=fopen("merged.dat","wb"))==NULL)
    {
        printf("Error in opening file\n");
        exit(1);
    }
    i=fread(&stu1,sizeof(stu1),1,fp1);
    j=fread(&stu2,sizeof(stu2),1,fp2);
    while((i==1)&&(j==1))
    {
        if(stu1.marks>stu2.marks)
        {
            fwrite(&stu1,sizeof(stu1),1,fp3);
            i=fread(&stu1,sizeof(stu1),1,fp1);
        }
        else
        {
            fwrite(&stu2,sizeof(stu2),1,fp3);
            j=fread(&stu2,sizeof(stu2),1,fp2);
        }
    }
    while(i==1) /*Write remaining records of sectionA.dat into merged.dat*/
    {
        fwrite(&stu1,sizeof(stu1),1,fp3);
        i=fread(&stu1,sizeof(stu1),1,fp1);
    }
    while(j==1)/*Write remaining records of sectionB.dat into merged.dat*/
    {
        fwrite(&stu1,sizeof(stu1),1,fp3);
```

```

        j=fread(&stu2,sizeof(stu2),1,fp2);
    }
    fclose(fp1);
    fclose(fp2);
    fclose(fp3);
}
}

```

The next program is a menu driven program in which we manage a database of books. All the operations that we have studied till now have been combined in this single program.

```

/*P11.21 Write a program to manage a database of books*/
#include<stdio.h>
#include<string.h>
void insert(FILE *fp);
void del(FILE *fp);
void modify(FILE *fp);
void booksold(FILE *fp);
int search(FILE *fp,char *name);
void display(FILE *fp);
void list(FILE *fp);
struct {
    char name[50];
    int ncopies;
    float cost;
}book;
main()
{
    int choice;
    FILE *fp;
    fp=fopen("books.dat","rb+");
    if(fp==NULL)
    {
        fp=fopen("books.dat","wb+");
        if(fp==NULL)
        {
            puts("Error in opening file\n");
            exit(1);
        }
    }
    while(1)
    {
        printf("1.Insert a new record\n");
        printf("2.Delete a record\n");
        printf("3.Display record of a book\n");
        printf("4.Modify an existing record\n");
        printf("5.List all records\n");
        printf("6 Book sold\n");
        printf("7.Exit\n");
        printf("Enter your choice : ");
        scanf("%d",&choice);

        switch(choice)

```

```
{  
    case 1:  
        insert(fp);  
        break;  
    case 2:  
        del(fp);  
        break;  
    case 3:  
        display(fp);  
        break;  
    case 4:  
        modify(fp);  
        break;  
    case 5:  
        list(fp);  
        break;  
    case 6:  
        booksold(fp);  
        break;  
    case 7:  
        fclose(fp);  
        exit(1);  
    default :  
        printf("Wrong choice\n");  
    }/*End of switch */  
}/*End of while*/  
}/*End of main()*/  
  
void insert(FILE *fp)  
{  
    fseek(fp,0,2);  
    printf("Enter book name : ");  
    scanf("%s",book.name);  
    printf("Enter number of copies : ");  
    scanf("%d",&book.ncopies);  
    printf("Enter cost of book : ");  
    scanf("%f",&book.cost);  
    fwrite(&book,sizeof(book),1,fp);  
}/*End of insert()*/  
  
void del(FILE *fp)  
{  
    FILE *fptmp;  
    char name[20];  
    printf("Enter the name of book to be deleted from database : ");  
    scanf("%s",name);  
    if(search(fp,name)==0)  
        return;  
    fptmp=fopen("tempfile.dat","wb");  
    rewind(fp);  
    while(fread(&book,sizeof(book),1,fp)==1)
```

```

{
    if(strcmp(name,book.name)!=0)
        fwrite(&book,sizeof(book),1,fptmp);
}
fclose(fp);
fclose(fptmp);
remove("books.dat");
rename("tempfile.dat","books.dat");
printf("\nRecord deleted.....\n\n");
fp=fopen("books.dat", "rb+");
}/*End of del()*/



void modify(FILE *fp)
{
    char name[50];
    long size=sizeof(book);
    printf("Enter the name of the book to be modified : ");
    scanf("%s",name);
    if(search(fp,name)==1)
    {
        printf("Enter new data-->\n\n");
        printf("Enter book name : ");
        scanf("%s",book.name);
        printf("Enter number of copies : ");
        scanf("%d",&book.ncopies);
        printf("Enter cost of book : ");
        scanf("%f",&book.cost);
        fseek(fp,-size,1);
        fwrite(&book,sizeof(book),1,fp);
        printf("\nRecord successfully modified\n\n");
    }
}/*End of modify()*/



void booksold(FILE *fp)
{
    char name[50];
    long size=sizeof(book);
    printf("Enter the name of the book to be sold : ");
    scanf("%s",name);
    if(search(fp,name)==1)
    {
        if(book.ncopies>0)
        {
            book.ncopies--;
            fseek(fp,-size,1);
            fwrite(&book,sizeof(book),1,fp);
            printf("One book sold\n");
            printf("Now number of copies = %d\n",book.ncopies);
        }
        else
            printf("Book is out of stock\n\n");
    }
}

```

```

    }
}/*End of booksold( )*/

void display(FILE *fp)
{
    char name[50];
    printf("Enter the name of the book : ");
    scanf("%s",name);
    search(fp,name);
    printf("\nName\t%s\n",book.name);
    printf("Copies\t%d\n",book.ncopies);
    printf("Cost\t%f\n\n",book.cost);
}/*End of display()*/
}

int search(FILE *fp,char *name)
{
    unsigned flag=0;
    rewind(fp);

    while(fread(&book,sizeof(book),1,fp)==1)
    {
        if(strcmp(name,book.name)==0)
        {
            flag=1;
            break;
        }
    }

    if(flag==0)
        printf("\nName not found in file\n\n");
    return flag;
}/*End of search()*/
}

void list(FILE *fp)
{
    rewind(fp);
    printf("\nNAME\tCOPIES\t\tCOST\n\n");
    while(fread(&book,sizeof(book),1,fp)==1)
    {
        printf("%s\t",book.name);
        printf("%d\t\t",book.ncopies);
        printf("%f\n",book.cost);
    }
    printf("\n");
}/*End of list()*/
}

```

We have opened the file books.dat in update(rb+) mode. If the file doesn't exist then fopen() will return NULL, so we'll create the file by opening in "wb+" mode. So first time when we'll run this program, file will be opened in "wb+" mode. The operations permitted in "wb+" mode and "rb+" mode are same but the problem with "rb+" mode is that it can't create a file and with "wb+" mode is that it erases the contents of the file each time file is opened.

The function insert() is simple, we place the position pointer at the end of file by fseek() and then insert a new record there.

The function search() searches for a record in the file, and if the record does not exist then it displays a message and returns 0. If the record exists then the position pointer is placed at the end of the record, which was searched and the function returns 1. Before starting our search we have called rewind() function so that search starts from the beginning of the file. This function search() is used in del(), modify, booksold() and display() functions.

The del() function first calls search(), and if the record doesn't exist we return from the function. Otherwise the record is deleted using a temporary file as we have seen earlier. At the end of del() function the file books.dat is opened in "rb+" mode because this file was closed by this function, and it is the responsibility of this function to open the file in "rb+" before returning to main menu and performing any other operation.

The function modify() is simple. The function booksold() searches for the book to be sold and it reduces the number of copies by 1. If the number of copies is 0 then it displays the message that book is out of stock. For reducing number of copies we have written book.ncopies--. We could write this because book is a global variable and after calling search() it contains the record of the book to be sold.

The function display() searches for a particular record and displays its details on the screen while the function list() displays the details of all the books on the screen.

11.14 Other File Functions

11.14.1 feof()

So far we have used the return value of the function to detect the end of file while reading. For example fgetc(), getc(), getw(), fprintf() return EOF and fgets() returns NULL on end of file. But these functions return the same value both on end of file and on error. So if a function stops reading we can't make out whether the end of file was reached or an error was encountered. We can check separately for these two conditions with the help of perror() and feof().

Declaration: int feof(FILE *fptr);

The macro feof() is used to check the end of file condition. It returns a nonzero value if end of file has been reached otherwise it returns zero.

In the program P11.11 we could have used feof() function to read the file.

```
fread(&student, sizeof(student), 1, fp);
while(!feof(fp))
{
    printf("%s\t", student.name);
    printf("%d\t", student.roll);
    printf("%f\n", student.marks);
    fread(&student, sizeof(student), 1, fp);
}
```

Let us see what would happen if we write the above code like this-

```
while(!feof(fp))
{
    fread(&student, sizeof(student), 1, fp);
    printf("%s\t", student.name);
```

```
    printf("%d\t", student.roll);
    printf("%f\n", student.marks);
}
```

In this case, the last record of the program may be processed twice, since in C end of file is indicated only after an input function has made an attempt to read at the end of file.

11.14.2 ferror()

Declaration: int ferror(FILE *fptr);

The macro ferror() is used for detecting any error occurred during read or write operations on a file. It returns a nonzero value if an error occurs i.e. if the error indicator for that file is set, otherwise it returns zero.

```
/*P11.22 Program to understand the use of ferror()*/
#include<stdio.h>
main()
{
    FILE *fptr;
    int ch;
    fptr=fopen("test", "w");
    ch=getc(fptr);
    if(ferror(fptr))
    {
        printf("Error in read operation\n");
        exit(1);
    }
    else
        printf("%c", ch);
    fclose(fptr);
}
```

Output:

Error in read operation

Here the file is opened in write mode and an attempt is made to read from the file, this gives an error which is detected by ferror().

Generally the return value of the input function is checked to stop reading from the file. Almost all input functions return same value on end of file and on error, so to distinguish between these two cases, the functions feof() and ferror() are used.

```
/*P11.23 Program to understand the use of feof () and ferror()*/
#include<stdio.h>
main()
{
    FILE *fptr;
    int ch;
    if((fptr=fopen("myfile.c", "r"))==NULL)
    {
        printf("File doesn't exist\n");
        exit(1);
    }
    else
```

```

    {
        while((ch=getc(fptr))!=EOF)
            printf("%c",ch);
    }
    if(feof(fptr))
        printf("End of file\n");
    if(ferror(fptr))
        printf("Error\n");
    fclose(fptr);
}

```

11.14.3 clearerr()

Declaration: void clearerr(FILE *fptr);

This function is used to set the end of file and error indicators to 0.

```

/*P11.24 Function to understand the use of clearerr()*/
#include<stdio.h>
main()
{
    FILE *fptr;
    int ch;
    fptr=fopen("test", "w");
    ch=getc(fptr);
    if(ferror(fptr))
    {
        printf("Error in read operation\n");
        clearerr(fptr);
    }
    fclose(fptr);
}

```

11.14.4 perror()

Declaration: void perror(const char *str);

This function is used to display a message supplied by the user along with the error message generated by the system. It prints the argument str, then a colon, then the message related to the current value of errno, and then a newline. errno is a global variable which is set to a particular value whenever an error occurs. The use of this function can be understood by reading the next program P11.25.

11.14.5 rename()

Declaration: int rename(const char *oldname, const char *newname);

This function is used to rename a file. We can give full path also in the argument, but the drives should be the same although directories may differ. On success it returns 0 and on error it returns -1 and errno is set to one of these values-

ENOENT	No such file or directory
EACCES	Permission denied
ENOTSAM	Not same device

```
/*P11.25 Program to understand the use of rename()*/
#include <stdio.h>
main()
{
    char old_name[80], new_name[80];
    printf("Enter the name of file to be renamed : ");
    gets(old_name);
    printf("Enter a new name for the file : ");
    gets(new_name);
    if(rename(old_name, new_name)==0)
        printf("File %s renamed to %s\n", old_name, new_name);
    else
        perror("File not renamed");
}
```

Suppose we give a filename which does not exist then value of errno will be set to ENOENT and the message displayed by perror will be-

File not renamed: No such file or directory

11.14.6 unlink()

Declaration: int unlink(const char *filename);

This function is used for deleting a file from the directory. We can give full pathname also as the argument. The file should be closed before deleting. On success it returns 0 and on error it returns -1 and errno is set to one of these values-

ENOENT	Path or file name not found
EACCES	Permission denied

```
/*P11.26 Program to understand the use of unlink()*/
#include<stdio.h>
main()
{
    FILE *fptr;
    char name[15];
    printf("Enter the file name:");
    scanf("%s", name);
    fptr=fopen(name, "r");
    fclose(fptr);

    if(unlink(name)==0)
        printf("File %s is deleted\n", name);
    else
        perror("File not deleted : ");
}
```

11.14.7 remove()

Declaration: int remove(const char *filename);

remove is a macro that deletes a file from the directory. It is similar to the function unlink().

11.14.8 fflush()

Declaration: int fflush(FILE *fptr);

This function writes any buffered unwritten output to the file associated with fptr. The file is not closed after call to fflush(). On success it returns 0 and on error it returns EOF. If we call this function with stderr, then all the unwritten buffered error messages will be written.

```
fflush (stderr);
```

To flush all output streams we can write-

```
fflush(NULL);
```

Although fflush() is defined for output streams only, but in some compilers fflush() can be used with stdin to discard unread characters on standard input. For example consider this situation-

```
printf("Enter an integer : ");
scanf("%d", &i);
printf("Enter a character : ");
scanf("%c", &ch);
```

After entering the integer, we press Enter key so that the first scanf() terminates. The integer value is read by the scanf() but the newline character(ASCII 10) is still in the input buffer. So the next scanf() reads this newline character in the variable ch, and terminates even before user enters any character. Similar type of problem may occur when we use gets() after scanf(), since gets() terminates when it finds a newline character. To avoid such problems we can use fflush(stdin) to discard any unread characters. For example-

```
printf("Enter an integer : ");
scanf("%d", &i);
printf("Enter a character : ");
fflush(stdin);
scanf("%c", &ch);
```

Note that fflush(stdin) is not portable, it might not work on every compiler.

11.14.9 tmpfile()

Declaration: FILE *tmpfile(void);

tmpfile() creates a temporary binary file and opens it in "wb+" mode. On success it returns a pointer of type FILE that points to the temporary file and on error it returns NULL. The file created by tmpfile() is temporary because it is automatically deleted when closed or when the program terminates.

14.11.10 tmpnam()

Declaration: char *tmpnam(char str[L_tmpnam]);

This function is used to generate a unique name for a file. The argument str can be NULL or an array of minimum L_tmpnam characters. If str is NULL then tmpnam() stores the temporary file name in a internal static array and returns a pointer to that array. If str is not NULL, then tmpnam() stores the temporary file name in str and returns str. The number of different file names that can be generated by tmpnam() during the execution of program is specified by TMP_MAX, which is defined in stdio.h file. Note that tmpnam() only creates a file name, it does not create the file. A file with the name generated by tmpnam() can be opened using fopen(), and it should be closed using fclose().

14.11.11 freopen()

Declaration: FILE *freopen(const char *filename, const char *mode, FILE *fptr);

This function is used to associate a new file with a file pointer. The file associated with fptr is closed as by fclose(), after that the filename that is specified in the first argument is opened in the specified mode as by fopen(), and this fptr is now associated with this file. On success freopen() returns fptr, and on error it returns NULL.

This function is generally used to change the files associated with file pointers stdin, stdout and stderr. For example the file pointer stderr is associated with the screen, if we want to store the error messages in a file instead of displaying them on the screen, then we can use freopen() as-

```
freopen("errorfile", "a", stderr);
```

11.15 Command Line Arguments

The function main() can accept two parameters. The definition of main() when it accepts parameters can be written as-

```
main(int argc, char *argv[ ])
```

.....
.....

The first parameter is of type int and the second one is an array of pointers to strings. These parameters are conventionally named argc(argument counter) and argv(argument vector), and are used to access the arguments supplied at the command prompt. Let us see how this is done.

The program is compiled and executable file is created. In DOS, name of the executable file is same as the name of the program and it has a .exe extension. In UNIX, the executable file is named as a.out. We can give any other name to the executable file in UNIX using -o option as-

```
cc -o myprog myprog.c
```

Now the program is executed at the command prompt, and arguments are supplied to it. The first argument is always the name of executable file. The parameters argc and argv can be used to access the command line arguments. The parameter argc represents the number of arguments on command line. All the arguments supplied at the command line are stored internally as strings and their addresses are stored in the array of pointers named argv.

```
/*P11.27 Program to understand command line arguments*/
#include<stdio.h>
main(int argc, char *argv[ ])
{
    int i;
    printf("argc=%d\n", argc);
    for(i=0;i<argc;i++)
        printf("argv[%d]=%s\n", i, argv[i]);
```

Suppose the name of the program is myprog.c and it is executed on the command prompt as-

```
myprog you r 2 good
```

The variable argc will have value 5, since total five arguments are supplied at the command prompt. The first argument is the name of the program, so argv[0] points to the base address of string "myprog"

(it might contain the whole path of the file). Similarly argv[1] points to the base address of string "you", argv[2] points the base address of string "r", argv[3] points to the base address of string "2", argv[4] points to the base address of string "good".

The output will be :

```
argc = 5
argv[0] = myprog
argv[1] = you
argv[2] = r
argv[3] = 2
argv[4] = good
```

Note that each argument is stored as a string, so argv[3] does not represent the integer 2 but it is pointer to string "2". If we want to use it as an integer in the program then we'll have to convert it to an integer using function atoi().

11.16 Some Additional Problems

Problem 1

Write a program that copies a file to another file. The names of two files should be sent as command line arguments.

```
/*P11.28 Program to copy a file to another*/
#include<stdio.h>
main(int argc,char *argv[])
{
    FILE *source,*dest,*fopen();
    int c;
    if(argc!=3)
    {
        printf("Wrong number of arguments\n");
        exit(1);
    }
    if((source=fopen(argv[1], "r"))==NULL)
    {
        printf(" Can't open source file\n");
        exit(1);
    }
    if((dest=fopen(argv[2], "w"))==NULL)
    {
        printf(" Can't open destination file\n");
        exit(1);
    }
    while((ch=fgetc(source))!=EOF)
        fputc(dest,c);
    fclose(source);
    fclose(dest);
}
```

In this program, first we've checked the number of command line arguments using argc, it is a practice to do so.

Problem 2

Write a program to remove all comment lines from a syntax error free C source program.

```
/*P11.29*/
#include<stdio.h>
main()
{
    FILE *fp1,*fp2;
    char name[50];
    int c1,c2,found='n';
    printf("Enter the file name : ");
    scanf("%s",name);

    if((fp1=fopen(name, "r"))==NULL)
    {
        printf("Error in opening file\n");
        exit();
    }

    fp2=fopen("c:\\new.c", "w");
    c1=fgetc(fp1);
    c2=fgetc(fp1);

    while(c2!=EOF)
    {
        if(c1=='/'&&c2=='*')
            found='y';
        if(found=='n')
            fputc(c1,fp2);
        if(c1=='*'&&c2=='/')
        {
            found='n';
            c2=fgetc(fp1);
        }
        c1=c2;
        c2=fgetc(fp1);
    }
    fclose(fp1);
    fclose(fp2);
}
```

Problem 3

Write a program to count the number of words in a text file. Assume that a word ends with a space, tab, newline, comma, fullstop, semicolon, colon, hyphen.

```
/*P11.30 Program to count the number of words*/
#include<stdio.h>
#include<stdlib.h>
main()
{
    char line[81];
```

```

int i, count=0;
FILE *fptr;
if((fptr=fopen("test.txt", "r"))==NULL)
{
    printf("File doesn't exist\n");
    exit(1);
}
while((fgets(line, 81, fptr))!=NULL)
{
    for(i=0;line[i]!='\0';i++)
        if(is_end(line[i]))
            count++;
}
printf("Number of words in the file = %d\n", count);
fclose(fptr);
}

is_end(int ch)
{
    switch(ch)
    {
        case '\n': case '\t': case ' ': case ',': case '.': case ':':
        case ';': case '-':
            return 1;
    }
    return 0;
}

```

This is a simple program that counts the words in a text file, by counting the characters that terminate a word. This program may give incorrect output in some cases, let us consider those cases and modify the program accordingly.

If the file contains two or more adjacent terminating characters, then the output will be wrong. For example there are two or more adjacent spaces or a full stop followed by a space or a newline. In this case we can modify the if condition as-

```
if ( is_end(line[i]) && !is_end(line[i-1]) )
```

If there are two or more adjacent newline characters(blank lines) in the file then in spite of the modified if condition the output will be wrong. This is because fgets() stops reading when it encounters a newline character. So we can place this condition in the beginning of while loop.

```
if ( line[0] == '\n' )
    continue;
```

Problem 4

Write a program to count the number of occurrences of a given word in a text file. No line of text file contains more than 80 characters(including newline character). Suppose the name of file is test.txt and the word to be counted is "that" and the file contains the following data.

This is the house that Jack built, that is the house that John built.

This is better than that.

His name is John.

She is not sure whether that is his book.

The output should reproduce the lines containing the word along with the number of occurrences of given word, and at the end it should give the frequency of the given word in the file as-

Number of times the given word occurs in the file is 5.

```
/*P11.31*/
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int display(char line[ ],char wordtext[ ]);
main()
{
    char line[81];
    int total = 0;
    FILE *fptr;

    if((fptr=fopen("test.txt", "r"))==NULL)
    {
        printf("File doesn't exist\n");
        exit(1);
    }
    while((fgets(line,81,fptr))!=NULL)
        total=total+display(line,"that");
    printf("Number of times the given word occurs in file is %d\n",total);
    fclose(fptr);

int display(char line[ ],char wordtext[ ])

    int i,j,k,len;
    char str[80];
    int count=0;
    len=strlen(wordtext);
    for(i=0;line[i]!='\0';i++)
    {
        k=0;
        if(is_end(line[i-1])&&is_end(line[i+len]))
        {
            for(k=0,j=i;k<len;j++,k++)
                str[k]=line[j];
            str[k]='\0';
            if(strcmp(str,wordtext)==0)
                count++;
        }
    }
    if(count>0)
    {
        printf("%s",line);
        printf("count=%d\n",count);
    }
    return count;

is_end(int ch )
```

```

{
    switch(ch)
    {
        case '\n': case '\t': case ' ': case ',': case '.': case ':'
        case ';': case '-':
            return 1;
    }
    return 0;
}

```

Output:

This is the house that Jack built, that is the house that John built.

count = 3

This is better than that.

count = 1

She is not sure whether that is his book.

count = 1

Number of times the given word occurs in the file is 5.

The logic used inside the function display() is simple. We find the length of the given word by strlen function. Suppose the length is 4, then we'll extract the 4 letter words in the string str one by one and this string str is compared with the given word using strcmp().

This program would not count the word if it appears in the beginning of the line. For example if the following is a line in the file like this-

that is his book.

Here the word "that" will not be counted. The reason for this is that in our logic we have assumed that before the word, there is a word-ending character(eg space, newline, comma etc).

But this is the first character of the array, so this condition fails and the word is not counted. To get the correct output we can modify the if condition as-

```
if ( ( is_end( line[i-1] ) || i == 0 ) && is_end( line[i+len] ) )
```

If you want to count those occurrences of the word also, when it is embedded inside another word, then remove the above if condition totally. For example if the word to be searched is "rot", then its occurrence in the words like parrot, rotten, carrot, rotate, rotary will be counted.

The search performed by our program is case sensitive, since the comparison performed by strcmp is case sensitive. If we want to perform case insensitive search then we'll have to make a function of our own that performs case insensitive search of two strings.

```

strcmp_in(char *str1,char *str2)
{
    int i;
    for(i=0;str1[i]!='\0';i++)
        if(toupper(str1[i])!=toupper(str2[i]))
            return 1;
    return 0;
}

```

This function compares two strings of equal lengths and returns 1 if they are not same, otherwise 0.

The declaration for the function toupper is in the header file ctype.h.

Problem 5

Write a program to read a text file WRONG.DAT and copy to another file RIGHT.DAT after making corrections in the text as follows:

1. Replace the first character of the word to uppercase, if the previous word was terminated by a fullstop or newline.
2. Replace the word 'a' with the word 'an' if the next word starts with a vowel.
(Alphabets a, e, i, o, u or A, E, I, O, U known as vowels)

```
/*P11.32*/
#include<stdio.h>
#include<ctype.h>
main()
{
    char wrong[81],right[120];
    int ch,i,j;
    FILE *fptr1,*fptr2;

    if((fptr1=fopen("wrong.dat","r"))==NULL)
    {
        printf("File doesn't exist\n");
        exit(1);
    }
    if((fptr2=fopen("right.dat","w"))==NULL)
    {
        printf("File doesn't exist\n");
        exit(1);
    }
    while((fgets(wrong,81,fptr1))!=NULL)
    {
        i=0;j=0;
        while(wrong[i]!='\0')
        {
            if(islower(wrong[i])&&(i==0||wrong[i-1]=='.'))
                right[j++]=toupper(wrong[i++]);
            else if(wrong[i]=='a'&&wrong[i-1]=='a'&&is_vowel(wrong[i+1]))
            {
                right[j++]='n';
                right[j++]=wrong[i++];
            }
            else
                right[j++]=wrong[i++];
        }
        right[j]='\0';
        fputs(right,fptr2);
    }

    is_vowel(int ch)
    switch(ch)
```

```

    {
        case 'a': case 'A': case 'e': case 'E': case 'i': case 'I': cas
        'o': case 'O':
        case 'u': case 'U':
            -return 1;
    }
    return 0;
}

```

Exercise

Assume stdio.h is included in all programs.

```

(1) main()
{
    FILE *fptr;
    unsigned char ch;
    fptr=fopen("myfile.dat", "w");
    while((ch=fgetc(fptr))!=EOF)
        putchar(ch);
    fclose(fptr);
}

(2) main()
{
    FILE *fp;
    int ch;
    fp=fopen("myfile.dat", "w");
    fprintf(fp,"If equal love there cannot be..");
    fputc(26,fp);
    fprintf(fp,"..let the more loving one be me\n");
    fclose(fp);
    fp=fopen("myfile.dat", "r");
    while((ch=fgetc(fp))!=EOF)
        putchar(ch);
}

(3) main()
{
    FILE *fptr1,*fptr2;
    char fname[20];
    printf("Enter the path of first file : ");
    scanf("%s", fname);
    fptr1=fopen(fname, "r");
    if(fptr1==NULL)
    { printf("Error in opening first file\n");      exit(1); }
    fptr2=fopen("c:\mydir\names.dat", "r");
    if(fptr2==NULL)
    { printf("Error in opening second file\n");      exit(1); }
    fclose(fptr1); fclose(fptr2);
}

```

Suppose the path of first file is entered as-

c:\mydir\myfile.dat

```
(4) main()
{
    FILE *fptr;
    int ch;
    fptr=fopen("names.dat",'w');
    while((ch=fgetc(fptr))!=EOF)
        putchar(ch);
    fclose(fptr);
}

(5) main()
{
    char name[50];
    int empid;
    fprintf(stdout,"Enter your name : ");
    fgets(name,50,stdin);
    fprintf(stdout,"Enter your empid : ");
    fscanf(stdin,"%d",&empid);
    fprintf(stdout,"Your empid is : %d", empid);
    fputc('\n',stdout);
    fprintf(stdout,"Your name is : ");
    fputs(name,stdout);
}

(6) main()
{
    FILE *fptr;
    char str[80];
    fptr=fopen("test.txt","r");
    while(fgets(str,80,fptr)!=EOF)
        puts(str);
}

(7) main(int argc,char *argv[])
{
    int i, sum=0;
    for(i=1;i<argc;i++)
        sum=sum+argv[i];
    printf("%d\n",sum);
}
```

Suppose the name of executable file is add and on the command line it is invoked as-
add 2 4 6 8

Programming Exercise

1. Write a program to copy a file to another file such that blank lines are not written to the new file.
2. Write a program to convert all the lower case characters of a file to upper case.

3. Write a program to display the total number of alphabets and numeric characters in file.
4. Write a program to remove all comments from the file. Assume that a comment starts with double slash (//) and continues till the end of the line.
5. Write a program that concatenates any number of files and writes the output in a destination file. The names of files should be passed through command line arguments.
6. Write a program to insert line numbers and page numbers in a file.
7. Consider this structure-

```
struct {
    char name;
    int age;
    int sal;
}
```

Write a program to store 10 records of this structure in the file and sort them on the basis of name and age, where name is the primary key and age is secondary key.

8. Modify the program P11.21, so that the records to be deleted are marked as deleted, and make a function to physically delete the marked records from the file.

Answers

- (1) This program will display the contents of the file and when the end of file is reached the program goes into an infinite loop, since data type of ch is unsigned char while the value of EOF is - which is signed.
- (2) If equal love there cannot be..
Since we are reading in text mode, so 26 is regarded as the end of file character and any text beyond it is not read.
- (3) Error in opening second file
The first file opens successfully while the second file does not. Inside string constants, the backslash is considered as an escape character, so while opening second file we should use double backslashes in the path name. The name of first file is not a string constant, so there is no need to enter double backslashes in it.
- (4) The value of mode should be enclosed in double quotes since it is a string.
- (5) When fprintf(), fputs(), fputc() are called with stdout, they are equivalent to printf(), put() and putchar() respectively. When fscanf(), fgets(), fgetc() are called with stdin, they are equivalent to scanf(), gets() and getchar() respectively.
- (6) fgets() does not return NULL on end of file or on error, not EOF. The contents of the whole file will be displayed and then fgets() will return NULL at the end of file, but the loop condition checks for EOF, so the last line will be displayed infinitely.
- (7) Command line arguments are stored as strings, so we have to use atoi() function to convert them to integers. The correct statement would be-

```
sum = sum + atoi(argv[1]);
```

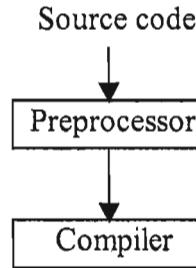
Chapter 12

The C Preprocessor

C has a special feature of preprocessor which makes it different from other high level languages that don't have this type of facility. Some advantages of using preprocessor are-

- (i) Readability of the program is increased.
- (ii) Program modification becomes easy.
- (iii) Makes the program portable and efficient.

We know that the code that we write is translated into object code by the compiler. But before being compiled, the code is passed through the C preprocessor. The preprocessor scans the whole source code and modifies it, which is then given to the compiler.



The lines starting with the symbol # are known as preprocessor directives. When the preprocessor finds a line starting with the symbol #, it considers it as a command for itself and works accordingly. All the directives are executed by the preprocessor, and the compiler does not receive any line starting with # symbol. Some features of preprocessor directives are-

- (i) Each preprocessor directive starts with a # symbol.
- (ii) There can be only one directive on a line.
- (iii) There is no semicolon at the end of a directive.
- (v) To continue a directive on next line, we should place a backslash at the end of the line.
- (vi) The preprocessor directives can be placed anywhere in a program (inside or outside functions) but they are usually written at the beginning of a program.
- (vi) A directive is active from the point of its appearance till the end of the program.

The main functions performed by the preprocessor directives are-

1. Simple Macro Substitution
2. Macros with arguments
3. Conditional Compilation

4. Including files
5. Error generation, pragmas and predefined macro names.

The preprocessor directives that perform these functions are as given below-

#define	#else	#error
#if	#elif	#line
#ifdef	#endif	#pragma
#ifndef	#undef	

There are three operators that are used with these directives-

Defined operator	-	defined()
Stringizing operator	-	#
Token pasting operator	-	##

Apart from the facilities provided through directives, another important task that preprocessor performs is that it replaces each comment by a single space.

12.1 #define

We have already used this directive to define symbolic constants. The general syntax is-

```
#define macro_name macro_expansion
```

Here `macro_name` is any valid C identifier, and it is generally taken in capital letters to distinguish it from other variables. The `macro_expansion` can be any text. A space is necessary between the `macro_name` and `macro_expansion`. The C preprocessor replaces all the occurrences of `macro_name` with the `macro_expansion`. For example-

```
#define TRUE 1
#define FALSE 0
#define PI 3.14159265
#define MAX 100
#define RETIRE_AGE 58
```

C preprocessor searches for `macro_name` in the C source code and replaces it with the `macro_expansion`. For example wherever the macro name `TRUE` appears in the code, it is replaced by 1. These type of constants are known as symbolic constants. These constants increase the readability of the program for example it is better to use the name `PI` instead of the value itself. Moreover if after some time we may decide to change the value of a constant, then the change has to be made only at one place. For example if we decide to increase the `RETIRE_AGE` from 58 to 60, then we have to change the `#define` directive only. If we had not defined this constant then we would have to replace each occurrence 58 by 60. We have already utilized this feature to define the size of an array.

We can also define any string constant in the place of `macro_expansion`.

```
/*P12.1 Program to show that macro expansion can be a string constant
#include<stdio.h>
#define MSSG "I understand the use of #define\n"
main()
{
    printf(MSSG);
}
```

Output:

I understand the use of #define

The C preprocessor searches for the macro name MSG and substitutes the message "I understand the use of #define\n" in the printf().

We can have macro expansions of any sort. Here are some more examples-

```
#define AND &&
#define OR ||
#define BEGIN main( ){
#define END }
#define INFINITE while( 1 )
#define NEW_LINE printf("\n");
#define ERROR printf("An error has occurred\n");
```

There is a semicolon in the last two directives. This semicolon is just a part of the macro_expansion. Now whenever we write NEW_LINE or ERROR in the program, they will be replaced by printf calls and the terminating semicolon.

Remember that the C preprocessor will simply replace the macro_name with the macro_expansion and the macro_expansion can be any text. So it is our responsibility to see that after replacement the resulting code is valid in C. For example if we define a macro MAX like this-

```
#define MAX 5;
```

Now consider this declaration statement-

```
int arr[MAX];
```

This will be translated by the preprocessor as-

```
int arr[5];
```

and this is not valid in C.

If the macro_name appears inside a character constant, string constant or a comment then it is not replaced and is left as it is. For example-

```
#define SUN 2
#define WINDOWS 3
printf("Rays of SUN are coming through the WINDOWS");
Here the replacement will not take place inside the string.
```

It is also possible to define a macro without any macro expansion. For example-

```
#define TEST
```

This is used in compilation directives where the preprocessor only needs to know whether the macro has been defined or not.

12.2 Macros with Arguments

The #define directive can also be used to define macros with arguments. The general syntax is-

```
#define macro_name(arg1, arg2, ....) macro_expansion
```

Here arg1, arg2 are the formal arguments. The macro_name is replaced with the macro_expansion and the formal arguments are replaced by the corresponding actual arguments supplied in the macro call. So the macro_expansion will be different for different actual arguments. For example suppose we define these two macros-

```
#define SUM(x, y) ( (x) + (y) )
#define PROD(x, y) ( (x) * (y) )
```

Now suppose we have these two statements in our program-

```
s = SUM(5, 6);
p = PROD(m, n);
```

After passing through the preprocessor these statements would be expanded as-

```
s = ( (5) + (6) );
p = ( (m) * (n) );
```

Since this is just replacement of text, hence the arguments can be of any data type. For example we may use the macro SUM to find out the sum of long or float types also.

The following statement-

```
s = SUM( 2.3, 4.8 );
```

would be expanded as-

```
s = ( (2.3) + (4.8) );
```

```
/*P12.2 Program to understand macros with arguments*/
#include<stdio.h>
#define SUM(x,y) ((x)+(y))
#define PROD(x,y) ((x)*(y))
main()
{
    int l,m,i,j,a=5,b=3;
    float p,q;
    l=SUM(a,b);
    m=PROD(a,b);
    i=SUM(4,6);
    j=PROD(4,6);
    p=SUM(2.2,3.4);
    q=PROD(2.2,3.4);
    printf("l=%d, m=%d, i=%d, j=%d, p=%f, q=%f\n", l, m, i, j, p, q);
}
```

Output:

l = 8, m = 15, i = 10, j = 24, p = 5.6, q = 7.5

Let us see some more examples of macros with arguments-

```
#define SQUARE(x) ( (x)*(x) )
#define MAX( x, y ) ( (x) > (y) ? (x) : (y) )
#define MIN( x, y ) ( (x) < (y) ? (x) : (y) )
#define ISLOWER(c) ( c >= 97 && c <= 122 )
#define ISUPPER(c) ( c >= 65 && c <= 90 )
```

```
#define TOUPPER(c) ((c) + 'A' - 'a')
#define ISLEAP(y) ((y%400 == 0) || (y%100!=0 && y%4 == 0))
#define BLANK_LINES(n) { int i; for( i = 0; i < n; i++ ) printf("\n"); }
#define CIRCLE_AREA(rad) (3.14 * (rad) *(rad))
```

Note that there should be no space between the macro_name and left parenthesis, otherwise the macro expansion is considered to start from the left parenthesis. For example if we write a macro like this-

```
#define SQUARE (x) ((x)*(x))
```

Now any call like `SQUARE(5)` would be expanded as-

```
(x) ((x)*(x))(5)
```

Here `SQUARE` is considered as a macro without arguments and the text `(x) ((x)*(x))` is regarded as macro expansion. So `SQUARE` is replaced by the macro expansion and `(5)` is written as it is.

12.3 Nesting in Macros

One macro name can also be used for defining another macro name i.e. the macro expansion can also contain the name of another macro. For example-

```
#define PI 3.14
#define PISQUARE PI*PI
```

Now if we have an expression-

```
k = PISQUARE;
```

First time the preprocessor will expand it as-

```
k = PI * PI;
```

Now again the preprocessor rescans this expanded expression, and if any macros are found then it replaces them. This process of rescanning continues till no macros are left in the expression. So finally the expanded expression would be-

```
k = 3.14 *3.14;
```

Here is another example of nesting of macros-

```
#define ISLOWER(c) (c >= 97 && c <= 122)
#define TOUPPER(c) ISLOWER(c) ? ((c) + 'A' - 'a') : c
```

Here the macro `TOUPPER` uses the macro `ISLOWER` in its expansion. If the character is lowercase then only it will be changed to uppercase otherwise it will remain unchanged.

The following program illustrates the use of nested macros-

```
/*P12.3 Program to understand nesting in macros*/
#include<stdio.h>
#define ISLOWER(c) (c>=97&&c<=122)
#define ISUPPER(c) (c>=65&&c<=90)
#define ISALPHA(c) ISLOWER(c) || ISUPPER(c)
#define ISNUM(c) (c>=48&&c<=57)
#define ISALPHANUM(c) ISALPHA(c) || ISNUM(c)
main()
{
```

```

char ch;
printf("Enter a character : \n");
scanf("%c", &ch);
if (ISALPHANUM(ch))
    printf("%c is an alphanumeric character\n", ch);
else
    printf("%c is not an alphanumeric character\n", ch);
}

```

If the macro name appears inside its own expansion, then it is not expanded. So don't try to write recursive macros like this one-

```
#define FACTORIAL(k) k == 0 ? 1 : k * FACTORIAL(k-1) /*Incorrect*/
```

12.4 Problems with Macros

You must be wondering why so many parentheses are used in the macro expansion. Since the preprocessor simply replaces the formal argument with the actual argument, we may not get the desired result in some cases. Let us see what sort of problems can occur while using macros and how they can be avoided.

In the following program we have defined a macro PROD without any parentheses in the macro expansion

```
/*P12.4*/
#include<stdio.h>
#define PROD(x,y) x*y
main()
{
    printf("%d\t", PROD(2,4));
    printf("%d\n", PROD(3+4,5+1));
}
```

Output:

```
8 24
```

Here the first printf() gives the correct result 8, but the second printf() prints 24 while we were expecting the result to be a product of 7 and 6 i.e. 42. Let us see how the preprocessor expands these macros.

PROD(2, 4)	is expanded as	2*4
PROD(3+4 , 5+1)	is expanded as	3+4*5+1

The precedence of multiplication operator is more than addition operator so the calculation went like this: $3+20+1 = 24$ and hence we got the wrong result. This problem can be solved by enclosing each formal argument inside parentheses. So now we rewrite our program like this-

```
/*P12.5*/
#include<stdio.h>
#define PROD(x,y) (x)*(y)
main()
{
    printf("%d\t", PROD(2,4));
    printf("%d\t", PROD(3+4,5+1));
    printf("%d\n", 60/PROD(2,3));
}
```

Output:

```
8 42 90
```

Now the first two printf() give us the desired result. In the third printf we had tried to divide 60 by the product of 2 and 3 and the expected result was 10 but we got 90, so still there is a problem in our macro. Again let us see how preprocessor has done the expansions.

PROD(2, 4)	is expanded as	(2)*(4)
PROD(3+4 , 5+1)	is expanded as	(3+4)*(5+1)
60/PROD(2, 3)	is expanded as	60 / (2)*(3)

Here we can see that in the last case, first 60 is divided by 2 and then the result is multiplied by 3 because / and * operators associate from left to right. So the result was not the expected one. To solve this sort of problem we should enclose whole macro expansion inside parentheses. So the correct way of defining macro is-

```
#define PROD(x, y) ( (x)*(y) )
```

Now 60/PROD(2, 3) would be expanded as 60 / ((2)*(3)) and hence 60 is divided by the product of 2 and 3, and we can get the desired result.

So if your macro is accepting arguments and you want it to work properly then it is better to put parentheses around the entire macro expansion and also around each argument.

Now consider this program and we will find another sort of problem.

```
/*P12.6*/
#include<stdio.h>
#define SQUARE(x) ((x)*(x))
main()
{
    int k=5,s;
    s=SQUARE(k++);
    printf("s = %d, k = %d\n",s,k);
}
```

Output:

```
s = 25, k = 7
```

Here the statement

```
s = SQUARE(k++); is expanded as s = ( (k++) * (k++) );
```

Here value of k is incremented twice while we intended to assign the square of k to variable s, and then increase the value of k only once. Moreover the result of expressions like (k++) *(k++) is undefined in C. So in this case it is better if we use a function instead of macro.

```
/*P12.7*/
#include<stdio.h>
int square(int x)
{
    return x*x;
}
main()
{
    int k=5,s;
```

```

    s=square(k++);
    printf("s = %d, k = %d\n",s,k);
}

```

Output:

s = 25, k = 6

Now we will see a different type of problem. Consider this program, in which we have written a macro to swap the value of 2 variables of any datatype.

```

/*P12.8*/
#include<stdio.h>
#define SWAP(dtype,x,y) { dtype t; t = x, x = y, y = t; }
main( )
{
    int a=2,b=5;
    SWAP(int,a,b)
    printf("a=%d,b=%d\n",a,b);
}

```

Output:

a = 5, b=2

There is no problem in this program. It worked according to our expectations and swapped the values of a and b. The macro was expanded in this way-

SWAP(int, a, b) is expanded to { int t; t = a, a = b, b = t; }

After macro expansion, the main() function would look like this-

```

main( )
{
    int a=2,b=5;
    { int t; t=a, a=b, b=t; }
    printf("a = %d, b = %d\n",a,b);
}

```

Here t is a variable that is local to the block of code in which it is defined.

Now consider the next program, it is similar to the above program except that the names of the variables to be swapped are s and t.

```

/*P12.9*/
#include<stdio.h>
#define SWAP(dtype,x,y) { dtype t; t=x, x=y, y=t; }
main( )
{
    int s=2, t=5;
    SWAP(int,s,t)
    printf("s = %d, t = %d\n",s,t);
}

```

Output:

s = 2, t = 5

The values of variables s and t were not swapped. After macro expansion, the main function would look like this-

```
main()
{
    int s=2, t=5;
    { int t; t=s, s=t, t=t; }
    printf("s = %d, t = %d\n", s, t);
}
```

So now you know why things went wrong. When the macro was expanded, there was a conflict between the variable `t` declared inside the block, and the variable `t` declared outside the block in `main()`. To avoid such type of problems, we can use some naming convention for local variables declared in macros. For example we may decide to write them in capitals.

Here is another problem that can cause the program to give unexpected results.

```
/*P12.10*/
#include<stdio.h>
#define MACRO(x) if(x==0) printf("Out for a Duck\n")
main()
{
    int runs=12;
    if(runs<100)
        MACRO(runs);
    else
        printf("Scored a century\n");
}
```

Output:

Scored a century

The runs were only 12, so the output is wrong. Let's see what happened after the code was expanded.

The expanded code is-

```
if(runs<100)
    if(runs==0) printf("Out for a duck");
else
    printf("Scored a century\n");
```

The `else` part was matched with the `if` part of the macro. To avoid this problem either we can enclose the whole macro inside parentheses or we may use a conditional operator.

```
#define MACRO(x) { if ( x == 0 ) printf("Out for a Duck\n") }
#define MACRO(x) x == 0 ? printf("Out for a duck\n") : printf("\n")
```

12.5 Macros Vs Functions

We have seen that macros with arguments can perform tasks similar to functions. In this topic we'll compare the advantages and disadvantages of functions and macros.

A macro is expanded into inline code so the text of macro is inserted into the code for each macro call. Hence macros make the code lengthy and the compilation time increases. On the other hand the code of a function is written only at one place, regardless of the number of times it is called so the use of functions makes the code smaller.

In functions, the passing of arguments and returning a value takes some time and hence the execution of the program becomes slow while in the case of macros this time is saved and they make the program

faster.

So functions are slow but take less memory while macros are fast but occupy more memory due to duplicity of code. If the macro is small it is good but if it is large and is called many times then it is better to change it into a function since it may increase the size of the program considerably. Sometimes macros can prove very useful and also improve the execution speed. For example in file stdio.h, getchar() and putchar() are defined as macros. If they were defined as functions then there would be a function call for processing each character, which would increase the run time. Preprocessor just replaces or substitutes the text without any sort of checking. So macros can be used with arguments of different types as we have seen earlier in the case of macro SUM in program P1. Functions perform type checking so separate functions have to be written for each data type. remember that lack of type checking facility in macros makes them more error prone.

So whether to use a macro or a function depends on the memory available, your requirement and nature of the task to be performed.

12.6 Generic Functions

Now we'll explore an interesting and powerful feature of preprocessor. We can define macros that can be used to generate functions for different data types. These types of macros are called generic functions. These macros generally take the function name and data type as arguments and the macro call is replaced by a function definition. All this may sound a bit confusing, don't worry the following program will clear all your confusions.

```
/*P12.11 Program to understand generic functions */
#include<stdio.h>
#define MAX(FNAME,DTYPE)
    DTYPF FNAME(DTYPF X, DTYPF Y) \
    { \
        return X>Y ? X:Y; \
    }
MAX(max_int,int)
MAX(max_float,float)
MAX(max_double,double)
main()
{
    int p;
    float q;
    double r;
    p=max_int(3,9);
    q=max_float(7.4,5.7);
    r=max_double(12.34,13.54);
    printf("p = %d,q = %.2f,r = %.2lf \n",p,q,r);
}
```

Output:

p = 9, q = 7.40, r = 13.54

The three macro calls written just before main() are expanded as-

```

MAX(max_int, int)      →    int max_int (int X, int Y)
{
    return X>Y ? X : Y;
}

MAX(max_float, float) →    float max_float (float X, float Y)
{
    return X>Y ? X : Y;
}

MAX(max_double, double) →   double max_double (double X, double Y)
{
    return X>Y ? X : Y;
}

```

So we can see that the three macro calls written before main() are expanded into function definitions. In this way we can generate function definitions for different data types.

12.7 #undef

The definition of a macro will exist from the #define directive till the end of the program. If we want to undefine this macro we can use the #undef directive.

Syntax:

```
#undef macro_name
```

After this directive if the macro_name is encountered in the program then it will not be replaced by the macro_expnsion. In other words now the scope of the macro is limited to the code between #define and #undef directives. This is used when we want the macro definition to be valid for only a part of the program. This directive is generally useful with other conditional compilation directives.

12.8 Stringizing Operator (#)

If in the definition of macro, a formal argument occurs inside a string in the macro expansion then it is not replaced by the actual argument. For example if we have a macro-

```
#define SHOW(var) printf("var = %d", var)
```

Then a call SHOW(x); will be expanded as printf("var = %d", x);

Here the formal argument var outside the string was replaced by the actual argument x, but inside the string this replacement did not take place.

To solve this sort of problem we can use the stringizing operator. This operator is used within the definition of macro. It causes the argument it precedes to be turned into a string or in other words it stringizes a macro argument. So if in the macro expansion, we have a formal argument preceded by #, then both this operator and argument are replaced by the actual argument surrounded within double quotes. We can write the above macro using stringizing operator as-

```
#define SHOW(var) printf(#var “=%d”, var)
```

Now a call `SHOW(x);` will be expanded as `printf("x" "= %d", x);`

We know that adjacent strings are concatenated so after string concatenation this becomes-

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

We can make the above macro more general so that it can display variable of any type.

```
/*P12.12 Program to understand the use of stringizing operator*/
#include<stdio.h>
#define SHOW(var,format) printf(#var " = %" #format "\n", var);
main()
{
    int x=9; float y=2.5; char z='$';
    SHOW(x,d);
    SHOW(y,0.2f);
    SHOW(z,c);
    SHOW(x*y,0.2f);
}
```

Output:

```
x = 9
y = 2.50
z = $
x*y = 22.50
```

The macros in the above program were expanded as-

```
SHOW(x, d); → printf("x" "= %" "d" "\n", x);
SHOW(y, 0.2f); → printf("y" "= %" "0.2f" "\n", y);
SHOW(z, c); → printf("z" "= %" "c" "\n", z);
SHOW(x*y, 0.2f); → printf("x*y" "= %" "0.2f" "\n", x*y);
```

After string concatenation the above statements look like this-

```
printf("x = %d\n", x);
printf("y = %0.2f \n", y);
printf("z = %c\n", z);
printf("x*y = %0.2f \n", x*y);
```

12.9 Token Pasting Operator(##)

Token pasting operator `##` is used in a macro definition to concatenate two tokens into a single token. As the name implies, this operator pastes the two token into one.

```
/*P12.13 Program to understand the use of token pasting operator*/
#include<stdio.h>
#define PASTE(a, b) a##b
#define MARKS(subject) marks_##subject
main()
{
    int k2=14, k3=25;
```

```

int marks_chem = 89, marks_maths = 98;
printf("%d %d ", PASTE(k, 2), PASTE(k, 3));
printf("%d %d\n", MARKS(chem), MARKS(maths));

```

Output:

14 25 89 98

The token pasting operator converts the above statements into-

```

printf( "%d %d", k2, k3 );
printf("%d %d", marks_chem , marks_maths );

```

12.10 Including Files

The preprocessor directive #include is used to include a file into the source code. We have already used this directive to include header files in our programs. The filename should be within angle brackets or double quotes. The syntax is-

```

#include "filename"
#include<filename>

```

The preprocessor replaces the #include directive by the contents of the specified file. After including the file, the entire contents of file can be used in the program. If the filename is in double quotes, first it is searched in the current directory (where the source file is present), if not found there then it is searched in the standard include directory. If the filename is within angle brackets, then the file is searched in the standard include directory only. The specification of standard include directory is implementation defined.

Generally angled brackets are used to include standard header files while double quotes are used to include header files related to a particular program. We can also specify the whole path instead of the path name. For example-

```
#include "C:\mydir\myfile.h"
```

Note that here the path is not a string constant so there is no need to double the backslashes.

#include directive should not be used to include the contents of other ‘.c’ files. It is generally used to include header files only. Header files are given ‘.h’ extension to distinguish them from other C files, although this is not necessary. Header files generally contain macro definitions, declarations of enum, structure, union, typdef, external functions and global variables. The function definitions or global variable definitions should not be put in the header files.

Include files can be nested i.e. an included file can contain another #include directive.

12.11 Conditional Compilation

There may be situations when we want to compile some parts of the code based on some condition. We know that before compilation the source code passes through the preprocessor. So we can direct the preprocessor to supply only some parts of the code to the compiler for compilation.

Conditional compilation means compilation of a part of code based on some condition.

These conditions are checked during the preprocessing phase. The directives used in conditional compilation are-

```
#ifdef      #ifndef      #if      #else      #elif      #endif
```

Every #if directive should end with a #endif directive. The working of these directives is somewhat similar to that of if...else construct.

12.11.1 #if And #endif

An expression which is followed by the #if is evaluated, if result is non-zero then the statements between #if and #endif are compiled, otherwise they are skipped. This is written as-

```
#if constant-expression
.....
statements
.....
#endif
```

The constant-expression should be an integral expression and it should not contain enum constant sizeof operator, cast operator or any keyword or variables. It can contain arithmetic, logical, relation operators. If any undefined identifier appears in the expression, it is treated as having the value zero.

```
/*P12.14 Program to understand the use of #if directive*/
#include<stdio.h>
#define FLAG 8
main()
{
    int a=20,b=4;
    #if FLAG >= 3
        printf("Value of FLAG is greater than or equal to 3\n");
        a=a+b;
        b=a*b;
        printf("Values of variables a and b have been changed\n");
    #endif
    printf("a = %d, b = %d\n",a,b);
    printf("Program completed\n");
}
```

Output:

```
Value of FLAG is greater than or equal to 3
Values of variables a and b have been changed
a = 24, b = 96
Program completed
```

In this program FLAG is defined with the value 8. First the constant expression FLAG ≥ 3 is evaluated since it is true(non zero), hence all the statements between #if and #endif are compiled. Suppose the value of FLAG is changed to 2, now the constant expression FLAG ≥ 3 would evaluate to false hence the statements between #if and #endif will not be compiled. In this case the output of the program would be-

```
a = 20, b = 4
Program completed
```

12.11.2 #else and #elif

#else is used with the #if preprocessor directive. It is analogous to if...else control structure. This is as-

```
#if constant-expression
    statements
#else
    statements
#endif
```

If the constant expression evaluates to non-zero then the statements between #if and #else are compiled otherwise the statements between #else and #endif are compiled.

```
/*P12.15 Program to understand the use of #else directive*/
#include<stdio.h>
#define FLAG 2
main()
{
    int a=20,b=4;
    #if FLAG>=3
        printf("Value of FLAG is greater than or equal to 3\n");
        a=a+b;
        b=a*b;
    #else
        printf("Value of FLAG is less than 3\n");
        a=a-b;
        b=a/b;
    #endif
    printf("a = %d, b = %d\n",a,b);
    printf("Program completed\n");
}
```

Output:

Value of FLAG is greater than or equal to 3

a = 24, b = 96

Program completed

Here the value of FLAG is 8, so the constant expression FLAG ≥ 3 is evaluated to true and hence the statements between #if and #else are compiled. If the value of FLAG is changed to 2, then the constant expression FLAG ≥ 3 would evaluate to zero, so now the statements between #else and #endif will be compiled and the output of the program would be-

Value of FLAG is less than 3

a = 16, b = 4

Program completed

Nesting of #if and #else is also possible and this can be done using #elif. This is analogous to the else...if ladder that we had studied in control statements. Every #elif has one constant expression, first the expression is evaluated if the value is true then that part of code is compiled and all other #elif expressions are skipped, otherwise the next #elif expression is evaluated.

Syntax:

```
#if constant-expression1
    .....
#elif constant-expression2
    .....
```

```
#elif constant-expression3
.....
#else
.....
#endif
```

The above code can be written using #if and #else directives as-

```
#if constant-expression1
.....
#else
#if constant-expression2
.....
#else
#if constant-expression3
.....
#else
.....
#endif
#endif
#endif
```

Here for each #if there should be a #endif, while when using #elif there is only one #endif. Let us take a program that uses #elif directive-

```
/*P12.16 Program to understand the use of #elif directive*/
#include<stdio.h>
#define FLAG 1
main()
{
    int a=20,b=4;
    #if FLAG==0
        printf("Value of FLAG is zero\n");
        a++;
        b++;
    #elif FLAG==1
        printf("Value of FLAG is one\n");
        a--;
        b--;
    #elif FLAG==2
        printf("Value of FLAG is two \n");
        a=a-3;
        b=b-3;
    #else
        printf("Value of FLAG is greater than two or less than zero\n");
        a=a+b;
        b=a-b;
    #endif
    printf("a = %d, b = %d\n",a,b);
    printf("Program completed\n");
}
```

Output:

Value of FLAG is one

```
a = 19, b = 3
Program completed
```

Here the expression FLAG == 1 is true, hence the statements corresponding to this #elif are compiled.

12.11.3 defined Operator

Syntax: defined(macro_name)

This operator is used only with #if and #elif directives. It evaluates to 1 if the macro_name has been defined using #define, otherwise it evaluates to zero. For example:

```
#if defined(FLAG)
```

If the macro FLAG has been defined using #define, then the value of expression defined(FLAG) would be 1, otherwise its value would be 0.

12.11.4 #ifdef and #ifndef

The directives #ifdef and #ifndef provide an alternative short form for combining #if with defined operator.

```
#if defined(macro_name ) is equivalent to #ifdef macro_name
```

```
#if !defined(macro_name ) is equivalent to #ifndef macro_name
```

Syntax of #ifdef-

```
#ifdef macro_name
.....
#endif
```

If the macro_name has been defined with the #define directive, then the statements between #ifdef and #endif will be compiled. If the macro_name has not been defined or was undefined using #undef then these statements are not compiled.

Syntax of #ifndef-

```
#ifndef macro_name
.....
#endif
```

If the macro_name has not been defined using #define or was undefined using #undef, then the statements between #ifndef and #endif are compiled. If the macro_name has been defined, then these statements are not compiled.

Let us take a program-

```
/*P12.17 Program to understand the use of #ifdef directive*/
#include<stdio.h>
#define FLAG
main()
{
    int a=20,b=4;
    #ifdef FLAG
        printf("FLAG is defined\n" );
        a++;
        b++;
    #endif
    printf("a = %d, b= %d\n",a,b);
    printf("Program completed\n");
}
```

Output:

```
FLAG is defined
a = 21, b = 5
Program completed
```

The macro FLAG has been defined, so the statements between #ifdef and #endif are compiled. If we delete the definition of macro FLAG from the program, then these statements won't be compiled and the output of the program would be-

```
a = 20, b = 4
Program completed
```

```
/*P12.18 Program to understand the use of #undef and #ifdef directives*/
#include<stdio.h>
#define FLAG
main()
{
    int a=20,b=4;
    #ifdef FLAG
        printf("FLAG is defined\n");
        a++;
        b++;
    #endif
    #undef FLAG
    #ifdef FLAG
        printf("Preprocessor\n");
        a++;
        b++;
    #endif
    printf("a = %d, b= %d\n",a,b);
    printf("Program completed\n");
}
```

Output:

```
FLAG is defined
a = 21, b = 5
Program completed
```

Here we have undefined the macro FLAG in the middle of program. So the macro FLAG is defined only for the part of the program that is before the #undef directive, and it is undefined for the rest of the program.

```
/*P12.19 Program to understand the use of #ifndef directive*/
#include<stdio.h>
main()
{
    int a=20,b=4;
    #ifndef MAX
        printf("MAX is not defined\n");
        a--;
        b--;
    #endif
```

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

Output:

```
MAX is not defined
a = 19, b = 3
Program completed
```

Here the macro MAX has not been defined, so the statements between #ifndef and #endif are compiled. The existence of defined operator is important in some cases. For example we can't use the short forms(#ifdef and #ifndef) here-

- (i) When we need to check the existence of more than one macros, for example-

```
#if defined(ABC) && defined(PQR) && !defined(XYZ)
```

- (ii) When the existence of macro is to be checked in #elif

```
#if defined(ABC)
.....
#elif defined( PQR )
.....
#elif defined(XYZ)
.....
#endif
```

There are lot of situations where conditional compilation can prove useful e.g. writing portable code, debugging, commenting. We'll discuss all these cases one by one.

12.11.5 Writing Portable Code

Suppose we want to write a program that can be run on different systems. In this case some lines of the code would be system dependent. These lines can be written using conditional compilation directives. Suppose we have made different header files corresponding to different machines. We can write code to include the appropriate header file while working on a specific machine.

```
#if MACHINE == ABC
#define hfile abc.h
#elif MACHINE == PQR
#define hfile pqr.h
#elif MACHINE == XYZ
#define hfile xyz.h
#else
#define hfile newfile.h
#endif
#include "hfile"
```

Now when we want to work on machine ABC, we can define the macro MACHINE as-

```
#define MACHINE ABC
```

So the file abc.h will be included.

If we want to work on machine XYZ, we can define MACHINE as-

```
#define MACHINE XYZ
```

Now the file xyz.h will be included.

Similarly if we have some code in the program that is different for different machines, we can use the above structure.

12.11.6 Debugging

The compiler can detect syntax errors but it is unable to detect run time errors, which result in incorrect output. To trace these types of errors in the program the debugging has to be done by the programmer. In a large program where many functions and variables are involved this can be a difficult task. For this debugging purpose we can insert some printf statements inside the code that tell us about the intermediate results. After the debugging is complete we don't need these statements and we can delete them. But suppose after few days we again need to debug the program, then we'll have to insert all those statements once again. We have another better alternative if we use preprocessor directives. Using these directives we can make these debugging statements active or inactive depending on our needs. The following program shows this-

```
#include<stdio.h>
#define DEBUG
main()
{
    int x;
    #ifdef DEBUG
        printf("Starting main()\n");
    #endif
    .....
    func();
    .....
    #ifdef DEBUG
        printf("Now value of x = %d and y = %d \n",x,y);
    #endif
    .....
}
func()
{
    #ifdef DEBUG
        printf("Inside func()\n");
    #endif
    .....
}
```

When the debugging is complete we can just undefine or delete the macro DEBUG. This will make the #ifdef condition false and all the debugging statements will not be compiled. So in this way we can switch between the debugging and normal mode by defining or deleting the macro DEBUG without having to delete all the debugging statements.

Including #ifdef and #endif for every debugging statement makes things look more lengthy and confusing. To make it more concise we can define another macro SHOW. The macro SHOW is defined in such a way that if macro DEBUG is defined, then SHOW will be replaced by a printf statement, and if DEBUG is not defined then SHOW won't be replaced by anything.

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

```

#define DEBUG
#define SHOW(message)      printf message
#else
#define SHOW(message)
#endif
main()
{
    int x, y ;
    SHOW(("Starting main()\n"));
    .....
    func();
    .....
    SHOW(("Now value of x = %d and y = %d \n",x,y));
    .....
}
func()
{
    SHOW(("Inside func()\n"));
    .....
}

```

Here the argument *message* sent to the macro SHOW consists of the control string and all arguments which we want to send to printf(), and these are to be enclosed in parentheses. This is why we have used two pairs of parentheses while calling SHOW.

12.11.7 Commenting A Part Of Code

Suppose while testing or for some other reason, we want to comment some part of our source code that contains many comments. This can't be done using our usual technique /*...*/ since comments can't be nested in C. So here we can use our conditional compilation directives to comment out parts of our source code.

```

#include<stdio.h>
#define COMMENT
main()
{
    .....
    .....

#ifndef COMMENT
    statement... /* ...comment ...*/
    statement... /* ...comment.....*/           Code to be commented
    .....
#endif
    .....
}

```

If the macro COMMENT is defined, then the code will be commented (not compiled) and if this macro is not defined then it will treated as usual code and will be compiled.

12.11.8 Other Uses of conditional compilation

If you open any header file, you will see that whole contents of the file are enclosed inside #ifndef

```
.. #endif directives. For example if we open stdio.h file
#ifndef __STDIO_H
#define __STDIO_H
....contents of stdio.h file....
#endif
```

This sort of coding ensures that the contents of this file will be included only once. The first time when the file stdio.h is included, the preprocessor finds that macro `__STDIO_H` is not defined, and hence it defines this macro and includes the contents of the file. After this whenever stdio.h is included in our program, the preprocessor finds that the macro `__STDIO_H` is defined and hence it skips all the statements between `#ifndef` and `#endif`.

We can use a similar strategy to avoid multiple definitions in our program. For example the definition of constant `NULL` may be needed by many files such as stdio.h, stdlib.h, string.h, alloc.h etc. This constant can be defined in all these files but if we include more than one files which have definition of `NULL`, then our program will have multiple definitions of `NULL`. So definition of `NULL` is put in a separate file `_null.h` and this file is included in all other files.

These files don't include the `_null.h` file directly, but enclose the include directive inside `#ifndef` and `#endif` like this-

```
#ifndef NULL
#include<_null.h>
#endif
```

Now this file will be included only when `NULL` has not been defined.

12.12 Predefined Macro Names

There are certain predefined macro names which can't be undefined or redefined by `#undef` or `#define`.

<code>__DATE__</code>	String constant that represents the date of compilation in the format "mm dd yyyy"
<code>__TIME__</code>	String constant that represents the time of compilation in the format "hh:mm:ss"
<code>__FILE__</code>	String constant that represents the name of the file being compiled.
<code>__LINE__</code>	Decimal constant that represents the line number being compiled.
<code>__STDC__</code>	Decimal constant, which is 1 if compiled with ANSI standard

```
/*P12.20 Program to display the values of predefined constants*/
#include<stdio.h>
main( )
{
    printf("%s\n", __DATE__);
    printf("%s\n", __TIME__);
    printf("%s\n", __FILE__);
    printf("%d\n", __LINE__);
}
```

Output:

Sep 24 2003
10:24:42

C:\P20.C

8

The macros `_FILE_` and `_LINE_` can be used to write error messages in the program, that can state the line number and file name where the error occurred.

12.13 #line

This directive is used for debugging purposes.

Syntax:

```
#line dec_const string_const
```

Here `dec_const` is any decimal constant and `string_const` is any string constant. This directive assigns `dec_const` and `string_const` to the macros `_LINE_` and `_FILE_` respectively. If the `string_const` is not specified then the macro `_FILE_` remains unchanged.

```
/* P12.21 Program to understand the use of #line, name of this file is
prog.c */
#include<stdio.h>
main()
{
    printf("C in depth\n");
    printf("%d %s\n", __LINE__, __FILE__);
    #line 25 "myprog.c"
    printf("%d %s\n", __LINE__, __FILE__);
}
```

Output:

```
C in depth
6 C:\prog.c
25 myprog.c
```

12.14 #error

This preprocessor directive is used for debugging purpose. `#error` directive stops compilation and displays a fatal error message attached with it.

Syntax:

```
#error message
```

For example:

```
#ifndef MACRO
#error MACRO is not defined
#endif

#ifndef __TINY__
#error This program will not run in the tiny model.
#endif

#ifndef _Windows_ && !defined(_BUILDRTL DLL)
```

```
#error Timer not available for Windows
#endif
```

Suppose a program has been written using ANSI C standard, then this line can be inserted at the beginning of the code.

```
#ifndef __STDC__
#error This program should be run using ANSI C
#endif
```

Suppose there are two files and both of them can't be included at a time, then we can use #error directive to stop inclusion of both files at a time. For example suppose the files stdarg.h and varargs.h can't be included at a time, then we can insert #error directive inside the files like this-

```
/*Structure of file stdarg.h*/
#ifndef __STDARG_H
#define __STDARG_H

#ifdef __VARARGS_H
#error Can't include both stdarg.h and varargs.h
#endif
....contents of file.....
#endif

/*Structure of file varargs.h*/
#ifndef __VARARGS_H
#define __VARARGS_H

#ifdef __STDARG_H
#error Can't include both stdarg.h and varargs.h
#endif
....contents of file.....
#endif
```

12.15 Null Directive

A preprocessor directive consisting only of the symbol # is known as the null directive and it has no effect.

12.16 #pragma

This is an implementation defined directive that allows various instructions to be given to the compiler.

Syntax:

```
#pragma name
```

Here name is the name of the pragma we want. The pragmas may be different for different compilers. You should check your compiler's manual for the pragmas available and their details. Any unrecognized pragma directive is just ignored, without showing any error or warning. Some #pragma statements available in Turbo C are as-

#pragma startup

It allows the programmer to specify the function that should be called upon program startup i.e. b

`main()` is called. For example-

```
#pragma startup func1
```

The function `func1()` will be called before the `main()` function.

#pragma exit

It allows the programmer to specify the function that should be called upon program exit i.e. before the program terminates.

```
#pragma exit func2
```

The function `func2()` will be called just before the program terminates.

#pragma inline

It tells the compiler that inline assembly code is contained in the program.

#pragma warn

We can turn warning messages on or off using this pragma.

12.7 How to see the code expanded by the Preprocessor

Throughout the chapter we have discussed all the concepts by explaining how the preprocessor expands your code. Now we'll tell you how to view the code after the preprocessing phase. The procedure may vary on different systems, we'll discuss it for Turbo C and UNIX.

With Turbo C, we have a utility(executable file) named `cpp`(C preprocessor). This utility creates a file that contains the expanded source code. The name of this file is same as that of source code file and it has a ".i " extension. For example suppose the source code is present in file `d:\myfile.c`, then the expanded code would be present in `myfile.i`. We can create the expanded file at the command prompt as-

```
C:\TC>cpp d:\myfile.c
```

If the source code contains `#include` directive, then we'll have to specify the path of include files. For example if `c:\tc\include` is path of included files then we have to write as-

```
C:\TC>cpp -Ic:\tc\include d:\myfile.c
```

The file `myfile.i` will be created in the directory where `cpp` is present. We can view this file using a text editor or by type command. To know more about the syntax of usage of `cpp` just type `cpp` at the command prompt.

In UNIX, we can view the expanded code by using the option `-E`. With this option the output of the preprocessor is displayed on the terminal. If we use option `-P`, then the output of the preprocessor is stored in a file with the same name as source file but with a ".i " extension. We can use `-I` to inform the preprocessor about the path of included files.

Exercise

Assume `stdio.h` is included in all programs.

```
(1) #define MAX 5;
main()
{
    printf("%d", MAX);
}
```

```

(2) #define MSSG printf("If you lapse, don't collapse\n");
main()
{
    MSSG
}

(3) #define PROD (x,y) ((x)*(y))
main()
{
    int a=3,b=4;
    printf("Product of a and b = %d",PROD(a,b));
}

(4) #define A 50
#define B A+100
main()
{
    int i,j;
    i=B/20;
    j=500-B;
    printf("i = %d, j = %d\n",i,j);
}

(5) #define NEW_LINE printf("\n");
#define BLANK_LINES(n) {int i; for(i=0;i<n;i++) printf("\n");}
main()
{
    printf("When you have a chance");
    NEW_LINE
    printf("to embrace an opportunity");
    BLANK_LINES(3)
    printf("Give it a big hug");
    NEW_LINE
}

(6) #define INFINITE while(1)
#define CHECK(a) if(a==0) break
main()
{
    int x=5;
    INFINITE
    {
        printf("%d ",x--);
        CHECK(x);
    }
}

(7) #define ABS(x) ((x)<0 ?-(x):(x))
main()
{
    int array[4]={1,-2,3,-4};
}

```

```
int *p=array+3;
while(p>=array)
{
    printf("%d ",ABS(*p));
    p--;
}
}

(8) #define . ;
main()
{
    printf("If the lift to success is broken, ");
    printf("Try the stairs.");
}

(9) #define CUBE(x) (x*x*x)
main()
{
    printf("%d\n",CUBE(1+2));
}

(10) #define CUBE(x) ((x)*(x)*(x))
main()
{
    int i=1;
    while(i<=8)
        printf("%d\t",CUBE(i++));
}

(11) #define SWAP(dtype,x,y) {dtype t; t=x+y, x=t-x, y=t-y;}
main()
{
    int a=1,b=2,x=3,y=4,s=25,t=26;
    SWAP(int,a,b)
    SWAP(int,x,y)
    SWAP(int,s,t)
    printf("a=%d,b=%d,x=%d,y=%d,s=%d,t=%d\n",a,b,x,y,s,t);
}

(12) #define INC(dtype,x,i) x=x+i
main()
{
    int arr[5]={20,34,56,12,96},*ptr=arr;
    INC(int,arr[2],3);
    INC(int*,ptr,2);
    printf("*ptr = %d\n",*ptr);
}

(13) #define INT int
main()
{
```

```

    INT a=2,*p=&a;
    printf("%d %d\n",a,*p);
}

(14) #define Y 10
main()
{
    #if X || Y && Z
        printf("Sea in Depth\n");
    #else
        printf("See in depth\n");
    #endif
}

(15) main()
{
    int x=3,y=4,z;
    z=x+y;
    #include<string.h>
    printf("%d\n",z);
}

(16) #define DIFF(FNAME, DTYPE, RTYPE) \
RTYPE FNAME(DTYPE X,DTYPE Y){ return X-Y; }
DIFF(diff_int,int,int)
DIFF(diff_iptr,int*,int)
DIFF(diff_float,float,float);
DIFF(diff_fptr,float*,int);
main()
{
    int iarr[5]={1,2,3,4,5},a,p,q;
    float farr[7]={1.2,2.3,3.4,4.5,5.6,6.7,7.8},b;
    a=diff_int(iarr[4],iarr[1]);
    b=diff_float(farr[6],farr[2]);
    p=diff_iptr(&iarr[4],&iarr[1]);
    q=diff_fptr(&farr[4],&farr[1]);
    printf("a = %d, b = %.1f, p = %d, q = %d\n",a,b,p,q);
}

(17) #define MAX 3
main()
{
    printf("Value of MAX is %d\n",MAX);
    #undef MAX
    #ifdef MAX
        printf("Have a good day");
    #endif
}

(18) #define PRINT1(message) printf(message);
#define PRINT2(message) printf("message");

```

```

#define PRINT3(message) printf(#message);
main()
{
    PRINT1("If we rest, we rust.\n")
    PRINT2("If we rest, we rust.\n")
    PRINT3("If we rest, we rust.\n")
}

(19) #define show(value) printf (" #value " = %d\n", value);
main()
{
    int a=10,b=5,c=4;
    show(a/b*c);
}

(20) #define MACRO(a) if(a<=5) printf(#a" = %d\n",a);
main()
{
    int x=6,y=15;
    if(x<=y)
        MACRO(x);
    else
        MACRO(y);
}

(21) main()
{
    #line 100 "system.c"
    printf("%d %s\n", __LINE__, __FILE__);
}

```

Answers

- (1) This program will show errors since there is a semicolon after 5, and due to this after expansion the printf statement looks like this- printf(" %d ", 5);
- (2) If you lapse, don't collapse
- (3) There is a space between the macro name PROD and left parenthesis, so after macro expansion the printf statement looks like this:

printf("Product of a and b = %d", (x, y) ((x)*(y))(a, b));

Therefore the program gives error that x and y are undefined symbols.

- (4) i = 55, j = 550
The values are calculated as : i = A+100/20; j = 500-A+100;
- (5) When you have a chance
to embrace an opportunity

Give it a big hug

- (6) 5 4 3 2 1
- (7) 4 3 2 1

- (8) This program will give error, since the name of the macro is not a valid C identifier.
- (9) 7
The macro call CUBE(1+2) is expanded as - (1+2*1+2*1+2)
- (10) 6 120 504
The macro CUBE(i++) is expanded as ((x++) *(x++)*(x++)), and the values of such expressions are undefined. So although we have got the output but these values may differ.
- (11) a = 2, b = 1, x = 4, y = 3, s = 7, t = 26
The first two macro calls swapped the values, but the third one gave unexpected results.
The third macro call was expanded as-
- ```
{ int t; t = s+t, s = t-s, t = t-t; }
```
- These calculations were performed using the variable t that is declared inside this block. So the value of variable t that was defined outside the block remains unchanged(26). The variable t declared inside the block contains garbage value, so the variable s gets this garbage value.
- (12) \*ptr = 59
- (13) 2 2
- (14) See in depth
- (15) This program will give many errors, all of them stating that declaration is not allowed here.  
We know that after expansion, the #include directive is replaced by the contents of the file. In this case also the preprocessor inserts all the contents of file string.h after the statement z = x +y; The file string.h contains many declarations, and we know that in C, declarations are allowed only at the beginning of a block before any executable statement. So we get all these errors. This program will compile correctly if we put the #include directive before all executable statements.
- (16) a = 3, b = 4.4, p = 3, q = 3
- (17) Value of MAX is 3
- (18) If we rest, we rust.  
message "If we rest, we rust.\n"
- (19) a/b\*c = 8
- (20) In this program we'll get the error of misplaced else. Since x <= y is true, so MACRO(x) will be expanded as-
- ```
if(x<=y)
    if(x<=5) printf("x = %d\n", x);
else
    if(y<=5) printf("y = %d\n", y);
```
- The double semicolons cause the problem. A single semicolon is considered as a null statement so there are two statements in the if part, but they are not inside parentheses. So to compile the program correctly, we'll have to remove one semicolon from the macro expansion or from the macro call, or we may enclose the macro call inside parentheses.
- (21) 100 system.c

Chapter 13

Operations on Bits

We know that inside the computer, data is represented in binary digits called bits (0 and 1). Till now we were able to access and manipulate bytes only. But some applications, such as system programming require manipulation of individual bits within a byte. In most high-level languages this facility is not available, but C has the special feature to manipulate individual bits of a byte. This feature is implemented through bitwise operators that support bitwise operations. These bitwise operators are-

Operator	Meaning
&	bitwise AND
	bitwise OR
^	bitwise exclusive OR (XOR)
~	One's complement
<<	bitwise left shift
>>	bitwise right shift

Here all the operators are binary, except the complement operator, which is unary. These bitwise operators work on data of integral types only i.e. char, int, short, long including both signed and unsigned types. When these operators are applied to signed types, the result may be implementation dependent, because different implementations represent the signed data in different ways.

These operators operate on each bit of the operand, so while using these operators we'll consider the binary representation (bit pattern) of the operand. While writing the bit pattern, the numbering of bits starts from 0 and they are numbered from right to left. For example if we have an integer variable `x = 0x3C60`, the binary pattern of this integer is 0011 1100 0110 0000. The numbering of bits is-

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	1	0	0	0	1	1	0	0	0	0	0

Here the rightmost(0th) bit is the least significant bit, while the leftmost(15th) bit is the most significant bit. A bit is on if it has value 1, while it is off if it has value 0.

We'll use a function `bit_pattern()` in our programs, that will take an integer as input and print the 16 bit binary pattern of that integer. The definition of this function is given in program P13.13. In our

examples, we'll represent the integers in hexadecimal, because it is easier to convert hexadecimal to binary and vice versa.

All bitwise operators except the complement operator can be combined with the assignment operator to form the compound assignment operators-

$\&=$, $|=$, $<<=$, $>>=$, $^=$

13.1 Bitwise AND (&)

It is a binary operator and requires two operands. These operands are compared bitwise i.e. all the corresponding bits in both operands are compared. The resulting bit is 1, only when the bits in both operands are 1, otherwise it is 0.

Boolean Table

Bit of operand1	Bit of operand2	Resulting Bit
0	0	0
0	1	0
1	0	0
1	1	1

Let us take $a = 0x293B$ and $b = 0x1A2F$ are two integer variables. The binary representations of these variables and the result after performing bitwise AND operation is shown below-

a	0010 1001 0011 1011	(0x293B)
b	0001 1010 0010 1111	(0x1A2F)
$a \& b$	0000 1000 0010 1011	(0x082B)

```
/*P13.1*/
#include<stdio.h>
main()
{
    int a,b;
    printf("Enter values for a and b : ");
    scanf("%d%d",&a,&b);
    printf("a = %d\t\t",a); bit_pattern(a);
    printf("b = %d\t\t",b); bit_pattern(b);
    printf("a&b = %d\t\t",a&b); bit_pattern(a&b);
}
```

If you want to enter and display the values of a and b in hexadecimal, then use `%x`. The definition of function `bit_pattern()` is given in program P13.13.

13.2 Bitwise OR (|)

The corresponding bits of both operands are compared and the resulting bit is 0, only when the bits in both operands are 0, otherwise it is 1.

Boolean Table

Bit of operand 1	Bit of operand 2	Resulting Bit
0	0	0
0	1	1
1	0	1
1	1	1

The result of bitwise OR operation performed between variables a and b is shown below-

a	0010 1001 0011 1011	(0x293B)
b	0001 1010 0010 1111	(0x1A2F)
a b	0011 1011 0011 1111	(0x3B3F)

```
/*P13.2*/
#include<stdio.h>
main()
{
    int a,b;
    printf("Enter values for a and b : ");
    scanf("%d%d",&a,&b);
    printf("a = %d\t\t",a); bit_pattern(a);
    printf("b = %d\t\t",b); bit_pattern(b);
    printf("a|b = %d\t\t",a|b); bit_pattern(a|b);
}
```

13.3 Bitwise XOR (^)

The corresponding bits of both operands are compared and the resulting bit is 1, if bits of both operands have different value, otherwise it is 0.

Boolean Table

Bit of operand 1	Bit of operand 2	Resulting Bit
0	0	0
0	1	1
1	0	1
1	1	0

The result of bitwise XOR operation performed between variable a and b is shown below-

a	0010 1001 0011 1011	(0x293B)
b	0001 1010 0010 1111	(0x1A2F)
a ^ b	0011 0011 0001 0100	(0x3314)

```
/*P13.3*/
#include<stdio.h>
main()
```

```

{
    int a,b;
    printf("Enter values for a and b : ");
    scanf("%d%d", &a, &b);
    printf("a = %d\t", a);      bit_pattern(a);
    printf("b = %d\t", b);      bit_pattern(b);
    printf("a^b = %d\t", a^b);  bit_pattern(a^b);
}

```

When the bitwise operators `&`, `|`, `^` operate on two operands of different sizes, then the size of smaller operand is increased to match the size of larger operand. For example if there are two operands of sizes 16 and 32 bits, then the 16-bit operand will be converted to 32 bits. The extra bits are added to the left of the smaller operand. If the smaller operand is *unsigned* then all these extra bits are filled with zeros, and if it is *signed* then these bits are filled with the sign bit.

13.4 One's Complement (`~`)

One's complement operator is a unary operator and requires only one operand. It negates the value of the bit. If the bit of the operand is 1 then the resulting bit is 0 and if the bit of the operand is 0 then the resulting bit is 1.

Boolean Table

Bit of operand	Resulting Bit
0	1
1	0

a	0010 1001 0011 1011	(0x293B)
<code>~a</code>	1101 0110 1100 0100	(0xD6C4)
b	0001 1010 0010 1111	(0x1A2F)
<code>~b</code>	1110 0101 1101 0000	(0xE5D0)

```

/*P13.4*/
#include<stdio.h>
main( )
{
    int a;
    printf("Enter value for a : ");
    scanf("%d",&a);
    printf("a = %d\t",a);      bit_pattern(a);
    printf("~-a = %d\t",~a);  bit_pattern(~a);
}

```

When the complement operator is applied to an operand twice, then result is the original operand i.e. `~-a` is equal to `a`. This feature of complement operator can be used for encrypting and decrypting data. To encrypt the data, we can apply complement operator to it, and to decrypt the data i.e. to get back the original data we can apply the complement operator to the encrypted data. For example

Original data : 0000 1111 0101 0011

Encrypted data : 1111 0000 1010 1100 (By applying `~` to original data)

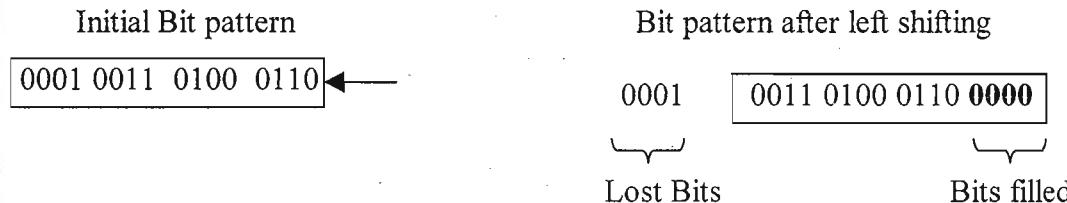
Decrypted data : 0000 1111 0101 0011 (By applying ~ to encrypted data)

13.5 Bitwise Left Shift (<<)

This operator is used for shifting the bits left. It requires two operands. The left operand is the operand whose bits are shifted and the right operand indicates the number of bits to be shifted. On shifting the bits left, an equal number of bit positions on the right are vacated. These positions are filled in with 0 bits. Let us take an integer variable a = 0x1346. The binary representation of x is-

0001 0011 0100 0110

Now we'll find out a << 4

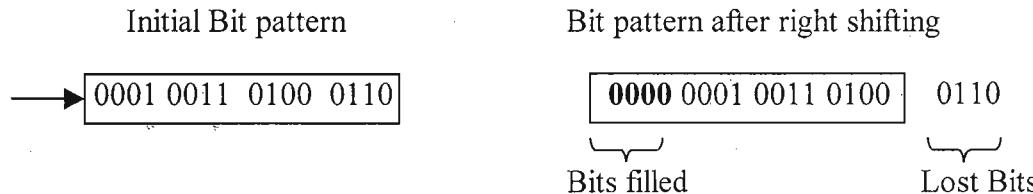


On shifting all bits to the left by 4, the leftmost 4 bits are lost while the rightmost 4 bit positions become empty which are filled with 0 bits.

```
/*P13.5*/
#include<stdio.h>
main()
{
    int a;
    printf("Enter value for a : ");
    scanf("%d",&a);
    printf("a = %d\t",a);    bit_pattern(a);
    a=a<<2;
    printf("a = %d\t",a);    bit_pattern(a);
}
```

13.6 Bitwise Right Shift (>>)

This operator is similar to the left shift operator, except that it shifts the bits to the right side. On shifting the bits right, an equal number of bit positions on the left are vacated. These positions are filled in with 0 bits in *unsigned integers*. We'll again take a variable a = 0x1346, and this time we'll find out a >>



Right shift in an unsigned integer

In right shift if the first operand is a signed integer, then the result is compiler dependent. Some compilers follow logical shift while others may follow arithmetic shift.

Logical shift – The vacated bits are always filled with zeros.

Arithmetic shift – The vacated bits are filled with the value of the leftmost bit in the initial bit pattern. If the leftmost bit is 1, then the vacated positions are filled with 1, and if the leftmost bit is 0, then the vacated positions are filled with 0.

The following two examples show arithmetic shift in signed integers. In first case the leftmost bit is 1 so the vacated bits are filled with 1, and in the second case the leftmost bit is 0, so the vacated bits are filled with 0.

→ 1000 1010 0011 1011	1111 1000 1010 0011	1011
→ 0100 1010 0011 1011	0000 0100 1010 0011	1011

Arithmetic right shift in signed integers

Actually the leftmost bit represents the sign bit. If the number is negative, then it is set to 1. So in other words we can say that, in arithmetic shift the vacated bits are filled with the sign bit.

We know that expressions like $x/2$ or $x*5$ do not change the value of x , similarly $x<<4$ or $x>>3$ will not change the value of x . If we write $x = x<<4$, then only the value of x will be changed.

While using left shift and right shift operators, the result is undefined if the right operand is negative or it is more than the number of bits used to represent the left operand.

```

/*P13.6*/
#include<stdio.h>
main()
{
    int a;
    printf("Enter value for a : ");
    scanf("%x",&a);
    printf("a = %x\t",a);    bit_pattern(a);
    a=a>>2;
    printf("a = %x\t",a);    bit_pattern(a);
}
```

13.7 Multiplication and Division by 2 using shift operators

The effect of shifting one bit right is equivalent to integer division by 2, and the effect of shifting one bit left, is equivalent to multiplication by 2. So we can use shift operators to multiply and divide integers by power of 2. For example to multiply an integer by 2^2 , we'll shift it left by 2 bits, to multiply by 2^3 we'll shift it left by 3 bits. Similarly to divide an integer by 2^2 , we'll shift it right by 2 bits, to divide by 2^3 we'll shift it right by 3 bits.

In the case of right shift, if the compiler follows logical shift then the effect of division by 2 is not seen in signed integers.

	Statement	Bit Pattern of a	Decimal value of a	
(i)	a = 45;	0000 0000 0010 1101	45	
(ii)	a = a << 1;	0000 0000 0101 1010	90	(= 45*2 ¹)
(iii)	a = a << 2;	0000 0001 0110 1000	360	(= 90*2 ²)
(iv)	a = a << 5;	0010 1101 0000 0000	11520	(= 360*2 ⁵)
(v)	a = a << 3;	0110 1000 0000 0000	26624	(!= 11520*2 ³)
(vi)	a = a >> 1;	0011 0100 0000 0000	13312	(= 26624 / 2 ¹)
(vii)	a = a >> 6;	0000 0000 1101 0000	208	(= 13312 / 2 ⁶)
(viii)	a = a >> 4;	0000 0000 0000 1101	13	(= 208 / 2 ⁴)
(ix)	a = a >> 1;	0000 0000 0000 0110	6	(= 13 / 2 ¹)

In the statement (v), the effect of multiplication by 2³ is not seen, this is because a bit with a value of 1 has been dropped from the left. So in the case of left shift, if a bit with value of 1 is shifted and lost, then the effect of multiplication by powers of 2 is not seen.

In statement (ix), value of a is 13 and it is shifted right by 1 bit, the result is 6 which shows that the remainder is discarded.

13.8 Masking

Masking is an operation in which we can selectively mask or filter the bits of a variable, such that some bits are changed according to our needs while the others remain unchanged. Through masking, we can manipulate bits in a bit pattern and perform operations such as testing a bit, inverting a bit, switching on or off a bit. Masking is performed with the help of bitwise operators. The bit pattern to be masked is taken as the first operand and the second operand is called mask. The mask is selected according to our needs.

In further discussion, we will take an arbitrary bit pattern of 16 bits and show different types of masking on it.

b₁₅ b₁₄ b₁₃ b₁₂ b₁₁ b₁₀ b₉ b₈ b₇ b₆ b₅ b₄ b₃ b₂ b₁ b₀

Here b₁₅, b₁₄, ..., b₁, b₀ are bits and they may be 0 or 1.

13.8.1 Masking Using Bitwise AND

b ₁₅	b ₁₄	b ₁₃	b ₁₂	b ₁₁	b ₁₀	b ₉	b ₈	b ₇	b ₆	b ₅	b ₄	b ₃	b ₂	b ₁	b ₀	(Original Value)	
0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	(Mask)	
0	0	0	0	0	b ₁₁	b ₁₀	b ₉	b ₈	0	0	0	0	b ₃	b ₂	b ₁	b ₀	(New Value)

Here we can see that whenever there is a 0 in the mask, new bit value becomes 0 while the new bit value remains unchanged when there is a 1 in the mask.

We can switch off any bit by using & operator. For example if we want to switch off the last 4 bits in a bit pattern, we can choose the mask such that last four bits are 0.

a = a & 0xFFFF0

$b_{15} b_{14} b_{13} b_{12} b_{11} b_{10} b_9 b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$	(Original Value)
1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0	(Mask)
$b_{15} b_{14} b_{13} b_{12} b_{11} b_{10} b_9 b_8 b_7 b_6 b_5 b_4 0 0 0 0$	(New Value)

Later in this chapter, we'll see a better method to switch off bits using bitwise AND and complement operator.

Bitwise AND operator is generally used to test whether a particular bit is on or off. Suppose we want to test the 5th bit, we will select the mask with only 5th bit on-

$b_{15} b_{14} b_{13} b_{12} b_{11} b_{10} b_9 b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$	(Original Value)
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0	(Mask)
0 0 0 0 0 0 0 0 0 0 b_5 0 0 0 0 0	(New value)

If 5th bit in the original bit pattern is 0, then the new value will be zero, and if 5th bit in original bit pattern is 1, then the new value will be non zero

```
mask = 0x20;
if((a&mask)==0)
    printf("5th bit is off");
else
    printf("5th bit is on");
```

So by choosing an appropriate mask and using bitwise AND operator, we can test whether a bit is on or off. We can retrieve bit value by using if...else, conditional operator or shift operator. Let us see different methods for retrieving 5th bit-

```
mask = 0x20;
1. if((a&mask)==0)
    bit=0;
else
    bit=1;

2. bit=(a&mask)?1:0;
3. bit=(a&mask)>>5;

/*P13.7 A program to test 5th bit*/
#include<stdio.h>
main()
{
    int a,bit,mask=0x20;
    printf("Enter an integer : ");
    scanf("%d",&a);
    printf("a = %d\t",a);    bit_pattern(a);
    if((a&mask)==0)
        bit=0;
    else
        bit=1;
    printf("5th bit is %d\n",bit);
}
```

13.8.2 Masking Using Bitwise OR

$b_{15} b_{14} b_{13} b_{12} b_{11} b_{10} b_9 b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$	(Original Value)
0 0 0 0 1 1 1 0 0 0 0 0 1 1 1 1	(Mask)
$b_{15} b_{14} b_{13} b_{12} 1 1 1 b_7 b_6 b_5 b_4 1 1 1 1 1$	(New Value)

Here we can see that whenever there is a 1 in the mask, new bit value becomes 1 while the new bit value remains unchanged when there is a 0 in the mask.

We can use bitwise OR operator to switch on a particular bit. The mask is chosen in such a way that the bits to be switched on should be 1 and rest of the bits should be 0.

Suppose we want to switch on the 5th bit

$b_{15} b_{14} b_{13} b_{12} b_{11} b_{10} b_9 b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$	(Original Value)
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0	(Mask)
$b_{15} b_{14} b_{13} b_{12} b_{11} b_{10} b_9 b_8 b_7 b_6 1 b_4 b_3 b_2 b_1 b_0$	(New Value)

```

mask = 0x20;
a = a | mask;
/*P13.8 A program to switch on the 5th bit*/
#include<stdio.h>
main()
{
    int a,bit,mask=0x20;
    printf("Enter an integer : ");
    scanf("%d",&a);
    printf("a = %d\t",a); bit_pattern(a);
    a=a|mask;
    printf("After switching on 5th bit, the value of a is :\n");
    printf("a = %d\t",a); bit_pattern(a);
}

```

13.8.3 Masking Using Bitwise XOR

$b_{15} b_{14} b_{13} b_{12} b_{11} b_{10} b_9 b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$	(Original Value)
0 0 0 0 1 1 1 0 0 0 0 0 1 1 1 1	(Mask)
$b_{15} b_{14} b_{13} b_{12} \bar{b}_{11} b_{10} \bar{b}_9 \bar{b}_8 \bar{b}_7 b_6 b_5 b_4 \bar{b}_3 \bar{b}_2 \bar{b}_1 \bar{b}_0$	(New Value)

Here we can see that whenever there is 1 in the mask the corresponding bit value is inverted while the new bit value remains unchanged when there is a 0 in the mask.

We have seen earlier that the complement operator(\sim) complements all the bits in a number. But if we want to complement some particular bits in a number, then we can use bitwise XOR operator.

The mask should be chosen in such a way that the bits to be complemented should be 1, rest of the bits should be 0.

Now suppose we want to invert the value of 5th bit.

$b_{15} \ b_{14} \ b_{13} \ b_{12} \ b_{11} \ b_{10} \ b_9 \ b_8 \ b_7 \ b_6 \ b_5 \ b_4 \ b_3 \ b_2 \ b_1 \ b_0$	(Original Value)
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0	(Mask)

$b_{15} \ b_{14} \ b_{13} \ b_{12} \ b_{11} \ b_{10} \ b_9 \ b_8 \ b_7 \ b_6 \ b_5 \ b_4 \ b_3 \ b_2 \ b_1 \ b_0$ (New Value)

mask = 0x20;

a = a ^ mask

So we can use this operator to toggle bits between values 0 and 1. For example if we have an integer variable a, and we want to toggle its 5th bit.

Suppose initially 5th bit in variable a is 1

mask = 0x20;

a = a ^ mask (5th bit changes to 0, and all other bits unchanged)

a = a ^ mask (5th bit changes to 1, and all other bits unchanged)

a = a ^ mask (5th bit changes to 0, and all other bits unchanged)

From the above operations we can also see that when an operand is XORed with a value twice, result is same as the original operand. This feature can be used for encryption.

For example, if initial value of a variable var = 0xF13A

var = var ^ 0x1F; (Encryption)

var = var ^ 0x1F; (Decryption)

After first statement the value of var becomes 0xF125, while after second statement we get back original value of var (0xF13A).

```
/*P13.9 A program to toggle 5th bit using bitwise XOR operator*/
#include<stdio.h>
main()
{
    int a,bit,mask=0x20;
    printf("Enter an integer : ");
    scanf("%d",&a);
    printf("a = %d\t",a);    bit_pattern(a);
    a = a^mask;
    printf("a = %d\t",a);    bit_pattern(a);
    a = a^mask;
    printf("a = %d\t",a);    bit_pattern(a);
}
```

When a value is XORed with itself the result is zero, this is obvious since all corresponding bits will be same. For example value of expressions like var^var will be zero. This feature of XOR operator can be utilized to compare two values for equality.

13.8.4 Switching off Bits Using Bitwise AND and Complement Operator

We have seen that bits can be switched off using bitwise AND operator. Let us see what sort of portability problem arises when we use bitwise AND operator.

Suppose we want to switch off the 5th bit of an integer variable a, we can take the mask as

1111 1111 1101 1111 (0xFFDF)

`a = a & 0xFFDF;`

Now suppose we use this code on a computer that uses 32 bits to store an integer, then 0xFFDF would be represented as-

0000 0000 0000 0000 1111 1111 1101 1111

Now this mask will switch off the 16 leftmost bits also, which is not intended. So on a computer using 32 bits for an integer, the appropriate mask to switch off the 5th bit should be-

1111 1111 1111 1111 1111 1101 1111 (0xFFFFFFFDF)

`a = a & 0xFFFFFFFDF;`

It would be better if we could write the same line of code for both type of computers. We can do this by combining the complement operator with the AND operator as-

`a = a & ~0x20`

On a computer using 16 bits for an integer, this is interpreted as-

`a = a & 1111 1111 1101 1111`

And on a computer using 32 bits for an integer, this is interpreted as-

`a = a & 1111 1111 1111 1111 1111 1101 1111`

```
/*P13.10 A program to switch off the 5th bit*/
#include<stdio.h>
main()
```

```
int a,bit,mask=0x20;
printf("Enter an integer : ");
scanf("%d",&a);
printf("a = %d\t",a);    bit_pattern(a);
a=a&~mask;
printf("After switching off the 5th bit, the value of a is :\n");
printf("a = %d\t",a);    bit_pattern(a);
```

Now let us summarize the different manipulations on 5th bit in an integer variable a-
`mask = 0x20;`

Test 5th bit in variable a : `a & mask`

Switch on 5th bit in variable a : `a = a | mask;`

Switch off 5th bit in variable a : `a = a & ~ mask;`

Invert 5th bit in variable a : `a = a ^ mask;`

Similarly we can manipulate any bit by these operations, we just have to choose the mask appropriately.

For manipulating 6th bit : `mask = 0x40` (0000 0000 0100 0000)

For manipulating 9th bit : `mask = 0x200` (0000 0010 0000 0000)

For manipulating 3rd and 6th bits : `mask = 0x48` (0000 0000 0100 1000)

We can calculate the mask by left shifting integer 1 by position of bit.

`mask = 1 << bitposition;`

For manipulating 4th bit : `mask = 1<<4` (0000 0000 0001 0000)

For manipulating 6th bit : `mask = 1<<6` (0000 0000 0100 0000)

```
/*P13.11 Program to test any bit in an integer*/
#include<stdio.h>
main()
{
    int a,bit,mask,bitposition;
    printf("Enter an integer : ");
    scanf("%d",&a);
    printf("Enter the bit position : ");
    scanf("%d",&bitposition);
    mask=1<<bitposition;
    printf("a = %d\t",a);    bit_pattern(a);
    if((a&mask)==0)
        bit=0;
    else
        bit=1;
    printf("The bit at position %d is %d\n",bitposition,bit);
}
```

If we want to manipulate all bits one by one, then we can use a loop, for example this program will switch on all the bits in the integer variable a.

```
/*P13.12 Program to switch on all the bits in an integer variable*/
#include<stdio.h>
main()
{
    int a,mask,i;
    printf("Enter the value of a : ");
    scanf("%d",&a);
    printf("%d\t", a); bit_pattern(a);
    for(i=0;i<=15;i++)
    {
        mask=1<<i;
        a=a|mask; /*switch on the ith bit*/
    }
    printf("%d\t",a);    bit_pattern(a);
}
```

There is one other way also to calculate the appropriate mask for a particular bit position. We can take an array of masks as-

```
unsigned int arr_mask[] = { 0x1, 0x2, 0x4, 0x8, 0x10, 0x20, 0x40, 0x80, 0x100, 0x200, 0x400, 0x800,
                           0x1000, 0x2000, 0x4000, 0x4000};
```

Now the mask for manipulating any bit can be calculated as-

```
mask = arr_mask[bitposition];
```

For example to switch on the 4th bit we can write

```
x = x | arr_mask[4];
```

13.9 Some additional Problems

Problem 1

Write a program to print bit pattern of a 16 bit integer.

Now we are in a position to write the definition of the function bit_pattern(), that we have been using till now. Printing the binary pattern of an integer requires testing of each bit in the integer. The bit numbering is from right to left, but we'll have to print the bits from left to right so firstly 15th bit will be printed, then 14th and so on till 0th bit. So we will start testing bits from the 15th bit onwards. To test 15th bit, mask should be $1 \ll 15$. Similarly to test 14th bit, mask should be $1 \ll 14$ and so on.

```
/*P13.13 Printing the binary pattern of a 16 bit integer*/
#include<stdio.h>
main()
{
    int a;
    printf("Enter an integer : ");
    scanf("%d",&a);
    bit_pattern(a);
}
bit_pattern(int a)
{
    int i,mask;
    for(i=15;i>=0;i--)
    {
        mask=1<<i;
        if((a&mask)==0)
            printf("0");
        else
            printf("1");
    }
    printf("\n");
}
```

In 1st iteration mask is : 1000 0000 0000 0000 15th bit is tested and printed

In 2nd iteration mask is : 0100 0000 0000 0000 14th bit is tested and printed

In 3rd iteration mask is : 0010 0000 0000 0000 13th bit is tested and printed

.....

.....

In last iteration mask is : 0000 0000 0000 0001 0th bit is tested and printed

To make this program portable, we can modify the for loop as-

```
for( i = sizeof(int)-1 ; i>=0; i- -)
```

Problem 2

Write a program to find whether the number is even or odd, using bitwise operators.

A number will be odd if its least significant bit(rightmost) is 1 and it will be even if its least significant bit is 0. This means we just have to test whether the least significant bit is 1 or 0. We know that the bitwise AND operator is suitable for testing bits. Now we have to choose an appropriate mask. Since we have to check the rightmost bit, so we will take a mask in which the rightmost bit is 1 and all other bits are 0.

```
/*P13.14 Program to find whether a number is even or odd*/
#include<stdio.h>
```

```

main()
{
    int num;
    int mask=0x1;
    printf("Enter a number : ");
    scanf("%d",&num);
    if((num&mask)==0)
        printf("Number is even\n");
    else
        printf("Number is odd\n");
}

```

Problem 3

Write a program that inputs a binary pattern less than or equal to 16 bits and converts it to an integer

```

/*P13.15 Program to convert a binary pattern to an integer*/
#include<stdio.h>
main()
{
    char bit;
    int i,num=0;
    printf("Enter any bit pattern less than or equal to 16 bits ::\n");
    for(i=0;i<=15;i++)
    {
        bit=getchar();
        if(bit=='0')
            num=num<<1;
        else if(bit=='1')
            num=(num<<1)+1;
        else
            break;
    }
    printf("Hexadecimal : %x\n",num);
    printf("Decimal : %d\n",num);
}

```

Problem 4

Write a program to swap the first 8 bits with the last 8 bits in a given unsigned 16 bit integer.

For example if initially the bit pattern is - 1000 1111 0011 0001

After swapping it should become - 0011 0001 1000 1111

This can be achieved as-

x	1000 1111 0011 0001
x<<8	0011 0001 0000 0000
x>>8	0000 0000 1000 1111
x<<8 x>>8	0011 0001 1000 1111

If the integer is unsigned then this method will work properly but if the integer is a signed negative number, then according to arithmetic shift after right shifting, the leftmost 8 bits will be filled with zeros.

and we will not get our desired result. To avoid this problem we can force the leftmost 8 bits to be zero after the right shift. So now this will be as:

```
x = (x<<8) | ((x>>8) & 0x00FF)
/*P13.16 Program to swap first 8 bits with the last 8 bits in a 16-
bit integer*/
#include<stdio.h>
main()
{
    int num,i;
    printf("Enter number in hexadecimal : ");
    scanf("%x",&num);
    printf("Before swapping, num = %x\n",num);
    bit_pattern(num);
    num=(num<<8) | ((num>>8)&0x00FF) ;
    printf("After swapping, num = %x\n",num);
    bit_pattern(num);
}
```

Problem 5

Write a program to swap the values of 2 variables using bitwise XOR.

```
/*P13.17 Swapping the values without using a temporary variable through
bitwise XOR*/
#include<stdio.h>
main()
{
    int x,y;
    printf("Enter values for x and y : ");
    scanf("%d%d",&x,&y);
    printf("x = %d, y = %d\n",x,y);
    x=x^y;
    y=x^y;
    x=x^y;
    printf("x = %d, y = %d\n",x,y);
}
```

Problem 6

Write a program to print the bit pattern of 2's complement of a number

We know that the 2's complement can be found out by adding 1 to the one's complement. Another way to obtain the 2's complement is that, scan the bit pattern from right to left, and invert all the bits after the first appearance of a bit with value 1. For example-

Bit pattern : 0000 0001 0110 0000

2's complement : 1111 1110 1010 0000

```
/*P13.18 Program to print the two's complement of a number.*/
#include<stdio.h>
main()
```

```

int num,i,mask;
printf("Enter a number : ");
scanf("%d",&num);
printf("Two's complement is : %d\n",~num+1);

for(i=0;i<=15;i++)
{
    mask=1<<i;
    if((num&mask)!=0)           /*Find a bit with value 1*/
        break;
}
for(i=i+1;i<=15;i++)
{
    mask=1<<i;
    num=num^mask;   /*Invert the bit*/
}
printf("Two's complement is : %d\n",num);
}

```

Problem 7

Write a function that rotates bits to right by n positions.

Initial bit pattern : 0000 0000 0000 1101

After rotating right by 4 bits : 1101 0000 0000 0000

Similarly write a function that rotates bits to left by n positions.

Initial bit pattern : 1001 0000 1001 0011

After rotating left by 3 bits : 1000 0100 1001 1100

```

/*P13.19*/
#include<stdio.h>
main()
{
    int num,n,i,bit;
    printf("Enter number in hexadecimal : ");
    scanf("%x",&num);
    bit_pattern(num);
    printf("Enter number of positions to be rotated : ");
    scanf("%d",&n);
    n=n%16;
    num=rotate_right(num, n);
    printf("Number after right rotation is : %x\n",num);
    bit_pattern(num);
    num=rotate_left(num,n);
    printf("Number after left rotation is : %x\n",num);
    bit_pattern(num);
}
rotate_right(int num,int n)
{
    int i,lsbit;

```

```

for(i=1;i<=n;i++)
{
    lsbbit=num&1?1:0;           /* Test LSB */
    num=num>>1;
    if(lsbbit==0)
        num=num&~(1<<15); /*Switch off MSB*/
    else
        num=num|(1<<15);   /*Switch on MSB*/
}
return num;

rotate_left(int num,int n)

int i,msbit;

for(i=1;i<=n;i++)
{
    msbit=num&(1<<15)?1:0;      /*Test MSB*/
    num=num<<1;
    if(msbit==0)
        num=num&~1;           /*Switch off LSB*/
    else
        num=num|1;            /*Switch on LSB*/
}
return num;

```

Problem 8

Write a function mult(num, n, err) that uses the shift operators `>>` and `<<` to compute the value of $(\text{num} * 2^n)$. Assume that both num and n are unsigned positive integers and are 32 bits in size. The 'err' parameter is set by the function to zero if calculation is successful and 1 if an overflow occurs. An overflow occurs when the resultant value doesn't fit in the 32 bit unsigned integer. Do not use add, subtract, multiply or divide to compare the result.

```

/*P13.20*/
#include<stdio.h>
main()
{
    unsigned int num,n,err=0;
    printf("Enter the number and power of 2 : ");
    scanf("%u%u",&num,&n);
    mult(num,n,err);

mult(int num,int n,int err)
{
    unsigned int result=num;
    while(n>0)
    {
        num=num<<1; /*Multiply by 2*/
        n--;
        if(num<result)

```

```

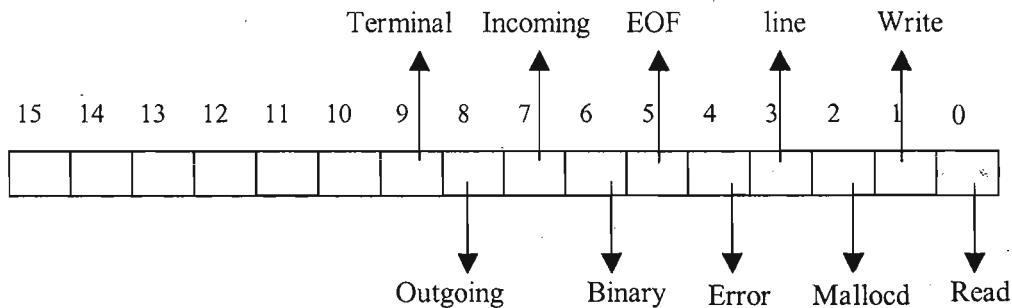
    {
        err=1;
        break;
    }
    result=num;
}
if(err==1)
    printf("Overflow\n");
else
    printf("Result = %u\n",result);
}

```

Now we know very well how to manipulate individual bits within a byte. This bit manipulation is useful in maintaining boolean flags. For example we can use a character variable to hold 8 flags or an integer variable to hold 16 flags. Each bit can be used to represent a flag. This will definitely save memory when many flags are to be used in a program and also it is easy to maintain single variable for several flags. We can test, switch on or off, toggle any individual flag by bitwise operators. Different mask is taken for manipulating each flag. To increase readability, generally each mask corresponding to a flag is given a name using #define. Let us take an example and understand how this is done. We'll declare a variable of unsigned integer type, and then we'll use the individual bits to represent different flag associated with the attributes of a file. The variable is declared as-

```
unsigned int flags;
```

Now we'll use 10 bits of this variable to represent 10 different flags.



To increase readability, we'll give a name to each mask corresponding to these bits.

#define _F_RDWR 0x0003	/* Read/write flag	0 th and 1 st bit */
#define _F_READ 0x0001	/* Read only file	0 th bit */
#define _F_WRIT 0x0002	/* Write only file	1 st bit */
#define _F_BUF 0x0004	/* Malloc'ed Buffer data	2 nd bit */
#define _F_LBUF 0x0008	/* line-buffered file	3 rd bit*/
#define _F_ERR 0x0010	/* Error indicator	4 th bit*/
#define _F_EOF 0x0020	/* EOF indicator	5 th bit*/
#define _F_BIN 0x0040	/* Binary file indicator	6 th bit*/
#define _F_IN 0x0080	/* Data is incoming	7 th bit*/
#define _F_OUT 0x0100	/* Data is outgoing	8 th bit*/
#define _F_TERM 0x0200	/* File is a terminal	9 th bit */

Now we can perform masking operations to manipulate the different flags like this-

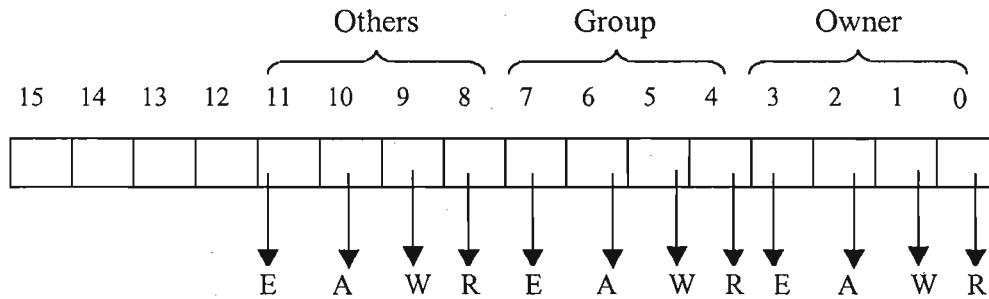
```
/*To switch on error flag*/
flags=flags|_F_ERR;

/*To switch off write flag*/
flags=flags&~_F_WRIT;

/*To toggle incoming flag*/
flags=flags^_F_IN;

/*To test EOF flag*/
if((flags&_F_EOF)==0)
    printf("EOF flag is not set ");
else
    printf("EOF flag is set ");
```

Now we'll take another example in which a variable named permissions will contain information about the different access permissions given to different class of users. There are three types of users viz. owner, group, others, and there are four types of permissions that can be given to these users viz. read, write, append, execute.



The masks corresponding to these bits can be named as:

```
#define R_OWNER 0x0001          /* 0th bit */
#define W_OWNER 0x0002          /* 1st bit */
#define A_OWNER 0x0004          /* 2nd bit */
#define E_OWNER 0x0008          /* 3rd bit */
#define R_GROUP 0x0010          /* 4th bit */
#define W_GROUP 0x0020          /* 5th bit */
#define A_GROUP 0x0040          /* 6th bit */
#define E_GROUP 0x0080          /* 7th bit */
#define R_OTHERS 0x0100          /* 8th bit */
#define W_OTHERS 0x0200          /* 9th bit */
#define A_OTHERS 0x0400          /* 10th bit */
#define E_OTHERS 0x0800          /* 11th bit */

unsigned int permission;
permission = 0x19F;
```

After this statement owners will get all permissions, group will get read and execute permission, and others will get only read permission.

To grant execute permission to others we can write-

```
permission = permission | E_OTHERS; /*Switch on 11th bit*/
```

To take away read permission from group we can write-

```
permission = permission & ~R_GROUP; /*Switch off 4th bit*/
```

13.10 Bit Fields

We have seen how to access and manipulate individual bits or group of bits using bitwise operators. Bit fields provide an alternative method of accessing bits. A bit field is a collection of adjacent bits is defined inside a structure but is different from other members because its size is specified in terms of bits. The data type of a bit field can be int, signed int or unsigned int.

Let us take an example that shows the syntax of defining bit fields.

```
struct tag {
    unsigned a:2;
    unsigned b:5;
    unsigned c:1;
    unsigned d:3;
};
```

Here the structure has four bit fields a, b, c and d. The sizes of a, b, c and d are 2, 5, 1 and 3 respectively. The bit fields can be accessed like other members of the structure using the dot operator. Whenever they appear inside expressions, they are treated like integers(of smaller range). The following are some valid expressions using bit fields.

```
struct tag var;
var.a=2;
printf("%d", var.b);
x=var.a+var.b;
if(var.c==1)
    printf("Flag is on\n");
```

If the size of a bit field is n, then the range of values that the bit field can take is from 0 to $2^n - 1$.

Bit field	Size in bits	Range of values
a	2	0 to $2^2 - 1$ (0 to 3)
b	5	0 to $2^5 - 1$ (0 to 31)
c	1	0 to $2^1 - 1$ (0 and 1)
d	3	0 to $2^3 - 1$ (0 to 7)

It would be invalid to assign any value to a bitfield outside its range, for example-

```
var.b = 54; /*Invalid*/
```

We can't apply sizeof and address operators to bit fields. So we can't use scanf to input a value in a bit field.

```
scanf("%d", &var.a); /*Invalid*/
```

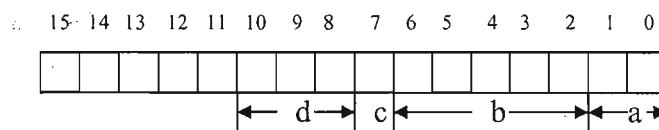
We may input the value into a temporary variable and then assign it to the bit field.

```
scanf("%d", &temp);
var.a = temp;
```

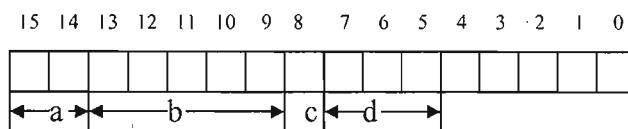
If we have pointer to structure then arrow operator(\rightarrow) can be used to access the bit fields. Code using bitfields is easier to understand than the equivalent masking operations, but bitfields are considered non-portable as most of the issues related with them are implementation dependent.

We had mentioned that the data type of bit fields can be int, signed int or unsigned int. A plain int field may be treated as signed by some compilers while as unsigned by others. So for portability, it is better to clearly specify signed or unsigned in the declaration of bit fields. If a bitfield is defined as signed, then it should be at least 2 bits long because one bit is used for sign.

The direction of allocation of bit fields within an integer is also implementation dependent. Some C compilers allocate the bit fields from right to left, while others may allocate them from left to right. If the bit fields are assigned from right to left, then the first field occupies the least significant bit. If the bit fields are assigned from left to right, then the first field occupies the most significant bit.



Four bit fields assigned from right to left

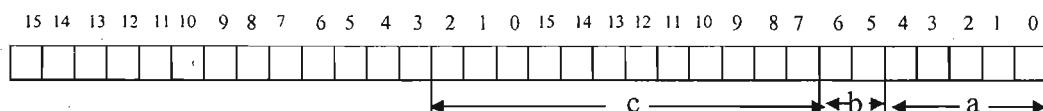


Four bit fields assigned from left to right

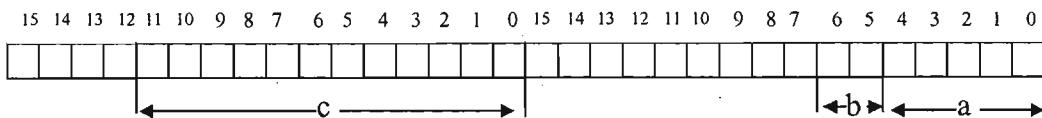
The other implementation dependent issue about bit fields is that whether they can cross integer boundaries or not. For example consider this structure-

```
struct tag{
    unsigned a:5;
    unsigned b:2;
    unsigned c:12;
};
```

The first two fields occupy only 7 bits in a 16 bit integer, so 9 bits can still be used for another bit field. But the bits needed for next bit field is more than 9. Some implementations may start the next field(c) from a new integer, while others may just place the next field in 12 adjacent bits i.e. 9 unused from the previous integer and 3 bits from the next integer.



Bit fields cross integer boundary



Bit fields do not cross integer boundary

We can define unnamed bitfields for controlling the alignment of bitfields within an integer. The size of unnamed bitfield provides padding within the integer.

```
struct tag{
    unsigned a:5;
    unsigned b:2;
    unsigned :9; /* padding within first integer */
    unsigned c:12;
};
```

Here the unnamed bitfield fills out the 9 unused bits of first integer, so the bitfield c starts with the second integer. Since these 9 bits don't have any name, so they can't be accessed from the program.

We can also take the size of the unnamed bitfield zero, so we have no need to provide the padding and next bitfield will start with second integer.

```
struct tag{
    unsigned a : 5;
    unsigned b : 2;
    unsigned : 0;
    unsigned c : 12; /*this field starts from next integer */
};
```

Now we'll take the example of file permissions that we had seen earlier using bitwise operators.

```
struct permission
{
    unsigned r_owner:1;
    unsigned w_owner:1;
    unsigned a_owner:1;
    unsigned e_owner:1;
    unsigned r_group:1;
    unsigned w_group:1;
    unsigned a_group:1;
    unsigned e_group:1;
    unsigned r_others:1;
    unsigned w_others:1;
    unsigned a_others:1;
    unsigned e_others:1;
};
```

```
struct permissions perm = {1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0} ;
```

After this statement owners will get all permissions, group will get read and execute permission, others will get only read permission.

Now we can write statements to grant and take permissions like this-

```
perm.e_others = 1; /*grant execute permission to others*/
```

```
perm.r_group = 0;           /*take away read permission from group*/
```

Exercise

Assume stdio.h is included in all programs.

- (1) main()
{
 int x=5,y=4;
 if(x||y)
 printf("x&y = %d, x&&y = %d\n",x&y,x&&y);
}
- (2) main()
{
 int x=5,y=13,z;
 z=x^y;
 printf("z=%d\n",z);
}
- (3) main()
{
 int x;
 x=(0xFF>>8)<<8;
 printf("%x\n",x);
}
- (4) main()
{
 int k;
 k=((3<<4)^ (96>>1));
 printf("%d\n",k);
}
- (5) main()
{
 int k=0xC9FB; /*1100 1001 1111 1011*/
 k&=~(1<<5);
 printf("%x\t",k);
 k|=(1<<2);
 printf("%x\t",k);
 k^=(1<<14);
 printf("%x\n",k);
}
- (6) main()
{
 int x=0x1F;
 x<<2;
 printf("%x ",x);
 x>>2;
 printf("%x\n",x);
}

```
}

(7) main()
{
    unsigned int arr_mask[ ]={ 0x1,0x2,0x4,0x8,0x10,0x20,0x40,
                                0x80,0x100,0x200,0x400,0x800,
                                0x1000,0x2000,0x4000,0x4000};

    int i,num=0x38F;
    for(i=1;i>=0;i--)
        num&arr_mask[i] ? printf("1"):printf("0");
    printf("\n");
}

(8) main()
{
    int num=0xA01D,pos=3,bit;
    int mask=1<<pos;
    bit=(num&mask)>>pos;
    printf("%d",bit);
}

(9) main()
{
    int i,bit,num=0x238E;
    unsigned int mask;
    for(i=15;i>=0;i--)
    {
        mask=1<<i;
        bit=(num&mask)>>i;
        printf("%d",bit);
    }
}

(10)main()
{
    int num=0x1F,pos=3,bit;
    bit=(num>>pos)&1;
    printf("%d\n",bit);
}

(11)main()
{
    int i,num=0xA0DF ;
    for(i=15;i>=0;i--)
        printf("%d", (num>>i)&1);
}

(12)main()
{
    int i,num=0x1A3B;
    unsigned int mask;
```

```

mask=1<<15;
for(i=15;i>=0;i--)
{
    (num&mask) ? printf("1"):printf("0");
    mask=mask>>1;
}
}

```

Answers

(1) $x \& y = 1, x \& \&y = 1$

(2) $z = 8$

(3) 0

(4) 0

$3 << 4$ evaluates to 48, $96 >> 1$ also evaluates to 48, and 48^48 evaluates to 0.

(5) c9db c9df 89df

$1 << 5$ is 0000000000100000

$\sim(1 << 5)$ is 111111111011111

$1 << 2$ is 0000000000000000100

$1 << 14$ is 01000000000000000000

(6) 1f 1f

There will be no change, if the statement is written as $x = x << 2;$, then the value of x will change.

(7) 0000001110001111

Prints the binary pattern of an integer, we have taken the masks in an array.

(8) 1

Extracts the bit at position pos

Bit pattern of num is 1010 0000 0001 1101

Bit pattern of 1 is 0000 0000 0000 0001

Bit pattern of mask = $1 << 3$ is 0000 0000 0000 1000

Bit pattern of num & mask is 0000 0000 0000 1000

Bit pattern of $(\text{num} \& \text{mask}) >> 3$ 0000 0000 0000 0001

In this way we can extract a bit at position pos.

(9) 0010001110001110

Prints the binary pattern of an integer, the logic of extracting a bit is same as used in previous question.

(10) 1

Extracts the bit at position pos

Bit pattern of num 0000 0000 0001 1111

Bit pattern of $\text{num} >> 3$ 0000 0000 0000 0011

(Now the bit that was at position 3 has come to the rightmost position)

Bit pattern of 1 0000 0000 0000 0001

Bit pattern of $(\text{num} >> 3) \& 1$ 0000 0000 0000 0001

(11) 1010 0000 1101 1111

Prints the binary pattern of an integer, the logic of extracting a bit is same as used in previous question.

(12) 0001 1010 0011 1011

Prints the binary pattern of an integer

Initially the mask is $1 \ll 15$ 1000 0000 0000 0000

mask = mask>>1 0100 0000 0000 0000

mask = mask>>1 0010 0000 0000 0000

So we can see that in each iteration the mask changes, and hence we can extract all the bits.

Chapter 14

Miscellaneous Features In C

14.1 Enumeration

Words speak more than numbers and this is the reason for the inclusion of enumerated data types in C. Sometimes the replacement of integer constants like 1, 2, 3 by some meaningful and descriptive names, enhances the readability of the code and makes it self documenting. For example suppose we are making a date related program, then it would be better if we could use names like Jan, Feb, Mar, Apr instead of the numbers 1, 2, 3, 4.

An enumeration type is a user defined data type, which can take values only from a user defined list of named integer constants called enumerators. The syntax of defining an enumeration data type is same as that of structure or union. The general format of definition is -

```
enum tag{  
    member1;  
    member2;  
    .....  
    .....  
};
```

Here enum is a keyword, tag is an identifier that specifies the name of the new enumeration type being defined, and member1, member2 are identifiers which represent integer constants and are called enumerator constants or enumerators. The list of these enumerators is called enumerator list. Note that unlike structure and union, here the members inside the braces are **not** variables, they are named integer constants.

After the definition, we can declare variables of this new data type as-

```
enum tag var1, var2, var3;
```

var1, var2, var3 are variables of type **enum tag**. These variables can take values only from the enumerator list.

The variables can also be declared with the definition as-

```
enum tag {  
    member1;  
    member2;  
    .....  
    .....  
} var1, var2, var3;
```

the tag is optional. Let us take an example-

```
enum month{ Jan, Feb, Mar, Apr, May, Jun };
```

Here a new data type month is defined and the enumerator list contains six enumerators.

Internally the compiler treats these enumerators as integer constants. These are automatically assigned integer values beginning from 0, 1, 2... etc till the last member of the enumeration. In the above example these enumerators will take following values-

Jan	0
Feb	1
Mar	2
Apr	3
May	4
Jun	5

These are the default values assigned to the enumerators. It is also possible to explicitly assign value to enumerators but in this case, the successive unassigned enumerators will take values one greater than the value of the previous enumerator. For example-

```
enum month{ Jan, Feb = 4, Mar, Apr, May = 11, Jun };
```

Now the enumerators will take following values-

Jan	0
Feb	4
Mar	5
Apr	6
May	11
Jun	12

We can assign any signed integer value to enumerators, provided the value is within the range of -32768 to 32767. It is also possible to assign same value to more than one enumerator.

```
enum month{ Jan, January = 0, Feb = 1, February = 1, Mar = 2, March = 2 };
```

The enumerated variables can be processed like other integer variables. We can assign values to them from the enumerator list or they can be compared to other variables and values of the same type. For example-

```
enum month{ Jan, Feb, Mar, Apr, May, Jun }m1, m2;
m1 = Mar;
m2 = May;
```

Now m1 has integer value 2 and m2 will take the value 4.

Any variable of type enum month can take values only from the 6 enumerators specified in the list. For example this is invalid-

```
m1 = Dec; /*Invalid*/
```

Some other examples of enum data type definitions are-

```
enum suit{ Spades, Hearts, Clubs, Diamonds};
enum position { Ace = 1, King, Queen, Jack, Ten, Nine, Eight, Seven, Six, Five, Three, Two };
enum month{ Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec };
enum day{ Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };
enum color{ white, black, red, green, blue, yellow, pink, brown};
```

```
enum boolean{ true , false};  
enum switch{ off, on};  
enum subject{ Hindi, English, Maths, Physics, Chemistry, Biology, History };  
enum base{ Binary = 2, Octal = 8, Decimal = 10, Hexadecimal = 16};
```

are some examples of code using these enum definitions-

```
enum color walls, floor;  
if(walls==pink)  
    floor=blue;  
else  
    floor=white;  
  
enum day today;  
today=Monday;  
if(today==Sunday)  
    printf("Holiday\n");  
else if(today==Saturday)  
    printf("Half working day\n");  
else  
    printf("Full working Day\n");  
  
enum subject s;  
int passmarks;  
switch(s)  
{  
    case hindi:  
    case english:  
        passmarks=25;  
        break;  
    case maths:  
        passmarks=40;  
        break;  
    case physics:  
    case chemistry:  
        passmarks=35;  
        break;  
    default:  
        passmarks=33;  
}
```

Suppose we need a function that returns number of days in a month. It can be written as-

```
enum month m;  
int days;  
if( m==Apr || m==Jun || m==Sep || m==Nov )  
    days=30;  
if( m==Jan || m==Mar || m==May || m==Jul || m==Aug || m==Oct || m==Dec)  
    days=31;  
if( m==Feb )  
    days=28 ;
```

```
/* P14.1 Program to print the value of enum variables*/
#include<stdio.h>
main()
{
    enum month{Jan, Feb, Mar, Apr, May, Jun}m1, m2;
    m1=Mar;
    printf("m1 = %d \n", m1);
    printf("Enter value for m2 : ");
    scanf("%d", &m2);
    printf("m2 = %d \n", m2);
}
```

Output :

```
m1 = 2
Enter the value for m2 : 5
m2 = 5
```

It is not possible to perform input and output in terms of enumerator names. The input and output only in the form of their integer values. You may be tempted to use %s to output and input enum variables but this is invalid since enumerators are not strings.

```
printf("%s", m1); /*Invalid*/
```

Since enumerators are identifiers so their name should be different from other identifiers in the scope. The following code is wrong because the identifier chemistry has been used at two different places.

```
enum group1{ physics, chemistry, maths};
enum group2{ zoology, botany, chemistry }; /*Invalid*/
float maths; /* Invalid*/
```

We can use typedef in the definition of enum, for example:-

```
typedef enum{false, true} boolean;
```

Now we can define variables like this-

```
boolean flag = true;
```

We have seen earlier that we can define constants using #define also, for example-

```
#define Sun 0
#define Mon 1
#define Tue 2
#define Wed 3
```

Now we'll compare the constants defined by #define preprocessor directive and enum.

- By defining an enum, we define a new type whose variables can be declared but using #define we can only give name to some constant values.
- enum is a part of C language(32 keywords) but #define is not.
- All #define directives are global, while enum obeys the scope rules.
- In #define we have to explicitly assign the values to all constants, while in enum the values assigned by the compiler automatically.

We can always write our programs by using integer variables instead of enumerated variables, but use of enum in complicated programs makes the program more understandable.

14.2 Storage Classes

In addition to data type, each variable has one more attribute known as storage class. The proper use of storage classes makes our program efficient and fast. In larger multifile programs the knowledge of storage classes is indispensable. We can specify a storage class while declaring a variable. The general syntax is-

```
storage_class datatype variable_name;
```

There are four types of storage classes-

1. Automatic
2. External
3. Static
4. Register

The keywords **auto**, **static**, **register**, **extern** are used for these storage classes. So we may write declaration statements like this-

```
auto int x, y;
static float d;
register int z;
```

When the storage class specifier is not present in the declaration, compiler assumes a default storage class based on the place of declaration.

A storage class decides about these four aspects of a variable-

- (1) Lifetime - Time between the creation and destruction of a variable.
- (2) Scope - Locations where the variable is available for use.
- (3) Initial value - Default value taken by an uninitialized variable.
- (4) Place of storage - Place in memory where the storage is allocated for the variable.

Now we'll discuss all these storage classes one by one in detail.

14.2.1 Automatic

All the variables declared inside a block/function without any storage class specifier are called automatic variables. We may also use the keyword **auto** to declare automatic variables, although this is generally not done. The following two declaration statements are equivalent and both declare a and b to be automatic variables of type int.

```
func( )
{
    int a,b;
    .....
}

func( )
{
    auto int a,b;
    .....
}
```

The uninitialized automatic variables initially contain garbage value. The scope of these variables is inside the function or block in which they are declared and they can't be used in any other function/block.

They are named automatic since storage for them is reserved automatically each time when the control enters the function/block and are released automatically when the function/block terminates. For example-

```
/*P14.2 Program to understand automatic variables*/
#include<stdio.h>
main()
{
    func();
    func();
    func();
}
func()
{
    int x=2,y=5;
    printf("x=%d,y=%d",x,y);
    x++; y++;
}
```

Output:

```
x = 2, y = 5
x = 2, y = 5
x = 2, y = 5
```

Here when the function func() is called first time, the variables x and y are created and initialized, and when the control returns to main(), these variables are destroyed. When the function func() is called for the second time, again these variables are created and initialized, and are destroyed after execution of the function. So automatic variables come into existence each time the function is executed and are destroyed when the function terminates.

Since automatic variables are known inside a function or block only, so we can have variables of same name in different functions or blocks without any conflict. For example in the following program the variable x is used in different blocks (here blocks consist of function body) without any conflict.

```
/*P14.3 Program to understand automatic variables*/
#include<stdio.h>
main()
{
    int x=5;
    printf("x = %d\t",x);
    func();
}
func()
{
    int x=15;
    printf("x = %d\n",x);
}
```

Output:

```
x = 5      x = 15
```

Here the variable x declared inside main() is different from the variable x declared inside the function func().

In the next program, there are different blocks inside main(), and the variable x declared in different blocks is different.

```
*P14.4 Program to understand automatic variables*
#include<stdio.h>
main()
{
    int x=3;
    printf("%d\t",x);
    {
        int x=10;
        printf("%d\t",x);
    }
    {
        int x=26;
        printf("%d\t",x);
    }
    printf("%d\n",x);
}
```

Output:

3 10 26 3

14.2.2 External

The variables that have to be used by many functions and different files can be declared as external variables. The initial value of an uninitialized external variable is zero.

Before studying external variables, let us first understand the difference between their definition and declaration. The declaration of an external variable declares the type and name of the variable, while the definition reserves storage for the variable as well as behaves as a declaration. The keyword **extern** is specified in declaration but not in definition. For example the definition of an external variable salary will be written as-

float salary;

Its declaration will be written as-

extern float salary;

The following points will clarify the concept of definition and declaration of an external variable.

Definition of an external variable-

- Definition creates the variable, so memory is allocated at the time of definition.
- There can be only one definition.
- The variable can be initialized with the definition and initializer should be constant.
- The keyword **extern** is not specified in the definition. Some compilers may allow the use of **extern** but it is better to omit this keyword in the definition.
- The definition can be written only outside functions.

Declaration of an external variable-

- The declaration does not create the variable, it only refers to a variable that has already been created somewhere, so memory is not allocated at the time of declaration.

- (b) There can be many declarations.
- (c) The variable cannot be initialized at the time of declaration.
- (d) The keyword extern is always specified in the declaration.
- (e) The declaration can be placed inside functions also.

Consider this program:

```
#include<stdio.h>
int x=8;
main()
{
    .....
}
func1()
{
    .....
}
func2()
{
    .....
}
```

In this program the variable x will be available to all the functions, since an external variable is active from the point of its definition till the end of a program. Now if we change the place of definition of variable x like this-

```
#include<stdio.h>
main()
{
    .....
}
func1()
{
    .....
}
int x=8;
func2()
{
    .....
}
```

Now x is defined after main() and func1(), so it can't be used by these functions, it is available only to function func2(). Suppose we want to use this variable in function main() then we can place declaration in this function like this-

```
#include<stdio.h>
main()
{
    extern int x;
    .....
}
func1()
{
    .....
}
```

```

}
int x=8;
func2( )
{
    .....
}

```

Now x will be available to functions func2() and main().

Till now we had written our program in a single file. When the program is large it is written in different files and these files are compiled separately and linked together afterwards to form an executable program. Now we'll consider a multifile program which is written in three files viz. first.c, second.c and third.c.

first.c

```

int x = 8;
main()
{
    .....
}
func1()
{
    .....
}

```

second.c

```

func2()
{
    .....
}
func3()
{
    .....
}

```

third.c

```

func4()
{
    .....
}
func5()
{
    .....
}

```

Here in the file first.c, an external variable x is defined and initialized. This variable can be used both in main() and func1() but it is not accessible to other files. Suppose file second.c wants to access this variable then we can put the declaration in this file as-

first.c

```

int x = 8;
main()
{
    .....
}
func1()
{
    .....
}

```

second.c

```

extern int x;
func2()
{
    .....
}
func3()
{
    .....
}

```

third.c

```

func4()
{
    .....
}
func5()
{
    .....
}

```

Now this variable can be accessed and modified in files first.c and second.c and any changes made to it will be visible in both files. If this variable is needed by only one function in the file second.c, then the extern declaration can be put inside that function.

So the declaration of an external variable using extern keyword is used to extend the scope of that variable.

Suppose our program consists of many files and in file first.c, we have defined many variables that may be needed by other files also. We can put an extern declaration for each variable in every file that needs it. Another better and practical approach is to collect all extern declarations in a header file and include that header file in the files, which require access to those variables.

14.2.3 Static

There are two types of static variables-

- (1) Local static variables
- (2) Global static variables

14.2.3.1 Local Static Variables

The scope of a local static variable is same as that of an automatic variable i.e. it can be used only inside the function or block in which it is defined. The lifetime of a static variable is more than that of an automatic variable. A static variable is created at the compilation time and it remains alive till the end of a program. It is not created and destroyed each time the control enters a function/block. Hence a static variable is created only once and its value is retained between function calls. If it has been initialized, then the initialization value is placed in it only once at the time of creation. It is not initialized each time the function is called.

A static variable can be initialized only by constants or constant expressions. Unlike automatic variables, we can't use the values of previously initialized variables to initialize static variables. If a static variable is not explicitly initialized then by default it takes initial value zero.

```
int x = 8;
int y = x; /*Valid*/
static int z = x; /*Invalid, initializer should be constant*/
```

Let us take a program to understand the use of local static variables. This program is similar to P14.2 but here x and y are declared as static variables.

```
/*P14.5 Program to understand the use of local static variables*/
#include<stdio.h>
main()
{
    func();
    func();
    func();
}
func()
{
    static int x=2,y=5;
    printf("x=%d,y=%d\n",x,y);
    x++; y++;
}
```

Output:

```
x = 2, y = 5
x = 3, y = 6
x = 4, y = 7
```

Note that the effect of initialization is seen only in the first call. In subsequent calls initialization is no

performed and variables x and y contain values left by the previous function call. The next program also illustrates the use of local static variables.

```
/*P14.6 Program to understand the use of local static variable */
#include<stdio.h>
main()
{
    int n,i;
    printf("Enter a number :");
    scanf("%d",&n);
    for(i=1;i<=10;i++)
        func(n);
    printf("\n");
}
func(int n)
{
    static int step; /* Automatically initialized to 0 */
    step=step+n;
    printf("%d\t",step);
}
```

Output:

Enter the number : 4

4	8	12	16	20	24	28	32	36	40
---	---	----	----	----	----	----	----	----	----

The next program uses a recursive function to find out the sum of digits of a number. The variable sum taken inside function sumd() should be taken as static.

```
/*P14.7 Program to find out the sum of digits of a number recursively*/
#include<stdio.h>
int sumd(int num);
main()
{
    int num;
    printf("Enter a number : ");
    scanf("%d",&num);
    printf("Sum of digits of %d is %d\n",num,sumd(num));
}
int sumd(int num)
{
    static int sum=0;
    if(num>0)
    {
        sum=sum+(num%10);
        sumd(num/10);
    }
    return sum;
}
```

14.2.3.2 Global Static Variables

If a local variable is declared as static then it remains alive throughout the program. In the case of

global variables, the static specifier is not used to extend the lifetime since global variables have already a lifetime equal to the life of program. Here the static specifier is used for information hiding. If an external variable is defined as static, then it can't be used by other files of the program. So we can make an external variable private to a file by making it static.

first.c

```
int x = 8;
static int y = 10;
main()
{
    .....
}
func1()
{
    .....
}
```

second.c

```
extern int x;
func2()
{
    .....
}
func3()
{
    .....
}
```

third.c

```
func4()
{
    .....
}
func5()
{
    .....
}
```

Here the variable y is defined as a static external variable, so it can be used only in the file first.c. We can't use it in other files by putting extern declaration for it.

14.2.4 Register

Register storage class can be applied only to local variables. The scope, lifetime and initial value of register variables are same as that of automatic variables. The only difference between the two is in the place where they are stored. Automatic variables are stored in memory while register variables are stored in CPU registers. Registers are small storage units present in the processor. The variables stored in registers can be accessed much faster than the variables stored in memory. So the variables that are frequently used can be assigned register storage class for faster processing. For example the variables used as loop counters may be declared as register variables since they are frequently used.

```
/*P14.8 Program to understand the use of register variable*/
#include<stdio.h>
main( )
{
    register int i;
    for(i=0;i<20000;i++)
        printf("%d\n",i);
}
```

Register variables don't have memory addresses so we can't apply address operator(&) to them. There are limited number of registers in the processor hence we can declare only few variables as register. If many variables are declared as register and the CPU registers are not available then compiler will treat them as automatic variables.

The CPU registers are generally of 16 bits, so we can specify register storage class only for int, char or pointer types. If a variable other than these types is declared as register variable then compiler treats it as an automatic variable.

The register storage class specifier can be applied to formal arguments of a function while the other three storage class specifiers can't be used in this way.

14.3 Storage Classes in Functions

The storage class specifiers **extern** and **static** can be used with function definitions. The definition of a function without any storage specifier is equivalent to its definition with the keyword **extern** i.e. by default the definition of a function is considered external. If a function is external then it can be used by all the files of the program and if it is static then it can be used only in the file where it is defined.

If an external function is to be used in another file, then that file should contain function declaration and it is a good practice to specify **extern** in that declaration.

first.c

```
main( )
{
    .....
}
float func1(int )
{
    .....
}
```

second.c

```
extern float func1(int );
func2()
{
    .....
}
static int func3(int )
{
    .....
}
```

third.c

```
void func4()
{
    .....
}
void func5()
{
    .....
}
```

Here the function **func1()** is defined in file **first.c**. Its declaration is put in file **second.c**, so it can be used in this file also. The function **func3()** in file **second.c** is defined as static so it can't be used by any other file. Generally declarations of all functions are collected in a header file and that header file is included in other files.

14.4 Linkage

There are three types of linkages in C-

- External linkage
- Internal linkage
- No linkage

Local variables have no linkage, so their scope is only within the block where they are declared. Global variables and functions have external linkage, so they can be used in any file of the program. Static global variables and static functions have internal linkage, so their scope is only in the file where they are declared.

14.5 Memory During Program Execution

The register variables are stored in CPU registers and rest of the variables are stored in memory. Now let us see how memory is organized when a C program is run.

Code Area: This area is used to store executable code of the program. The size of this area does not change during run time.

Data Area: This area stores static and global variables. It is further subdivided into two areas viz. initialized data area and uninitialized data area. The initialized data area stores all the initialized static and global

variables while the uninitialized data area stores all the uninitialized static and global variables. The reason for division of this area is that all uninitialized variables can be collectively assigned value zero. The size of data area is also fixed and does not change during run time.

Code	Code of the program
Initialized Data	Initialized static variables Initialized global variables
Uninitialized Data	Uninitialized static variables Uninitialized global variables
Heap	Dynamically allocated memory
Stack	Initialized and uninitialized automatic variables

Memory during execution of the program

Heap: This area is used for dynamically allocated memory. It is the responsibility of the program to allocate memory from the heap. The size of this area is dynamic i.e. it may change during run time.

Stack: Automatic variables are stored in this area. The size of this area keeps on changing during time.

The following table summarizes all the features of storage classes-

Keyword	Place of declaration	Lifetime	Scope	Initial Value	Place of storage	Linkage	Initialization
auto (or none)	Inside Function /Block	Function /Block	Function /Block	Garbage	Memory (stack)	None	Constant Variable
register	Inside Function /Block	Function /Block	Function /Block	Garbage	Registers	None	Constant Variable
static (local)	Inside Function /Block	Program	Function / Block	Zero	Memory (data area)	None	Constant Variable
(none)	Outside function (Definition of external)	Program	Definition to end of file (can be shared in other files)	Zero	Memory (data area)	External	Constant Variable

extern	variable) Outside or inside function (Declaration of external variable)	Program	using extern declaration) Declaration to end of file / Function	-	Memory (data area)	External	Can't be initialized
static (global)	Outside function	Program	Definition to end of file (can't be shared in other files using extern declaration)	Zero	Memory (data area)	Internal	Constant

14.6 const

If any variable is declared with const qualifier, then the value of that variable can't be changed by the program. The const qualifier can occur in the declaration before or after the data type. For example-

```
const int x = 9;
int const x = 9;
```

Both these declarations are equivalent and declare x as a const variable. Any attempt to change the value of this variable in the program will result in an error. For example these statements are invalid-

```
x = 10;           /*Invalid*/
x = func( );      /*Invalid*/
x++;             /*Invalid*/
```

The const qualifier informs the compiler that the variable can be stored in read only memory. A const variable can be given a value only through initialization or by some hardware devices. Note that the value of a const variable can't be modified by the program, but any external event outside the program can change its value.

If an array, structure or union is declared as const then each member in it becomes constant. For example-

```
const int arr[5] = { 10, 11, 12, 13, 14 };
const struct { char x; int y; float z; }var ={ 'A', 12, 29.5 };
arr[2] = 22;          /*Invalid*/
var.x = 'B';          /*Invalid*/
```

The const variables are not true compile time constants in C. So we can't use const variables where constant expressions are required e.g. in array dimensions or case labels.

For example the following code is invalid in C.

```
const int size=10;
int arr[size];
```

The const qualifier can be useful while passing array arguments to functions. We know that when an array is passed as an argument, the function gets access to the original array and can modify it. If we don't want the function to make any changes in the array then we can declare the array as const in the formal parameter list.

```
func( const char arr[] )
{
    .....
}
```

Now we'll see how to use const in pointer declarations. We can declare three types of pointers using the qualifier const:

- (i) Pointer to const data
- (ii) const pointer
- (iii) const pointer to const data

Consider these declarations-

```
const int a = 2, b = 6;
const int *p1 = &a; /* or int const *p1 = &a; */
```

Here p1 is declared as a pointer to const integer. We can change the pointer p1 but we can't change the variable pointed to by p1.

```
*p1 = 9; /*Invalid*/
p1 = &b; /*Valid since p1 is not a constant itself*/
```

Now consider these declarations-

```
int a = 2, b = 6;
int *const p2 = &a;
```

Here p2 is declared as a const pointer. We can't change the pointer variable p2, but we can change the variable pointed to by p2.

```
*p2 = 9; /*Valid*/
p2 = &b; /*Invalid since p2 is a constant*/
```

Now consider these declarations-

```
const int a=2, b=6;
const int const *p3=&a;
```

Here p3 is declared as a const pointer to const integer. We can neither change the pointer variable p3 nor the variable pointed to by it.

```
*p3 = 9; /*Invalid*/
p3 = &b; /*Invalid*/
```

So the three different types of declarations of pointers using const are-

int const *ptr;	or	const int *ptr;	/* pointer to const integer */
int *const ptr;			/* const pointer to an integer */
const int *const ptr;			/* const pointer to a const integer */

The following example will further clarify this concept.

```
char str1[] = "wealthy";
```

```

char str2[]="strong";
const char *pc=str1;
char *const cp=str1;
const char *const cpc=str1;
                                /*pointer to const*/
                                /*const pointer*/
                                /*const pointer to const*/

pc=str2;          /*Valid*/
cp=str2;          /*Invalid*/
cpc=str2;         /*Invalid*/

*pc='h';    /*Invalid*/
*cp='h';    /*Valid*/
*cpc='h';   /*Invalid*/

```

If we want to assign the address of a const variable to a non const pointer, then we'll have to use cast operator. For example-

```

const int a = 10;
int *ptr;
ptr = (int *)&a;

```

Now if we try to change the value of variable a indirectly through ptr then the result is undefined and there can be a run time error.

```
*ptr = 12;      /* undefined*/
```

14.7 volatile

Whenever a variable is encountered in the program, compiler reads the value of that variable from the memory. But sometimes for optimization purposes the compiler stores the value of the currently used variable in any unused register. Now when that variable is encountered next time in the program and the compiler sees that the program has not changed value of the variable, it reads the value from the register instead of the memory. This process saves time since accessing a register is faster than accessing memory.

This sort of automatic optimization by the compiler may sometimes lead to incorrect results. This generally happens when the value of a variable can be modified by some external process outside the program. These types of situations arise when we have memory mapped I/O, variables shared among multiple processes or variables that can be modified by interrupt routines.

For example suppose we have a variable time that represents current system time and its value is controlled by the system clock. Now consider this loop-

```

while( time != T )
    /* Do nothing till value of variable time equals T*/

```

The optimizing phase of the compiler may observe that the value of variable time is not changing inside the loop, so it may decide to access its value once from the memory and then cache this value in a register. Now for each iteration of the loop, compiler reads this cached value from the register and hence the loop will never terminate, but this is not what we wanted. It is our responsibility to inform the compiler that the value of a particular variable may change through some external process also.

To solve these types of problems we can use volatile qualifier. If a variable is declared with the qualifier volatile, then we are instructing the compiler to turn off the optimization process for that variable i.e. we are forcing the compiler to read the value of that variable from memory only, each time it is

encountered in the program. The optimization phase of compiler will never try to cache the value of a volatile variable in a register.

The value of a volatile variable can be changed from inside the program. If we don't want this to happen we can use the qualifier const along with the qualifier volatile. For example-

```
const volatile int x;
```

If an array, structure or union is declared as volatile then each member becomes volatile.

14.8 Functions With Variable Number Of Arguments

In the functions that we had created till now, the number and data type of arguments was fixed at the time of function definition. In some situations we may need functions that can accept variable number of arguments of different types. The library functions printf() and scanf() are examples of these type of functions. We have already used these functions many times with different number and type of arguments. We can also create our own functions that can accept variable number of arguments. These types of functions are also known as variadic functions.

The header file stdarg.h provides the facilities needed to define functions with variable number of arguments. This file defines a new type called va_list and three macros va_start, va_arg, va_end that can operate on variables of this new type.

Type

va_list - Used to declare argument pointer variables.

Macros

- va_start - Initializes the argument pointer variable.
- va_arg - Retrieves the current argument and increments the argument pointer.
- va_end - Assigns NULL to argument pointer.

A function that accepts variable number of arguments should be defined with ellipsis(...) at the end of argument list. The ellipsis should occur only at the end of argument list and there should be at least one fixed argument. For example-

```
func( char *str, int num, ... )
{
    .....
    .....
}
```

Here func() takes two fixed arguments viz. str, num and after that it can accept any number of arguments. For example all these calls of func() are valid-

```
func("Chennai", 40, 67.89, 'P', "Madras", 23, 67 );
func("Lucknow", 35, 66 );
func("Bareilly", 30, 'x', 20, 39, 12.5);
```

In all these calls first two arguments are always a string and an integer, rest of the arguments can be of any type. Now inside the function definition we can access the fixed arguments using their names but the remaining arguments don't have a name so they are accessed using the macros defined in stdarg.h file. This is why fixed arguments are known as named arguments and variable number of arguments are known as unnamed arguments. Now we'll see how we can access these unnamed arguments inside the function body.

Initially we'll declare a variable of type va_list.

```
va_list ap;
```

This variable is conventionally named ap. Here ap is known as argument pointer and will be used to point to the unnamed arguments. The macro va_start initializes ap and makes it point to the first unnamed argument passed to the function. This macro takes two arguments, first one is the argument pointer ap and second one is the name of the last fixed argument passed to the function (i.e. argument which is just before the ellipsis). For example in the above function func(), va_start will be called as-

```
va_start( ap, num);
```

Now we can access individual variable arguments sequentially by using va_arg. This macro takes ap and the data type of the current argument.

```
arg = va_arg( ap, datatype);
```

It returns the value of the current argument and increments the pointer ap so that it points to the next argument. After calling va_start, the first call of va_arg returns first unnamed argument, second call returns second unnamed argument and so on. If datatype of the current unnamed argument in the function call does not match with the datatype in va_arg then the behaviour is undefined.

The macro va_end should be called before exiting from the function. This macro sets the argument pointer to NULL.

```
va_end(ap);
```

The unnamed arguments can't be used after calling va_end. If we want to use those arguments then once again we'll have to initialize ap with va_start.

The whole procedure is summarized in these steps-

- (i) Include the header file stdarg.h
- (ii) The function header should contain ellipsis to denote the variable argument list.
- (iii) Declare a variable of type va_list.
- (iv) Initialize this argument pointer using va_start, so that it points to the first unnamed argument.
- (v) Use va_arg to retrieve the value of arguments.
- (vi) Call the macro va_end when you have finished working with these arguments,

There is no facility to count how many arguments were passed in the function call and what was the type of each argument. It is programmer's responsibility to pass this information to the function through fixed arguments. For example we may decide to take the first fixed argument as an integer that represents the total number of unnamed arguments. Generally a format string is passed as a fixed argument, which contains information about the type of each argument. For example printf() uses the format string that contains conversion specifications which denote the data type of each unnamed argument.

Now we'll take an example program and apply all these concepts. In this program we will make a function sum() which returns the sum of integers passed to it.

```
/*P14.9 Program to find out the sum of integers */
#include<stdio.h>
#include<stdarg.h>
int sum(int , ...);
main()
{
    printf("Total = %d\n", sum(2, 99, 68));
}
```

```

        printf("Total = %d\n", sum(3,11,79,32));
        printf("Total = %d\n", sum(5,23,34,45,56,78));
    }
int sum(int num, ...)
{
    int i;
    va_list ap;
    int arg, total=0;
    va_start(ap,num);
    for(i=0;i<num;i++)
    {
        arg=va_arg(ap,int);
        printf("%d ",arg);
        total+=arg;
    }
    va_end(ap);
    return total;
}

```

Output :

```

99 68 Total = 167
11 79 32 Total = 122
23 34 45 56 78 Total = 236

```

Here we have called the function sum with different number of arguments. We have taken only one fixed argument num that represents the number of unnamed arguments passed to the function. It is used in the loop to step through the unnamed arguments. Inside the function, firstly a variable of va_list type is declared. Then va_start is called and num is passed to it since it is the last fixed argument. After that each variable argument is obtained by calling va_arg repeatedly inside a loop.

In the above program we have taken variable number of arguments but all of them were of the same type. Now we'll take another program in which the unnamed arguments are of different types. The working of this function is somewhat similar to that of printf() function. In this function we'll see only one fixed argument and that will be a format string. Inside the function we'll scan this format string to know the data type of the unnamed argument. The text in the format string is printed through putchar(). We have printed character values and string values using putchar() and fputs() respectively. The integer and float values are first converted to strings and then printed using fputs(). We have used i_to_str() function that converts an integer to a string and f_to_str() function that converts a float to a string. The definitions of these functions are given in chapter string (program P9.31). This printf() function is a bit smarter than printf(), since it can print the binary equivalent of an integer also.

```

/*P14.10 Program that uses a function similar to printf()*/
#include<stdarg.h>
#include<stdio.h>
void i_to_str(int num,char str[],int b);
void f_to_str(float num,char str[]);
void print(char *format, ... );
main()
{
    int a=125;
    float b= 563.66;

```

```
char c='Q';
char *name="Ranju";
print("Value of a in binary = %b\n",a);
print("Value of a in octal = %o\n",a);
print("Value of a in decimal = %d\n",a);
print("Value of a in hexadecimal = %x\n",a);
print("Value of b = %f\n",b);
print("Value of c = %c\n",c);
print("Value of name = %s\n",name);
}
void print(char *format, ...)
{
    char *p,*str;
    int k;
    float l;
    va_list ap;
    va_start(ap,format);

    for(p=format;*p]!='\0';p++)
    {
        if((*p)=='%')
        {
            p++;
            switch(*p)
            {
                case 'b':
                    k=va_arg(ap,int);
                    i_to_str(k,str,2);
                    fputs(str,stdout);
                    break;
                case 'd':
                    k=va_arg(ap,int);
                    i_to_str(k,str,10);
                    fputs(str,stdout);
                    break;
                case 'o':
                    k=va_arg(ap,int);
                    i_to_str(k,str,8);
                    fputs(str,stdout);
                    break;
                case 'x':
                    k=va_arg(ap,int);
                    i_to_str(k,str, 6);
                    fputs(str,stdout);
                    break;
                case 'c':
                    k=va_arg(ap,int);
                    putchar(k);
                    break;
                case 's':
                    str=va_arg(ap,char *);

```

```

        fputs(str, stdout);
        break;
    case 'f':
        l=va_arg(ap, double);
        f_to_str(l, str);
        fputs(str, stdout);
        break;
    case '%':
        putchar('%');
        break;
    }
}
else
putchar(*p);
}
}

```

Here we have used int in va_arg to extract a char value and double to extract a float value. This is because in the case of variable number of arguments the default argument promotions are applied i.e. arguments of type char and short int are promoted to int and arguments of type float are converted to double. So use int for extracting a char, short int or int and use double to extract a float or double. We have taken the function print to be of void type, if you want you can return the number of characters output as in printf().

The functions i_to_str() and f_to_str() can't handle negative numbers, so before calling these functions we'll have to write an additional condition for negative numbers like this-

```

if ( k<0)
{
    k = -k; putchar('-'); }

```

14.8.1 Passing Variable Number of Arguments To Another Function

Suppose we write a function func1() that takes variable number of arguments and we want to pass all these unnamed arguments to another function func2(). For this the function func2() should be declared such that it can accept a variable of va_list type.

```

func1(int num, ...)
{
    va_list ap;
    va_start(ap, num);
    func2(num, ap);
    .....
    .....
}

func2(int num, va_list ap)
{
    .....
    .....
}

```

So here inside func2() there is no need to declare a variable of va_list type and call to macro va_start is also not needed. The variable ap can be used in the macro va_arg to retrieve the arguments. But in func2() we can use the unnamed arguments only once, it is not possible to call va_start again as

retraverse the variable argument list. Now we'll take a simple example and see how this can be done. In the program P14.9 we had made a function sum() that displays the integers and returns their total. Now we'll make a separate function display() for displaying the numbers and will send the initialized argument pointer to this function.

```
/*P14.11 Program in which the variable length list is passed to another
function*/
#include<stdio.h>
#include<stdarg.h>
int sum(int , ...);
void display(int,va_list);
main()
{
    printf("Total= %d\n",sum(2,99,68));
    printf("Total =%d\n",sum(3,11,79,32));
    printf("Total= %d\n",sum(5,23,34,45,56,78));
}
int sum(int num, ...)
{
    int i;
    va_list ap;
    int arg,total=0;
    va_start(ap,num);
    display(num,ap);
    for(i=0;i<num;i++)
    {
        arg=va_arg(ap,int);
        total+=arg;
    }
    va_end(ap);
    return total;
}
void display(int num,va_list ap)
{
    int i,arg;
    for(i=0;i<num;i++)
    {
        arg=va_arg(ap,int);
        printf("%d ",arg);
    }
}
```

The library function vprintf is one such function that accepts a variable of type va_list. It is just like printf function except that it takes a variable of va_list type instead of variable argument list. The prototypes of printf and vprintf are-

```
int printf ( const char *format [, argument, ...] );
int vprintf ( const char *format, va_list arglist);
```

Similarly vfprintf and vsprintf are analogous to fprintf and sprintf, except for this single difference.

```
int fprintf (FILE *stream, const char *format [, argument, ...]);
```

```
int vfprintf(FILE *stream, const char *format, va_list arglist);
int sprintf (char *buffer, const char *format [, argument, ...]);
int vsprintf(char *buffer, const char *format, va_list arglist);
```

Now we'll write a debugging function error() that will display the error message along with file name, function name and line number where the error occurred. This function won't be much useful if it takes only a fixed number of arguments of specific types because at different times the error message would have to display values of different variables. So we'll define it as a variadic function. The first four arguments are fixed in which the first three represent filename, function name and line number respectively. The fourth argument is a format string which represents the error message and is similar to the format string of printf() function. After this we can have variable number of arguments which will be written along with the error message, according to the conversion specifications(%d, %s etc) present in the format string. The definition of error() function can be written as-

```
error(char *file,char *fname,int line,char *format, ...)
{
    va_list ap;
    va_start(ap,format);
    printf("ERROR : ");
    printf("File - %s , ", file);
    printf("Function - %s , ", fname);
    printf("Line - %d\n", line);
    vprintf(format,ap);
    printf("\n");
}
```

Here we have used vprintf to print the error message along with the variable number of arguments. Now suppose this function is invoked in function myfunc() of file myfile.c, at line 44 like this-

```
myfunc( )
{
    .....
    error(__FILE__,"myfunc",__LINE__,"Value of a must be less than
    %d and greater than %d\n",100,5);
    .....
}
```

The output will be this-

```
ERROR : File - c:\dir\myfile.c , Function - myfunc , Line - 44
Value of a must be less than 100 and greater than 5
```

If we want to send the output to the stream stderr, then we can use fprintf and vfprintf instead of printf and vprintf in the definition of error function.

14.9 lvalue and rvalue

Sometimes we get compiler errors that use the terms lvalue and rvalue. An expression that can occur on the left hand side of assignment operator is known as lvalue and an expression that can occur only on right hand side of assignment operator is known as rvalue.

Some examples of rvalues are-

x+y*z	
monday	(here monday is an enum constant)
&x	(here x is an integer variable)
arr	(here arr is the name of an array)

The value of these expressions can be used in the program, but they can't be modified, we can't write them on the left side of assignment operator. We will get error 'lvalue required' if we try to write statements like this-

```
5 = x;
x+y*z = 12;
&x = p;
arr = ptr;
```

Some examples of lvalues are-

var	(var is an integer variable)
ptr	(ptr is a pointer)
*ptr	(ptr is a pointer variable)
emp	(emp is a structure variable)
emp->salary	(here emp is a structure variable and salary is its member)
arr[2]	(arr is a 1-d array)

All these expressions can occur both on the left side and right side of assignment operator i.e. we can use their values in the program and their value can be modified.

14.10 Compilation And Execution of C Programs

There are different phases through which our program passes before being transformed into an executable form. The following figure shows these phases-

14.10.1 Preprocessor

The source code is the code that we write using any text editor and the source code file is given an extension '.c'. This source code is firstly passed through the preprocessor. The preprocessor expands this code and passes it to the compiler. We have discussed the role of preprocessor in chapter 12.

14.10.2 Compiler

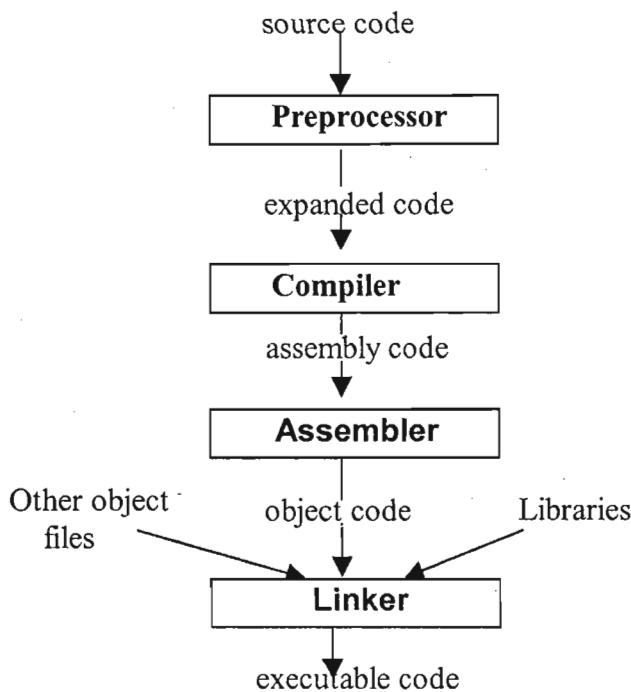
The expanded code is passed to the compiler. The compiler converts this code into the machine's assembly language code.

14.10.3 Assembler

This assembly language is converted to object code by the system's assembler. The name of object file is same as that of source file. In DOS the object file has extension '.obj' and in UNIX the extension is '.o'.

14.10.4 Linker

Generally all programs written in C, use library functions. Library functions are precompiled and their object code is stored in library files with '.lib'(or '.a') extension. The linker combines this object code of the library functions with the object code of our program. Our program may also contain references



to functions that are defined in other files. The linker links the object code of these files also to our program. So the job of the linker is to combine the object code of our program with the object code of other files and object code of library functions. The output of the linker is an executable file. In DOS the executable file has same name as that of source code file and has extension '.exe', and in UNIX the executable file is named as a.out or the name of output with -o option.

Exercise

Assume stdio.h is included in all programs.

```
(1) enum month{jan,feb,mar,apr,may};
main()
{
    enum month m;
    m=++feb;
    printf("%d\n", m);
}
```

```
(2) enum day{sun=1,mon,tue,wed};
main()
{
    enum day d1;
    printf("%d\t",mon);
    d1=mon+2;
    printf("%d\n",d1);
}
```

```
(3) struct tag{
```

Miscellaneous Features in C

```

        auto int x;
        static int y;
    };
main( )
{
    struct tag s;
    s.x=4;
    s.y=5;
}

(4) int var=6;
main( )
{
    int var=18;
    printf("%d",var);
}

(5) main( )
{
    int i,sum=0;
    for(i=0;i>5;i++)
    {
        int i=10;
        sum=sum+i++;
    }
    printf("sum = %d",sum);
}

(6) main( )
{
    int i,sum=0;
    for(i=0;i>5;i++)
    {
        static int i=10;
        sum=sum+i++;
    }
    printf("sum = %d",sum);
}

(7) int x=89;
main( )
{
    func1(x);
    printf("%d\t",x);
    func2();
    printf("%d\n",x);
}
func1(int x)
{ x++; }
func2()
{ x++; }

```

```

(8) int x=2;
     static int y=5;
main()
{
    int x=3;
    func();
    func();
    printf("Inside main() : x=%d,y=%d\n",x,y);
}
func()
{
    static int x;
    x=x+2;
    printf("Inside func() : x=%d,y=%d\n",x,y);
}

(9) main()
{
    func1();
    func2();
}
func1()
{
    extern int x;
    x++;
    printf("%d\t",x);
}
int x=89;
func2()
{
    x++;
    printf("%d\n",x);
}

(10)main()
{
    const int *ptr=func();
    *ptr = 7;
    printf("*ptr=%d",*ptr);
}
func()
{
    int *p=malloc(1*sizeof(int));
    return p;
}

(11)main()
{
    char str1[]="hockey";
    char str2[]="Cricket";
    char const *p=str1;
}

```

```
*p='j';
p=str2;
}

(12)main()
{
    char str1[]="hockey";
    char str2[]="Cricket";
    char *const p=str1;
    *p='j';
    p=str2;
}

(13)main()
{
    int a[]={1,2,3,4};
    int b[]={5,6,7,8};
    int c[]={9,10,11,12};
    func(a,b,c);
}
func(int a[],const int b[],int c[])
{
    a=c;
    a[0]++;
    b=c;
    b[0]++;
}

(14)main()
{
    const int i=23;
    const int j=thrice(i);
    printf("j=%d\n",j);
}
thrice(int i)
{
    return 3*i;
}

(15)main()
{
    int i=3;
    const int j=i;
    const int k=func();
    int *ptr=&k;
    const int m=*ptr;
    printf("%d\t%d\t%d\t%d\n", i, j, k, m);
}
func()
{
    return 4;
}
```

```
(16)int *func();
main()
{
    int i, size;
    const int *arr=func(&size);
    for(i=0;i<size;i++)
    {
        printf("Enter a[%d] : ", i);
        scanf("%d", &arr[i]);
    }
    for(i=0;i<size;i++)
        printf("%d\t", arr[i]);
}
int *func(int *psize)
{
    int *p;
    printf("Enter size : ");
    scanf("%d", psize );
    p=malloc(*psize*sizeof(int));
    return p;
}

(17)#include<stdarg.h>
void func(int n,...);
main( )
{
    int count=4;
    func(count,2,3,4,5);
}
void func(int n,...)
{
    va_list ap;
    int a,i;
    for(i=0;i<n;i++)
    {
        a=va_arg(ap,int);
        printf("%d ",a);
    }
}

(18)#include<stdarg.h>
main( )
{
    int a=2,b=3,c=4,d=5;
    func(4,2,3,8,5);
}
func(int a, ... ,int b)
{
    va_list *ap;
    va_start(ap, a);
    for(i=0;i<b;i++)
```

```

        printf("%d", va_arg(ap,int));
    va_end:
}

(19)main()
{
    int x=6;
    ++x++;
    printf("%d\n",x);
}

```

Answers

(1) Error : lvalue required

(2) 2 4

(3) Error : storage specifiers can't be used inside definition of structure templates.

(4) 18

Whenever there is a conflict between a local and global variable of the same name the local variable gets the priority.

(5) sum = 0

(6) sum = 0

(7) 89 90

(8) Inside func() : x = 2, y = 5

Inside func() : x = 4, y = 5

Inside main() : x = 3, y = 5

(9) 90 91

(10) Error: can't modify a const object

(11) Error: can't modify a const object

Here p=str2 is valid , but *p='j' is invalid.

(12) Error: can't modify a const object

Here p=str2 is invalid, but *p='j' is valid

(13) Error: can't modify a const object

Both the statements b=c; and b[0]++; are invalid.

(14) 69

(15) 3 3 4 4

(16) Enter size : 3

Enter a[0] : 5

Enter a[1] : 6

Enter a[2] : 7

5 6 7

(17) Since ap was not initialized by va_start, so it may be pointing anywhere in memory and so we'll get garbage values.

(18) Ellipsis should always be at the end of argument list, and another mistake is that an asterisk is used while declaring argument pointer

(19) Error: lvalue required

Chapter 15

Building project and creation of library

We have studied about C language, now we'll come to the real life problems and see how we can solve them step by step. Here we will use our C program to develop one real life project with whole software life cycle of project. This will give you an idea of developing a full project in C. We will develop this project in Turbo C as well as in Unix. Also we will see how to develop a library and use the functions of this library in different programs.

The project that we are going to make is related with date. Date related operations are of great importance in many applications. Most of the application projects require date related manipulations. So our aim is to develop a project in C, which will perform all date related manipulations. Now we will go step by step in different phases of software development life cycle for developing this project. These phases are as-

1. Requirement analysis
2. Top level design
3. Detail design
4. Coding
5. Testing

15.1 Requirement Analysis

First step is requirement analysis. Here we think about requirement of our project by analyzing different manipulation of dates. We know very well what are the operations that are needed with date, as well as different attributes of date. So after analyzing we came up with a document that gives all the requirement of the project. This is called as SRS(Software Requirement Specification).

1. Given date is valid or invalid.
2. Day of week from a given date.
3. Comparing two dates.
4. Difference of two dates in days.
5. Difference of two dates in years, months and days.
6. Add years to a given date.
7. Subtract years from a given date.
8. Add months to a given date.
9. Subtract months from a given date.
10. Add days to a given date.
11. Subtract days from a given date.

15.2 Top Level Design

Now we will go to the second step that is top-level design. Here we will divide the whole project in different modules. So whenever any change is needed, it will affect only one particular module and not others. Now we will write different modules in different files. All the modules will be implemented in different files as-

1 Datefmt.c

This file will contain modules to convert a date from string form to integer variables and vice versa.

2 Valid.c

This file will contain module to check whether a given date is valid or not.

3 Leap .c

This file will contain module to check whether a given year is leap or not.

4 Julian.c

This file will contain module for getting julian number and also converting julian number to months and days from a given date.

5 Weekday.c

This file will contain module for getting day of week for a given date.

6 Cmpdate.c

This file will contain module for comparing two dates.

7 Diffymd.c

This file will contain module to find out difference in years, months and days for two dates.

8 Diffdays.c

This file will contain module to find out difference in days for two dates.

9 Addyear.c

This file will contain module for adding years to a given date.

10 Subyear.c

This file will contain module for subtracting years from a given date.

11 Addmonth.c

This file will contain module for adding months to a given date.

12 Submonth.c

This file will contain module for subtracting months from a given date.

13 Adddays.c

This file will contain module for adding days to a given date.

14 Subdays.c

This file will contain module for subtracting days from a given date.

15.3 Detail Design

Now we will go to the next step that is detail design. Here we analyze all modules one by one for detail design. We analyze what functions will be required, what would be the input and return value of that function in each module and come up with one document for detail design, which contains the information of function prototype for each module. Let us see the function required in each module-

Note that date will be always in dd/mm/yyyy format.

1. Datefmt.c

void splitDate(char *date, int *y, int *m, int *d)

Input :	date	string in dd/mm/yyyy format
	y	integer pointer, will get value of year.
	m	integer pointer, will get value of month.
	d	integer pointer, will get value of day

Return value - None

void formDate(char *date, int y, int m, int d)

Input :	date	pointer to character for getting date in string form
	y	Integer that represents the year of given date
	m	Integer that represents the month of given date
	d	Integer that represents the day of given date

Return value - None

2. Valid.c

int isValid(char *date)

Input:	date	
Return Value	0	Date is not valid
	1	Date is valid

3. Leap.c

int isLeap(int year)

Input:	year	
Return Value:	0	Not a leap year
	1	Leap year

4. Julian.c

int julian(int d, int m, int y)

Input:	d	day from given date
	m	month from given date
	y	year from given date
Return Value:		integer value containing julian number.

void revJulian(int j, int *y, int *d, int *m)

Input:	j	contains julian no. for given year from given date
	y	year from given date
	d	integer value, will get day from julian number
	m	integer value, will get month from julian number
Return Value:	None	

5. Weekday.c

```
void weekDay(char *date, char *dayWeek)
```

Input: date Given date in dd/mm/yyyy format
 dayWeek pointer to character for getting day of week
 Return value: None

6. Cmpdate.c

```
int cmpDate( char *date1, char *date2)
```

Input: date1 First date in dd/mm/yyyy format
 date2 Second date in dd/mm/yyyy format

Return value: -1 date1 > date2
 0 date1 == date2
 1 date1 < date2

7. Diffymd.c

```
void diffYMD(char *date1, char *date2, int *y, int *m, int *d)
```

Input: date1 First date in dd/mm/yyyy format
 date2 Second date in dd/mm/yyyy format
 y integer pointer, will get years
 m integer pointer, will get months
 d integer pointer, will get days

Return Value: None

8. Diffdays.c

```
int diffDays( char *date1, char *date2)
```

Input: date1 First date in dd/mm/yyyy format
 date2 Second date in dd/mm/yyyy format

Return Value: integer value that contains difference of two dates in days

9. Addyear.c

```
void addYear(char *date, int iyear, char *newDate)
```

Input: date Given date
 iyear Number of years to be added
 newDate Character pointer, will get new date after addition
 Return Value: None

10. Subyear.c

```
void subYear(char *date, int dyear, char *newDate)
```

Input: date Given date
 dyear Number of years to be subtracted

Return Value: newDate Character pointer, will get new date after subtraction
 None

11. Addmon th.c

void addMonth(char *date, int imonth, char *newDate)
 Input: date Given date
 imonth Number of months to be added
 newDate Character pointer, will get new date after addition
 Return Value: None

12. Submonth.c

void subMonth (char *date, int dmonth, char *newDate)
 Input: date Given date
 dmonth Number of months to be subtracted
 newDate Character pointer, will get new date after subtraction
 Return Value: None

13. Adddays.c

void addDays(char *date, int days, char *newDate)
 Input: date Given date
 days Number of days to be added
 newDate Character pointer, will get new date after addition
 Return Value: None

14. Subdays.c

void subDays(char *date, int days, char *newDate)
 Input: date Given date
 days Number of days to be subtracted
 newDate Character pointer, will get new date after subtraction
 Return Value: None

15.4 Coding

Now we will go to the next step that is coding. Here we write logic for each function given in module and finally write the code in C language. The coding for all functions is given below.

We maintain one header file that contains all function prototypes and will be included in the file whenever that function is used.

15.4.1 Dtmanip.h

```
void formDate(char *date, int y, int m, int d);
void splitDate(char *date,int *y,int *m, int *d);
int isValid(char *date);
int isLeap(int y);
```

```

void addDays(char *date,int days,char *newDate);
void addMonth(char *date,int imonth,char *newDate);
void addYear(char *date,int iyear,char *newDate);
int cmpDate(char *date1,char *date2);
int diffDays(char *date1,char *date2);
void diffYMD(char *date1,char *date2,int *y,int *m,int *d);
void subDays(char *date,int days,char *newDate);
void subMonth(char *date,int dmonth,char *newDate);
void subYear(char *date,int dyear,char *newDate);
void weekDay(char *date,char *dayWeek);
void revJulian(int j,int y,int *d,int *m);
int julian(int d,int m,int y);

```

15.4.2 Datefmt.c

```

/*To split and form the date*/
void splitDate(char *date,int *y,int *m,int *d)
{
    date[2]=date[5]='\0';
    *d=atoi(date);
    *m=atoi(date+3);
    *y=atoi(date+6);
    date[2]=date[5]='/';
}/*End of splitDate()*/

void formDate(char *date,int y,int m,int d)
{
    char arr[9][3]={"01","02","03","04","05","06","07","08","09"};
    if(d<10)
        strcpy(date,arr[d-1]);
    else
        sprintf(date,"%d",d);
    if(m<10)
        strcpy(date+3,arr[m-1]);
    else
        sprintf(date+3,"%d",m);
    sprintf(date+6,"%d",y);
    date[2]=date[5]='/';
}/*End of formDate()*/

```

The function `splitDate()` is used to get the values of day, month, year in different integer variables. We have used the library function `atoi()` that converts a string to an integer. The first argument to `splitDate()` is a string that contains the date in the format dd/mm/yyyy.

The function `formDate()` is used to form a date in format dd/mm/yyyy, from the values of day, month, year. Here we have used the library function `sprintf()` to convert an integer to string.

15.4.3 Valid.c

```
/*To find whether a date entered is valid or not.*/
```

```

int isValid(char *date)
{
    int d,m,y;

    if(date[2]!='/') || date[5]!='/') || strlen(date)!=10)
        return 0;
    splitDate(date,&y,&m,&d);

    if( d==0 || m==0 || y==0)
        return 0;
    if( d<1 || d>31 || m<1 || m>12)
        return 0;

    if(m==2) /*check for number of days in February*/
    {
        if(d==29 && isLeap(y))
            return 1;
        if( d>28)
            return 0;
    }

    if( m==4 || m==6 || m==9 || m==11) /*Check for days in april, june,sept,
    nov*/
    {
        if(d>30)
            return 0;
    }

    return 1;
}/*End of isValid()*/

```

This function returns 1 if the date is valid, otherwise it returns zero. In this function, initially we check the format of the date. If the date is not in the format dd/mm/yyyy, then it is invalid. After this we call function splitDate() to get the values of variables y, m, d. Inside splitDate() we have used atoi() which returns 0 if it can't successfully convert string to an integer. For example suppose the user enters the date as : 09/ii/1988, here the value of variable m will be zero since atoi() would not work successfully. So if any of the variables y, m, d has value zero, then the date is invalid. Rest of the function is simple.

15.4.4 Leap.c

```

int isLeap(int year)
{
    if(year%4==0 && year%100!=0 || year%400==0)
        return 1;
    else
        return 0;
}/*End of isLeap()*/

```

15.4.5 Julian.c

```
/*Function to calculate julian days */
```

```

int julian(int d,int m,int y)
{
    int j=d;
    switch(m-1)
    {
        case 11: j+=30;
        case 10: j+=31;
        case 9: j+=30;
        case 8: j+=31;
        case 7: j+=31;
        case 6: j+=30;
        case 5: j+=31;
        case 4: j+=30;
        case 3: j+=31;
        case 2: j+=28;
        case 1: j+=31;
    }
    if(isLeap(y) && m!=1 && m!=2)
        j=j+1;
    return j;
}/*End of julian()*/
/*Function to get the value of day and month from julian days */
void revJulian(int j, int y, int *d, int *m)
{
    int i;
    int month[13]={0,31,28,31,30,31,30,31,31,30,31,30,31};
    if(isLeap(y))
        month[2]=29;
    for(i=1;i<=12;i++)
    {
        if(j<=month[i])
            break;
        j=j-month[i];
    }
    *d=j;
    *m=i;
}/*End of revJulian()*/

```

The logic of finding out the julian day is given in Problem 5(program P5.40) of Chapter 5

The function revJulian() finds out the value of day and month when the value of julian day and year is known. The array month stores the number of days in each month.

15.4.6 Weekday.c

```

/*To find day of week from a given date */
#include<string.h>
void weekDay(char *date,char *dayWeek)
{
    int d,m,y,j,f,h,fh,day;
    splitDate(date,&y,&m,&d);

```

```

j=julian(d,m,y);
f=(y-1)/4;
h=(y-1)/100;
fh=(y-1)/400;
day=(y+j+f-h+fh)%7;
switch(day)
{
    case 0: strcpy(dayWeek,"Saturday"); break;
    case 1: strcpy(dayWeek,"Sunday"); break;
    case 2: strcpy(dayWeek,"Monday"); break;
    case 3: strcpy(dayWeek,"Tuesday"); break;
    case 4: strcpy(dayWeek,"Wednesday"); break;
    case 5: strcpy(dayWeek,"Thursday"); break;
    case 6: strcpy(dayWeek,"Friday"); break;
}
}/*End of weekDay()*/

```

The logic of this function is similar to that given in P5.40.

15.4.7 Cmpdate.c

```

/*To compare two dates*/
int cmpDate(char *date1,char *date2)
{
    int d1,m1,y1,d2,m2,y2;
    splitDate(date1,&y1,&m1,&d1);
    splitDate(date2,&y2,&m2,&d2);

    if(y1<y2)
        return 1;
    if(y1>y2)
        return -1;
    if(m1<m2)
        return 1;
    if(m1>m2)
        return -1;
    if(d1<d2)
        return 1;
    if(d1>d2)
        return -1;
    return 0;
}/*End of cmpDate()*/

```

This function returns 1 if date1 falls before date2,

returns -1 if date2 falls before date1

returns 0 if both dates are same.

15.4.8 Diffymd.c

```

/*To get difference of two dates in years, months and days*/
void diffYMD(char *date1, char *date2,int *y,int *m,int *d)
{

```

```

int d1,m1,y1,d2,m2,y2;
splitDate(date1,&y1,&m1,&d1);
splitDate(date2,&y2,&m2,&d2);
if(d2<d1)
{
    if(m2==3)
    {
        if(isLeap(y2))
            d2=d2+29;
        else
            d2=d2+28;
    }
    else if(m2==5 || m2==7 || m2==10 || m2==12)
        d2=d2+30;
    else
        d2=d2+31;
    m2=m2-1;
}
if(m2<m1)
{
    y2=y2-1;
    m2=m2+12;
}
*y=y2-y1;
*m=m2-m1;
*d=d2-d1;
}/*End of diffYMD( )*/

```

The logic of this function is similar to that given in program P5.37.

15.4.9 Diffdays.c

```

/*To get difference of two dates in days*/
int diffDays(char *date1,char *date2)
{
    int d1,m1,y1,d2,m2,y2,j1,j2;
    int y,d,i,days;
    d=0;
    splitDate(date1,&y1,&m1,&d1);
    splitDate(date2,&y2,&m2,&d2);
    j1=julian(d1,m1,y1);
    j2=julian(d2,m2,y2);
    if(y1==y2)
        return (j2-j1);
    for(i=y1+1;i<=y2-1;i++)
    {
        if(isLeap(i))
            d=d+366;
        else
            d=d+365;
    }
    if (isLeap(y1))

```

```

    days=(366-j1)+d+j2;
else
    days=(365-j1)+d+j2;

return days;
}/*End of diifDays()*/

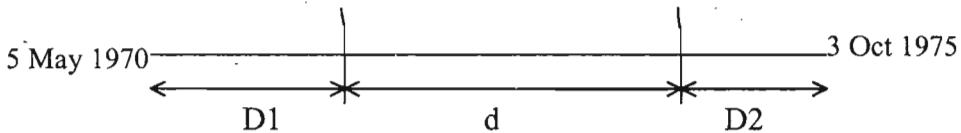
```

Suppose we want to find out the difference in dates 5 May 1970 and 3 Oct 1975. The difference in days will be sum of three components D1, d, D2.

$$D1 = 365 - \text{ Julian day of 5 May 1970} = 365 - 125 = 240$$

$$D2 = \text{ Julian day of 3 Oct 1975} = 276$$

$$d = \text{Total days in years from 1971 to 1974} = 365 + 366 + 365 + 365 = 1461$$



$$\text{So total difference in days} = 240 + 1461 + 276 = 1977$$

If both the dates are in the same year then we can find out the difference simply by subtracting the Julian days.

15.4.10 Addyear.c

```

/*To add years to a date*/
void addYear(char *date,int iyear,char *newDate)
{
    int d,m,y;
    splitDate(date,&y,&m,&d);
    y=y+iyear;
    if(d==29 && m==2 && !isLeap(y))
        d=28;
    formDate(newDate,y,m,d);
}/*End of addYear()*/

```

15.4.11 Subyear.c

```

/*To subtract years from a date*/
void subYear(char *date,int dyear,char *newDate)
{
    int d,m,y;
    splitDate(date,&y,&m,&d);
    y=y-dyear;
    if(d==29 && m==2 && !isLeap(y))
        d=28;
    formDate(newDate,y,m,d);
}/*End of subYear()*/

```

15.4.12 Addmonth.c

```
/* To add months to a date*/
void addMonth(char *date,int imonth,char *newDate)
{
    int d,m,y,quot,rem;
    splitDate(date, &y, &m,&d);
    quot=imonth/12;
    rem=imonth%12;
    y=y+quot;
    m=m+rem;
    if(m>12)
    {
        y=y+1;
        m=m-12;
    }
    if(m==2 && d>=29)
    {
        if(!isLeap(y))
            d=28;
        if(isLeap(y))
            d=29;
    }
    if((m==4 || m==6 || m==9 || m==11) && d==31)
        d=30;
    formDate(newDate,y,m,d);
} /*End of addMonth() */
```

15.4.13 Submonth.c

```
/*To subtract months from a date*/
void subMonth(char *date,int dmonth,char *newDate)
{
    int d,m,y,quot,rem;
    splitDate(date,&y,&m,&d);
    quot=dmonth/12;
    rem=dmonth%12;
    y=y-quot;
    m=m-rem;
    if(m<=0)
    {
        y=y-1;
        m=m+12;
    }
    if(m==2 && d>=29)
    {
        if(!isLeap(y))
            d=28;
        if(isLeap(y))
            d=29;
    }
    if((m==4 || m==6 || m==9 || m==11) && d==31)
```

```

    d=30;
    formDate(newDate,y,m,d);
}/*End of subMonth()*/

```

15.4.14 Adddays.c

```

/*To add days to a date*/
void addDays(char *date,int days,char *newDate)
{
    int d1,m1,y1,d2,m2,y2;
    int j1,x,j2,k;
    splitDate(date,&y1,&m1,&d1);
    j1=julian(d1,m1,y1);
    x=isLeap(y1) ? 366-j1 : 365-j1;
    if(days<=x)
    {
        j2=j1+days;
        y2=y1;
    }
    else
    {
        days=days-x;
        y2=y1+1;
        k=isLeap(y2) ? 366: 365;
        while( days>=k)
        {
            if(isLeap(y2))
                days=days-366;
            else
                days=days-365;
            y2++;
            k=isLeap(y2) ? 366:365;
        }/*End of while*/
        j2=days;
    }/*End of else*/
    revJulian(j2,y2,&d2,&m2);
    formDate(newDate,y2,m2,d2);
}/*End of addDays()*/

```

Suppose we want to add days to the date 5 May 1970. Initially we'll find out the julian day(j1) of this date which comes out to be 125. Now we'll try to find out the year and julian day(j2) of the new date, and after that we'll use function revJulian() to find out the exact day and month of the new date.

First of all we'll find out the remaining days of the year 1970 by subtracting j1 from 365.

This value comes out 240 and is stored in variable x. Now if the days to be added are less than or equal to x, then year of the new date will remain same and the julian day of the new date can be found out by writing j2 = j1 + days; For example if we have to add 200 days to 5 May 1970, then value of j2 will be $125 + 200 = 325$, so after calling revJulian() and formDate() we get the new date as 21/11/1970.

Suppose we have to add 1900 days to 5 May 1970. First we'll subtract x=240 from 1900

(comes out 1660), after this we'll reach the next year i.e. 1971. So now we have to add 1660 days to 1 Jan 1971. Now keep on subtracting the days of years from 1660 till you get a value less than 365(or 366 in case of leap year).

$1660 - 365 = 1295$, we've reached 1 Jan 1972

$1294 - 366 = 929$, we've reached 1 Jan 1973 (subtracted 366 since 1972 is leap)

$929 - 365 = 564$, we've reached 1 Jan 1974

$564 - 365 = 199$, we've reached 1 Jan 1975 ($199 < 365$)

Now the year y2 is 1975 and the julian day j2 is 199, so on calling revJulian() and formDate() we'll get the exact date which comes out to be 18/07/1975

15.4.15 Subdays.c

```
/*To subtract days from a date*/
void subDays(char *date,int days,char *newDate)
{
    int d1,m1,y1,d2,m2,y2;
    int j1,j2,x,k;
    splitDate(date,&y1,&m1,&d1);
    j1=julian(d1,m1,y1);
    if(days<j1)
    {
        j2=j1-days;
        y2=y1;
    }
    else
    {
        days=days-j1; /*Now subtract days from 1st Jan y1*/
        y2=y1-1;
        k=isLeap(y2)?366:365;
        while(days>=k)
        {
            if(isLeap(y2))
                days=days-366;
            else
                days=days-365;
            y2--;
            k=isLeap(y2)?366:365;
        }
        j2=isLeap(y2)?366-days:365-days;
    }
    revJulian(j2,y2,&d2,&m2);
    formDate(newDate,y2,m2,d2);
}/*End of subDays()*/
```

The logic of this function is somewhat similar to that of addDays().

15.4.16 Main.c

```
#include<stdio.h>
#include "dtmanip.h"
main()
```

```

{
    int choice;
    char date[11],date1[11],date2[11];
    char dayWeek[10],newDate[11];
    int iyear,imonth,dyear,dmonth,days;
    int y,m,d;

    while(1)
    {
        printf("1. Date validation\n");
        printf("2. Compare dates\n");
        printf("3. Difference of Dates in days\n");
        printf("4. Difference of Dates in years,months,days\n");
        printf("5. Day of week\n");
        printf("6. Add years\n");
        printf("7. Add months\n");
        printf("8. Add days\n");
        printf("9. Subtract years\n");
        printf("10. Subtract months\n");
        printf("11. Subtract days\n");
        printf("12. Exit\n");
        printf("Enter your choice : ");
        scanf("%d",&choice);

        switch(choice)
        {
            case 1:
                printf("Enter date (dd/mm/yyyy) : ");
                scanf("%s",date);
                if(isValid(date))
                    printf("Valid date\n");
                else
                    printf("Not a Valid Date\n");
                break;
            case 2:
                printf("Enter first date (dd/mm/yyyy) : ");
                scanf("%s",date1);
                printf("Enter second date (dd/mm/yyyy) : ");
                scanf("%s",date2);
                if(!isValid(date1) || !isValid(date2))
                {
                    printf("Enter only valid dates\n");
                    break;
                }
                if(cmpDate(date1,date2)==0)
                    printf("Date : %s and Date : %s are same.\n",date1,date2);
                else if(cmpDate(date1, date2) == 1)
                    printf("Date : %s < Date : %s \n",date1,date2);
                else if(cmpDate(date1, date2) == -1)
                    printf("Date : %s > Date : %s \n",date1,date2);
        }
    }
}

```

```
        break;
case 3:
    printf("Enter first date (dd/mm/yyyy) : ");
    scanf("%s",date1);
    printf("Enter second date (dd/mm/yyyy) : ");
    scanf("%s",date2);
    if(!isValid(date1) || !isValid(date2))
    {
        printf("Enter only valid dates\n");
        break;
    }
    if(cmpDate(date1,date2)==1)
        days=diffDays(date1,date2);
    else
        days=diffDays(date2,date1);
    printf("Difference in days = %d\n",days);
    break;
case 4:
    printf("Enter first date (dd/mm/yyyy) : ");
    scanf("%s",date1);
    printf("Enter second date (dd/mm/yyyy) : ");
    scanf("%s",date2);
    if(!isValid(date1) || !isValid(date2))
    {
        printf("Enter only valid dates\n");
        break;
    }
    if(cmpDate(date1,date2)==1)
        diffYMD(date1,date2,&y,&m,&d);
    else
        diffYMD(date2,date1,&y,&m,&d);
    printf("Difference of the two dates is : ");
    printf("%d years %d months %d days\n",y, m, d);
    break;
case 5:
    printf("Enter date (dd/mm/yyyy) : ");
    scanf("%s",date);
    weekDay(date,dayWeek);
    printf("Day of week is %s\n",dayWeek);
    break;
case 6:
    printf("Enter date (dd/mm/yyyy) : ");
    scanf("%s",date);
    if(!isValid(date))
    {
        printf("Enter a valid date\n");
        break;
    }
    printf("Enter the number of years to be added : ");
    scanf("%d",&iyear);
    addYear(date,iyear,newDate);
```

```
printf("Now the new date is %s\n",newDate);
break;
case 7:
    printf("Enter date (dd/mm/yyyy) : ");
    scanf("%s",date);
    if(!isValid(date))
    {
        printf("Enter a valid date\n");
        break;
    }
    printf("Enter the number of months to be added : ");
    scanf("%d",&imonth);
    addMonth(date,imonth,newDate);
    printf("Now the new date is %s\n",newDate);
    break;
case 8:
    printf("Enter date (dd/mm/yyyy) : ");
    scanf("%s",date);
    if(!isValid(date))
    {
        printf("Enter a valid date\n");
        break;
    }
    printf("Enter the number of days to be added : ");
    scanf("%d",&days);
    addDays(date,days,newDate);
    printf("Now the new date is %s\n",newDate);
    break;
case 9:
    printf("Enter date (dd/mm/yyyy) : ");
    scanf("%s",date);
    if(!isValid(date))
    {
        printf("Enter a valid date\n");
        break;
    }
    printf("Enter the number of years to be subtracted : ");
    scanf("%d",&dyear);
    subYear(date,dyear,newDate);
    printf("Now the new date is %s\n",newDate);
    break;
case 10:
    printf("Enter date (dd/mm/yyyy) : ");
    scanf("%s",date);
    if(!isValid(date))
    {
        printf("Enter a valid date\n");
        break;
    }
    printf("Enter the number of months to be subtracted : ");
    scanf("%d",&dmonth);
```

```

        subMonth(date,dmonth,newDate);
        printf("Now the new date is %s\n",newDate);
        break;
    case 11:
        printf("Enter date (dd/mm/yyyy) : ");
        scanf("%s",date);
        if(!isValid(date))
        {
            printf("Enter a valid date\n");
            break;
        }
        printf("Enter the number of days to be subtracted : ");
        scanf("%d",&days);
        subDays(date,days,newDate);
        printf("Now the new date is %s\n",newDate);
        break;
    case 12:
        exit(1);
    default:
        printf("Wrong choice\n");
    }/*End of switch*/
}/*End of while*/
}/*End of main()*/

```

15.5 Building Project in Turbo C

1. First go to Project → Open Project
Create a project file DT.prj
2. Now a window will open that will display all files in the project. Right now we don't have any file added in project.
3. Now go to menu option Project → Add Items and add a file in the project. The added file will come in project window.
4. Similarly add all files one by one in the project. Now you can see all the files in project window.
5. Now you can open any file by double clicking on that item in project window. Any file can be deleted from the project by selecting it and pressing Delete or from the menu option delete item.
6. Now go to Compile → Build All, it will compile all the files and build the project. All the errors will come in window for each file. If there is no error then it will show success.
7. Now go to Run → Run and program will run.
8. Now close all windows including project item window.
9. Now go to Project → Close Project for closing the project.

15.6 Testing

Now we go for next and last step that is testing. Here we write different test cases for all modules and see whether the modules give correct results for each case. We perform testing for each individual module which is called module testing, as well as we do integration testing which tests that after integration also all modules are working fine or not or we find out which module is giving problem after integration.

In our project we will test functionality after integration because here module testing and integration

testing is more or less same. So we will check functionality with different test cases if any errors are coming then we will modify that particular part of coding and do the testing again.

15.7 Creation Of Library And Using it in your Program in Turbo C

Suppose we want to add a module in library. First we will produce .obj file by compilation then we will add the function of .obj file to library with the help of tlib utility. The whole process will be as-

1. Produce the .obj file by compiling file as-

d:\> tcc -c xyz.c (ENTER)

Now xyz.obj will be created.

We can also make the .obj file by Alt + F9.

2. Now we will add all the functions of xyz.obj in new library as-

d:\> tlib ABC.lib+xyz.obj (ENTER)

Now new library ABC.lib will be created which will contain all the functions of XYZ.obj.

3. Create one header file named ABC.h which will contain prototype of all the functions available in library ABC.lib.

4. Now you can call any function of library ABC.lib by including header file ABC.h.

5. Suppose we want to add some more modules. First we will get the .obj file of all modules then they will be added as-

d:\> tlib ABC.lib+XYZ1.obj+XYZ2.obj+XYZ3.obj (ENTER)

Now we will create library dtmanip.lib for our Date project and will call the functions in one sample file.

1. First we will produce .obj files for all modules as-

d:\> tcc -c Datefmt.c Valid.c Leap.c Julian.c Weekday.c Cmpdate.c Diffymd.c Diffdays.c
Addyear.c Subyear.c Addmonth.c Submonth.c Adddays.c Subdays.c

This will produce .obj files for all modules.

2. Now we will create one new library dtmanip.lib for date manipulation by adding these modules.

d:\>tlib dtmanip.lib+Datefmt.obj+Valid.obj+Leap.obj+Weekday.obj+Cmpdate.obj+
Diffdays.obj+Diffymd.obj+Addyear.obj+Addmonth.obj+Adddays.obj+Subyear.obj+
Submonth.obj+Subdays.obj+ Julian.obj

Now new library dtmanip.lib will be created, which will contain all the functions of above modules.

3. We will create one header file dtmanip.h, which is already given before.

4. Now we will create one sample file like main.c of Date project and call the functions available in dtmanip.lib.

5. Create new project testlib and add dtmanip.h, main.c and dtmanip.lib in this project.

6. Now we can build and run this project.

So now we have date manipulation library that can be used anywhere else also whenever date related manipulation is required.

15.7.1 Deletion of a Module From Library

We can delete any module from library as-

d:\> tlib dtmanip-Diffdays

This will delete the module of Diffdays from dtmanip library.

15.7.2 Getting Modules From Library

We can get module from library as-

```
d:\> tlib dtmanip*Valid
```

Now we will get the Valid module from dtmanip.lib. This can be added in different library as-

```
d:\> tlib newdt.lib+Valid.obj
```

This will add Valid module which we got from dtmanip.lib, into newdt.lib.

15.7.3 Changing Version Of Module In Library

This can be done by deleting and then adding a particular module from library as-

```
d:\> tlib dtmanip-Adddays
```

This will delete the module Adddays from dtmanip.lib library.

```
d:\> tlib dtmanip+Adddays.obj
```

This will add the new version of module Adddays in dtmanip.lib library.

We can do the same thing in one step also as-

```
d:\> tlib dtmanip-+Adddays
```

This will delete the Adddays module from dtmanip.lib and add the new version of Adddays in dtmanip.lib.

15.8 Building Project On Unix

We require makefile concept for building project on unix platform. Unix provides a utility make to build project. This utility takes care of dependencies.

We know very well how to compile files with unix cc compiler and get the object and executable file.

Suppose we want to compile a single file main.c. Then it looks very easy-

```
cc -o Date main.c
```

It will compile main.c and give the executable Date.

Suppose we have more than one file then we can compile them as-

```
cc -o Date main.c Datefmt.c Valid.c Leap.c Weekday.c Cmpdate.c Diffdays.c Diffymd.c Addyear.c  
Addmonth.c Adddays.c Subyear.c Submonth.c Subdays.c Julian.c
```

Now it will compile all the files and give the executable Date as output.

Suppose we are compiling all these files first time then it looks fine. But suppose we are changing source code of some file and again compiling then it will be a time taking process. We can provide .o compiled object file of them, which are not changed. Suppose only main.c is changed and rest of them are same then we can use .o object file of rest of them as-

```
cc -o Date main.o Datefmt.o Valid.o Leap.o Weekday.o Cmpdate.o Diffdays.o Diffymd.o Addyear.o  
Addmonth.o Adddays.o Subyear.o Submonth.o Subdays.o Julian.o
```

We can create only object file with -c option as-

```
cc -c file.c
```

So now we know that we have no need to compile all the files again if their source is not changed. We can use their object file itself. But this is always very difficult to remember that which file is changed

and which one need not to recompile again. Even compiling these files again and again on command line is also a big problem.

The solution is make utility. This utility takes care of all these problems and it even provides some more facility to manage project in a better way.

15.8.1 Writing Makefile

make utility uses makefile for building project. This makefile contains commands that tell make utility how to build your project with given source files. This makefile contains the dependency rules and construction rules.

left side : right side

cc right side (compilation by using options for linking library)

Here in first line, left side is the target and right side is the source files on which target is dependent. Second line is for the construction. If any source file is changed then make will go for construction rule otherwise it will not do anything. We have to understand what exactly make utility does with makefile.

1. It reads makefile and comes to know which object file and library is needed for target and then it checks what source files are needed for each object.
2. It checks time and date for each object file against dependent source, if any source is of later date than its object file, then it will apply construction rule for that particular object.
3. Now it will check time and date of target against dependent object file if any object file is of later date than its target then it will apply construction rule for target.

Now we will see how will be our makefile for Date project then we will see what is the meaning of each line of our makefile.

Makefile 1:

```
#  
# Date project Makefile  
#  
#  
TARGET=Dt  
OBJS=Datefmt.o Valid.o Leap.o Julian.o Cmpdate.o Adddays.o Addmonth.o Addyear.o \  
      Subdays.o Submonth.o Subyear.o Diffymd.o Diffdays.o Weekday.o Main.o  
  
$(TARGET):$(OBJS)  
    cc -o $(TARGET) $(OBJS)  
Datefmt.o:Datefmt.c  
    cc -c Datefmt.c  
Valid.o:Valid.c  
    cc -c Valid.c  
Leap.o:Leap.c  
    cc -c Leap.c
```

```
Julian.o: Julian.c  
    cc -c Julian.c  
Cmpdate.o: Cmpdate.c  
    cc -c Cmpdate.c  
Adddays.o: Adddays.c  
    cc -c Adddays.c  
Addmonth.o: Addmonth.c  
    cc -c Addmonth.c  
Addyear.o: Addyear.c  
    cc -c Addyear.c  
Subdays.o: Subdays.c  
    cc -c Subdays.c  
Submonth.o: Submonth.c  
    cc -c Submonth.c  
Subyear.o: Subyear.c  
    cc -c Subyear.c  
Diffymd.o: Diffymd.c  
    cc -c Diffymd.c  
Diffdays.o: Diffdays.c  
    cc -c Diffdays.c  
Weekday.o: Weekday.c  
    cc -c Weekday.c  
Main.o: Main.c  
    cc -c Main.c
```

We can write the makefile as also.

Makefile2:

Date project Makefile

```
TARGET=Dt  
OBJS=Dtaefmt.o Valid.o Leap.o Julian.o Cmpdate.o Adddays.o Addmonth.o Addyear.o \  
      Subdays.o Submonth.o Subyear.o Diffymd.o Diffdays.o Weekday.o Main.o  
  
${TARGET}  
${TARGET}: ${OBJS}
```

```
cc $(OBJS) -o $@
```

15.8.2 Building Project With Make

We can simply execute make as-

```
$make (ENTER)
```

It will search for file makefile in current directory and will build the project. But suppose we have many makefiles for different projects, then we can use some naming convention to name makefile for projects. Like we can name our project makefile as project_makefile. Here our project is Date so we can name our makefile as Date_makefile.

We can execute this as-

```
$make -f Date_makefile (ENTER)
```

I would like to recommend you to see the unix man page of make. This can be very useful for build work, which also takes a slice of project time. Generally one person in team is devoted for handling build work of project.

15.9 Creation Of Library And Using In Your Program in Unix

Now we will see process for creation of library and how to call function of this library in another file.

1. First we will compile all the files for creating .o for source files.

```
$cc -c Datefmt.c Valid.c Leap.c Julian.c Cmpdate.c Adddays.c Addmonth.c Addyear.c \
Subdays.c Submonth.c Subyear.c Diffymd.c Diffdays.c Weekday.c Main.c
```

This will produce .o file for all source .c files.

2. Now we will assign all .o files to one variable.

```
$OBJS= Dtaefmt.o Valid.o Leap.o Julian.o Cmpdate.o Adddays.o Addmonth.o Addyear.o \
Subdays.o Submonth.o Subyear.o Diffymd.o Diffdays.o Weekday.o Main.o
```

3. Library dt will be created with archiver ar as-

```
ar rv Libdt.a $OBJS
```

4. Now we will write file Main.c and call the functions of dt library in it.

5. Now Main.c will be compiled as-

```
cc -o Dt Main.c -L. -ldt
```

Chapter 16

Code Optimization in C

16.1 Optimization is a Technique

When we think about optimization, it looks like we are talking only about making our code shorter. But its not true, sometimes shorter code takes more execution time compared to large code. So here comes performance, and it depends on whether you want shorter code that is good for maintenance but less in performance, or larger code that is bad for maintenance but good in performance. Now we will come to resource optimization. Generally resources are shared between different processes. Suppose your program takes more resources, then definitely it will affect performance of other processes that need same resources. So we have a need to write and optimize our program keeping in mind resources e.g. processor's time. You will be surprised to know that most of the time, increasing of hardware like processors comes only because of our bad programming like unnecessary variables, bad searching and sorting algorithms. This is because we think only about running our program properly, and consider about resources only after getting some problem. Companies are spending more than \$50 million on resources, adding more than 12 processors in single server, only because they don't consider other things in programs and are helpless to find this individual problem. So our work is not only to write program but also to develop the habit of writing optimized programs considering the overall view of system. That's why optimization is a technique because it considers many other things in program not only codes. You will see more things later in each topic.

16.2 Optimization With Tool

The first tool with our code is compiler. We can do lot of optimization with our compiler itself. It provides lot of options. So firstly we should understand our compiler better. Compiler manual can help you in understanding different options. We should know what our compiler can do for us and what it cannot do. So we do whatever it does not do for us and we will not do whatever it does for us. We have some different tools, which can help us to write our code in a better way. Like we can use tool purify to know where memory leak is coming in our code. Also we can use purecov to know how much coverage is done by our code, so that we can write code that has better coverage.

16.3 Optimization With Loop

16.3.1 Loop Unrolling

We can unroll small loops. Let us take an example-

```
int a=5;
for(i=0;i<4;i++)
{
```

```

    value[i]=a;
    a=a+5;
}

```

This can be simply written like this-

```
int value[ ] = { 5, 10, 15, 20 };
```

at initialization time. Alternatively it can be written in between program like this-

```

value[0]=5;
value[1]=10;
value[2]=15;
value[3]=20;

```

This is obviously better than previous one but most of the times compiler does this automatically, but we can expect little bit advantage and even it will be better to make a habit of writing optimized code. You can get this advantage in matrix program.

16.3.2 Avoiding Calculations In Loops

We should avoid any calculation, which is more or less constant in value within the loop. The same thing is true for avoiding calculation or anything, which can be removed from inner loop.

```

for(i=0;i<10;i++)
{
    value[i]=b*(20-b/3*c%2)*i/2;
}

```

We can avoid this calculation and it can be written as-

```

a=b*(20-b/3*c%2)/2
for(i=0;i<10;i++)
{
    value[i]=a*i;
}

```

Always try to minimize things in inner loop because this can improve efficiency.

16.4 Fast Mathematics

16.4.1 Avoid Unnecessary Integer Division

We know that division operation is slow so we should try to minimize the division operations. Let us take an example-

```

int a, b, c, d;
d = a/b/c;

```

This division operation can be eliminated as-

```
d=a/(b*c);
```

Multiplication operation is more effective than division operation. So we should try to convert division operation to multiplication.

16.4.2 Multiplication And Division by Power Of 2

We know that base of computer understanding is binary(0 or 1) i.e. base of 2. So we always need to do multiplication and division operations for numbers which are power of 2. One way is to do simple

division or multiplication with the number, which is power of 2 as-

```
a = a * 8 ( 8 is  $2^3$  )
a = a / 8 ( 8 is  $2^3$  )
```

The second way to perform the same operation is by using left shift $<<$ for multiplication operation and right shift $>>$ for division operation. This can be done as-

```
a = a << 3 (or a = a *  $2^3$ )
a = a >> 3 (or a = a /  $2^3$ )
```

So,

$a << b$ is equivalent to $a * 2^b$ and
 $a >> b$ is equivalent to $a / 2^b$

These bit operations will be much faster than multiplication and division operations. For simple operations, the compiler may automatically optimize the code but in case of complex expressions it is always advised to use bit operations, which is more effective optimization.

Suppose we want to multiply by 6 then it can be written as-

```
a = a << 1 + a << 2;
```

Compiler can not optimize this code so we should always make habit of writing optimized code without thinking that it can be optimized by compiler or not.

16.5 Simplifying Expressions

Sometimes we can reduce some operations by simplifying expressions. Let us see a simple example-

```
x * y + y * 4
```

This can be simplified as-

```
(x + 4) * y
```

This will reduce one multiplication operation.

16.6 Declare prototypes for Functions

We should always declare prototype for functions. This will tell the compiler about functions and it can be helpful for compiler to optimize the code.

16.7 Better Way Of Calling Function

We know that calling a function means a stack will be setup with parameters of functions, so we should always try to reduce the parameter of functions, which can improve the stack mechanism. It will be better to use pointer instead of passing structure or big array, which can be overhead for stack and can decrease the efficiency of program.

16.8 Prefer int to char or short

We should always prefer int to char because C performs all operations of char with integer. In all operations like passing char to a function or any arithmetic operation, first char will be converted into integer and after completion of operation it will again be converted into char. For a single char this may not affect efficiency but suppose same operation is performed 100 times in a loop then it can decrease the efficiency of program.

16.9 Use of Register Variables

Register variables are stored in CPU registers. If you are declaring a variable as register then it will be accessed from register instead of memory which will always be faster. We know that registers are limited so we should declare only those variables as register, which are heavily used. If we have nested loops, then we should take the inner loop variables as register, because they will be used more times than outside loop variables.

16.10 Optimization With Switch Statement

Compilers translate switch statements in different ways. If case labels are small contiguous integer values, then it creates jump table. This is very fast and doesn't depend on number of case labels also. If case labels are longer and not contiguous then it creates comparison tree i.e. if...else statements. So in this case we should keep the most frequent label first and least frequent label should be at last. Let us take an example-

```
switch( no_of_day )
{
    case 31:
        .....
        break;
    case 30:
        .....
        break;
    case 28:
    case 29:
        .....
        break;
    default:
        printf("Incorrect value\n");
        break;
} /*End of switch*/
```

Sometimes we see lot of repeated code written in all cases except one or two statements. Let us see an example-

```
switch(expression)
{
    case a:
        .....
        .....
        break;
    case b:
        .....
        .....
        break;
    case c:
        common statements;
        different statements;
        common statements;
        break;
    case d:
        common statements;
```

```
    different statements;
    common statements;
    break;
case e:
    common statements;
    different statements;
    common statements;
    break;
case f:
    common statements;
    different statements;
    common statements;
    break;
default:
    break;
}
```

Here we can take all cases together and can write common statements only once and different statements in related cases using another switch. Here we will take case c, d, e, f together and write common statements, then we can use another switch and write different statements in case c, d, e, f. Then after this switch we can again write common statements. This problem is repeated by lot of programmers because they don't use this trick.

```
switch(expression)
{
    case a:
        .....
        .....
        break;
    case b:
        .....
        .....
        break;
    case c:
    case d:
    case e:
    case f:
        common statements;
        switch(expression);
    {
        case c:
            different statements;
            break;
        case d:
            different statements;
            break;
        case e:
            different statements;
            break;
        case f:
            different statements;
            break;
    }
}
```

```

}/*End of switch*/
common statements;
break;
default:
    break;
}/*End of switch*/

```

Hopefully this will help lot of new programmers.

16.11 Avoid Pointer Dereference

We should always avoid pointer dereferencing in loop or while passing it as parameter. It creates lot of trouble in memory. Suppose it is used in loop then it will affect the efficiency. So better assign it to some temporary variable and then use that temporary variable in loop.

16.12 Prefer Pre Increment/Decrement to Post Increment/Decrement

We should prefer pre increment/decrement to post increment/decrement wherever both of them have been used for same work. In pre increment, it first increments the value and just copies to the variable location but in post increment it first copies the value to temporary variable, increments it and then copies the value to variable location. For a one time operation may be it will not affect but suppose this operation is in 1000 time loop then it will sure effect the efficiency. So it will always be better to use pre increment/decrement to get efficiency.

16.13 Prefer Array to Memory Allocation

Wherever possible we should use array instead of pointer with memory allocation. Accessing with array index is always fast compared to dynamic storage. Basically it depends on the requirement also. But if we are sure that we are not going to use storage more than particular size then we can prefer array for that particular size. Memory allocation and deallocation has some burden and in some circumstances it can be avoided with array. You can take example of creating stack or queue, which will be better with array instead of implementing with linked list when we know the size of stack and queue will not be more than particular size.

16.14 Use Array Style Code Rather Than Pointer

Pointer is special facility in C that distinguishes it from other languages. Effective use of pointer makes C language more efficient. But it also creates problem for compiler to optimize code. Pointer can read and write at any part of memory and it becomes difficult for compiler to optimize code which uses pointer. In some places we can use pointer as array. Like we can use `ptr[0], ptr[1].....` which will make compiler's job easier for optimization.

16.15 Expression Order Understanding

In C we use lot of expressions in control statements and we also join more than one expressions by `||` or `&&` operator. Let us take an example of `||` operator-

A || B

Here first A will be evaluated, if it's true then there is no need of evaluation of expression B and condition will become true. But if A is false then it will check for B, now if B is true then whole condition will become true otherwise it will be false. Suppose probability of expression A becoming true is very less and B becoming true is high then most of the time after evaluating expression A it will go for evaluation of B. So most of the time we are evaluating A unnecessarily. In this case, it would be better to put

the expression B first in condition, which has higher probability of becoming true.

B || A (preferred)

Now it will first check B and if it's true then condition will be true and it will skip most of the time evaluation of A and will increase the performance. Now let us check one example of && operator-

C && D

Here first expression C will be evaluated and if its false then no need of evaluation of expression D and condition will become false. If C is true then it will check for D, if D is also true then whole condition will become true otherwise it will become false. Here we will check which expression has more probability of becoming false, that expression should be placed first expression in condition.

Now suppose probability of becoming true or false is equal in all expressions which are joined with || or && operator then the expression which can be evaluated faster than others should be first in condition.

16.16 Declaration Of Local Function

We should always declare all the functions used in a program and if the function is used in that file only then it should always be declared as static, that means scope of that function will be limited to that file only. If it is not declared as static then it will be interpreted as extern and compiler will take unnecessary burden for external linkage of this function because it will think that this can be used outside file also.

16.17 Breaking Loop With Parallel Coding

We can write parallel coding by writing instructions in pipeline. We can write one instruction of loop into parts, which can help to reduce the number of cycles and definitely increase the efficiency. Let us take an example-

```
total=0;
for(i=1;i<=100;++i)
    total += i;
```

Here we are simply evaluating the total of numbers from 1 to 100. The same thing can be done as:

```
total1=0;
total2=0;
total3=0;
total4=0;
total=0;
for(i=1;i<=100;i=i+4)
{
    total1=total1+i;
    total2=total2+i+1;
    total3=total3+i+2;
    total4=total4+i+3;
}
total = total1 + total2 + total3 + total4;
```

It doesn't seem very attractive optimization but it can improve efficiency where big calculation is required like in mathematics or graphics.

16.18 Trick to use Common Expression

We can extract common expression, which is used at many places in the program. Let us take an example-

```
x = a*10/b;
y = c*5/b;
```

We can write the same as-

```
temp = 5/b;
x = a*2*temp;
y = c*temp;
```

Here we are avoiding evaluation of an expression by using temp variable and multiplication. Here the expression is very small, but suppose it is a big one and used at lot of places then it will be better choice. These things are to be done manually by properly reviewing the code, because compiler will not be able to do optimization for these types of tricks.

16.19 Declaring Local Variables Based On Size

The local variables should be declared based on their sizes. We should prefer to declare larger to smaller datatype size variables. Compiler allocates variables in the same sequence in which they are declared. Declaring larger to smaller datatype variables will help the compiler in alignment and padding. Let us take an example-

```
int a;
float b;
long c;
double d, e[2];
```

This should be declared as-

```
double e[2];
double d;
long c;
float b;
int a;
```

16.20 Prefer Integer Comparison

We should always prefer integer comparison in conditions because it is faster as compared to any other data type.

16.21 Avoid String Comparison

We should try to avoid strings comparisons as much as possible. String comparison is very slow as it compares each character. Generally we use `strcmp()` function for comparing two strings. But this is very slow and can be very inefficient if we have lot of data in strings for comparing. We know we can't avoid comparing strings, but we can use some trick for the comparison:

1. Compare the index of '\0' of first string with the same index of second string.
2. Firstly compare first character of both strings; if they are same then only compare the full strings.

These two points can be very important in avoiding most of the string comparison.

Chapter 17

C and Assembly Interaction

C has wonderful feature to interact with assembly. This makes it more useful. It's good that we can interact with other language but the question arises why is it needed. We know very well that assembly is a low level language and it can interact directly with hardware. It does manipulation in registers so it is always fast compared to other languages. There are so many things that can be done with assembly language only. We can use these features with C very easily. We can write inline assembly routine in our C code, as well as our C module can also interact with assembly module.

17.1 Inline Assembly Language

We can write assembly statement in C language by using asm keyword before each assembly statement in our program.

Ex-

```
asm mov AX , 5
```

Let us take a C program to print a character by using assembly statement.

```
main( )  
{  
    asm  mov  d1,65  
    asm  mov  ah,2  
    asm  int  21 h  
}
```

Here we are using INT 21 with assembly. Let us see its description-

INT 21 H Service 2 – Character Output on

Input:

DL= ASCII character

AH=2

For writing same thing in MASM we have to use keyword _asm and all the assembly statements will be in block starting with '{ ' and ending with '}'.

```
main( )  
{  
    _asm  {  
        mov  d1,65  
        mov  ah,2  
        int  21 h  
    }  
}
```

Now we can compile and run this program like other C programs. Now you will think how it will happen. Actually your C compiler parses assembly code from your program, creates one assembly file and then assembles it with the assembler (TASM or MASM), when it creates object file then this object file will be linked to your C executable.

Now the question arises this can be done very easily in C language program, then why inline assembly is needed. Remember interaction through hardware can be done best through assembly language only. There are so many things, which cannot be done with C, that can be done only through assembly language and there is no alternative C function available.

Suppose we want to get the scan code of any pressed key of keyboard. Every key has scan code including combination of special keys (Ctrl, Alt, Shift) and normal key. There is no C function available for getting scan code of pressed key. But we can use INT 16 with assembly for getting scan code. Let us see how INT 16 works.

INT 16H, service 0 - Read key from keyboard

```
Input:      AH=0
Output:     AH=Scan code      AL=ASCII code
main( )
{
    asm mov ah,0
    asm int 16 H
}
```

After these two asm statements, scan code will be available in AH register which we can use in another part of program. So now you got a brilliant idea, start writing the program for keyboard interaction.

Now we want to take the data in C statement and use it in asm statements. Suppose we want to print a string then we will need to use 9th routine of INT 21. See the description below-

INT 21 H Service 9 – String Print

Input:

AH=9

DS:DX points to a string that ends in '\$'

Service 9 of INT 21 is used to print the string which ends with '\$'. Let us take an example program.

```
main( )
{
    char *ptr="C with Assembly $";
    asm mov dx,ptr
    asm mov ah,9
    asm int 21 h
}
```

Here INT 21 Service 9 will print-

C with Assembly

Now we want to write assembly statements in user defined function. This can be written as-

```
main( )
{
    func( );
}
```

```
func( )
{
    asm mov dl, 65
    asm mov ah, 2
    asm int 21 h
}
```

Here we have written asm statements for printing character 'A' in function func(). Suppose we want to take this character in C statements then it can be as-

```
char ch='A';
main()
{
    func();
}
func()
{
    asm mov dl, ch
    asm mov ah, 2
    asm int 21h
}
```

Here we have written a function which is not passing any value and does not return any value.

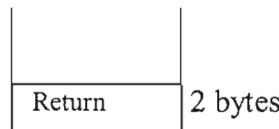
Suppose we want to write a function which passes some value and we want to use these parameter values in asm statement and want to return the value from function, then we will have to do it in some different way.

In C, when we pass parameters it goes to the stack. It pushes the value of parameter instead of address, and for returning we use assembly instruction RET. It pushes the parameter in reverse order.

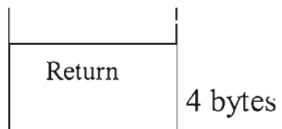
Let us take a program where we don't pass any parameter-

```
main()
{
    func();
}
func()
{
    asm mov dl, 65
    asm mov ah, 2
    asm int 21h
}
```

Here we are not passing any parameter, so the func() will be set up in stack as-



Small, Compact

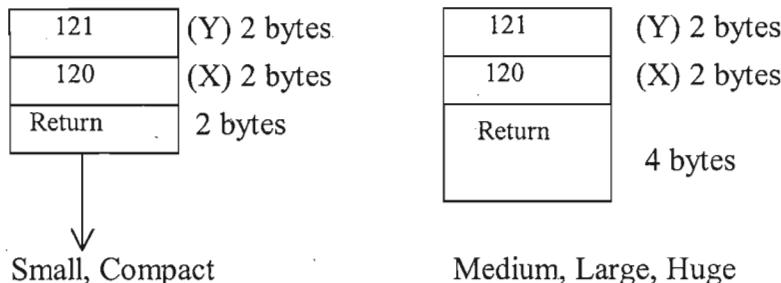


Medium, Large, Huge

Here in large, medium and huge return address is of 4 bytes because these memory models can have more than 1 code segment.

Suppose we are passing parameters in our function. Let us take an example-

```
main()
{
    display('x', 'y');
}
display(char x, char y)
{
    asm .....
    .....
    .....
}
```



We know C pushes the parameters in reverse order(compiler dependent), so first y will be pushed and then x will be pushed in stack.(undefined)

Now the question is how to get these parameters from stack to use in asm statement. SP is the stack pointer which points to the current word in stack. We can use Base pointer BP to store the position of SP, so that with the help of BP we can get the parameter value from stack and then it can be used in our asm statement.

17.2 Linking Of Two Assembly Files

Now we will see how we can write two assembly files with one file having a procedure that will be called in another file. Similar things we will do with other C and assembly files where we will call assembly procedure in C program as well as C functions in assembly program. But first we will learn how to link two assembly programs. Let us take a simple program that prints a single string-

CODE SEGMENT

```
ASSUME CS:CODE_SEG, DS:CODE_SEG, ES:CODE_SEG, SS:CODE_SEG
```

```
ORG 100H
```

```
START : JMP DISPLAY
```

```
STR DB "Suresh Srivastava.$"
```

```
DISPLAY PROC NEAR
```

```
MOV DX, OFFSET STR
```

```
MOV AH, 9
```

```
INT 21H
```

```
INT 20H
```

```
DISPLAY ENDP
```

```
END START
```

Here our program has procedure to print the string. Let us take another program that calls procedure, which is used to display the string.

```
CODE_SEG SEGMENT
```

```
ASSUME CS:CODE_SEG, DS:CODE_SEG, ES:CODE_SEG, SS:CODE_SEG
```

```
ORG 100H
```

```
START: DISPLAY PROC NEAR
```

```
CALL SUB_DISPLAY
```

```
INT 20H
```

```
DISPLAY ENDP
```

```
SUB_DISPLAY PROC NEAR
```

```
JMP GO
```

```
ALL_OK DB "Suresh Srivastava. $"
```

```
GO: MOV DX, OFFSET ALL_OK
```

```
MOV AH, 9
```

```
INT 21H
```

```
RET
```

```
SUB_DISPLAY ENDP
```

```
CODE_SEG ENDS
```

```
END START
```

Now think about modular programming. Suppose we want to write all the procedures in different files and those procedures will be called in the main module. So we will have to link all the files. Let us take an example program that has two files, one file has procedure which is called in another file.

MAIN.ASM

```
CODE_SEG SEGMENT
```

```
ASSUME CS:CODE_SEG, DS:CODE_SEG, ES:CODE_SEG, SS:CODE_SEG
```

```
ORG 100H
```

```
START:
```

```
DISPLAY PROC NEAR
```

```
CALL SUB_DISPLAY
```

```
INT 20H
```

```
DISPLAY ENDP
```

```
CODE_SEG ENDS
```

```
END START
```

EXPROC.ASM

```

CODE_SEG SEGMENT
    ASSUME CS:CODE_SEG, DS:CODE_SEG, ES:CODE_SEG, SS:CODE_SEG
SUB_DISPLAY PROC NEAR
    JMP GO
    ALL_DK DB "Suresh Srivastava. $"
GO: MOV DX, OFFSET ALL_OK
    MOV AH, 9
    INT 21H
    RET
SUB_DISPLAY ENDP
CODE_SEG ENDS
END

```

We can see here both the files are in code segment. END START and ORG statements are only in MAIN.ASM. Because it can have only one entry and we want both files one after another in memory.

17.2.1 Memory Models

We have different memory models – tiny, small, medium, compact, huge which set the limit of code and data area. So we select the memory model depending on our program requirements. Different models and their limit for code, data and array are as given below-

	Code Segment	Data Segment	Array
Tiny	All in same segment		
Small	< 64 K	< 64 K	< 64 K
Medium	> 64 K	< 64 K	< 64 K
Compact	< 64 K	> 64 K	< 64 K
Large	> 64 K	> 64 K	< 64 K
Huge	> 64 K	> 64 K	> 64 K

17.2.2 C And Segments in Library

We can link assembly program with C program only when both of them have same segment name. C standard library uses different segment names based on their memory models. To overcome this problem, we have simplified segment directive which is supported by turbo C as well as Microsoft C. We have following keywords for declaration.

.CODE	-	code
.DATA	-	Initialized data
.DATA?	-	Uninitialized data
.FARDATA	-	Initialized non-Dgroup data(Compact/Large/Huge)

.FARDATA? - Uninitialized non-Dgroup data(compact/Large/Huge)

Here we have no need to write like CODE_SEG, when we will use the memory model then correct segment name will be placed. Let us see a program-

MAIN.ASM

```
.MODEL SMALL
```

```
.CODE
```

```
    ORG 100H
```

```
START:
```

```
DISPLAY PROC NEAR
```

```
    CALL SUB_DISPLAY
```

```
    INT 20H
```

```
DISPLAY ENDP
```

```
END START
```

EXPROC.ASM

```
.MODEL SMALL
```

```
.CODE
```

```
SUB_DISPLAY PROC NEAR
```

```
    JMP GO
```

```
    ALL_OK DB "Suresh Srivastava.$"
```

```
GO: MOV DX, OFFSET ALL_OK
```

```
    MOV AH, 9
```

```
    INT 21H
```

```
    RET
```

```
SUB_DISPLAY ENDP
```

```
END
```

Here model is declared as SMALL, so the segment names will be automatically placed based on memory model. Here code segment is automatically ended with the end of file. If data segment is available then code segment will end when the data segment starts. We can see that we don't have to worry for maintaining different segment names, with simplified directive, segment names will be placed based on memory model. For using segment address, we will have to use with @ prefix eg @code, @data.

Now we have to link both the files, so procedure will be declared as EXTRN in MAIN.ASM and it will be declared PUBLIC when its definition is available. See the program below for linking.

MAIN.ASM

```
EXTRN SUB_DISPLAY:NEAR
```

```
MODEL SMALL
```

```
.CODE
    ORG 100H
START:
DISPLAY PROC NEAR
    CALL SUB_DISPLAY
INT 20H
DISPLAY ENDP
END START
```

EXPROC.ASM

```
PUBLIC SUB_DISPLAY
.MODEL SMALL
.CODE
SUB_DISPLAY PROC NEAR
    JMP GO
    ALL_OK DB "Suresh Srivastava.$"
GO: MOV DX, OFFSET ALL_OK
    MOV AH, 9
    INT 21H
    RET
SUB_DISPLAY ENDP
END
```

Here we can see that SUB_DISPLAY is declared as PUBLIC in file EXPROC.ASM, so that it can be called in another file. It is declared as EXTRN in MAIN.ASM, so it can be called in file MAIN.ASM.

```
d:\>TASM MAIN.ASM
d:\>TASM EXPROC.ASM
d:\>TLINK MAIN+EXPROC
d:\>EXE2BIN MAIN MAIN.COM
d:\>MAIN
Suresh Srivastava
```

17.3 Linking Assembly Procedure in C Program

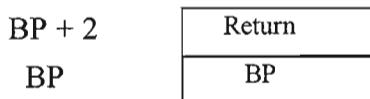
Now we will see how we can call one assembly procedure in C program. We will write main program in C, which will call assembly procedure from asm program for getting the scan code of any key of keyboard.

Main.c

```
extern int getsancode( );
main()
{
```

```
printf("Scan code of pressed key = %x\n",getscancode());
}
```

getscancode() function will be defined in asm program, so here it will be declared as extern to call this asm procedure in C program. But this procedure name will be with (_) underscore because for linking procedure in C this will be needed. Let us assume the memory model is small then our stack will be as-



Since we are not passing any parameter, so only return value of function will be available in stack frame. See the asm procedure below-

GET.ASM

```
.MODEL SMALL
.CODE
PUBLIC _getscancode
_getscancode PROC
    PUSH BP
    MOV BP, SP
    MOV AH, 0
    INT 16H
    POP BP
    RET
_getscancode ENDP
END
```

Here this procedure will get the scan code of pressed key and it will return this value. Here procedure name is in small letters because C is case sensitive and it will search for label in small letters. But assembler makes all public variables in capital letters, so we should use -mx option of assembler, so the same case of public variable will be used. Now we can link the assembly procedure with C program as-

```
d:\>tasm -mx GET.ASM
d:\>tcc Main.c GET.obj
```

We have seen how to call assembly procedure in C program, but there was no parameter in function. Now we will see how to pass parameter in C function and how it can be used in assembly procedure. Suppose we are calling one function ADDNUM in C program as-

MAIN.C

```
extern int ADDNUM(int,int);
main()
{
    int total,a=5,b=10;
    total=ADDNUM(a,b);
```

```

        printf("Total=%d\n",total);
}

```

Now we will see how it will be in memory.

5
10
Return
Old BP

BP+6
BP+4
BP+2
BP

Small, Compact

5
10
Return
Old BP

BP+8
BP+6
BP

Medium, Large, Huge

Suppose we are taking small memory model then we can see position of a is BP+6 and that of b is BP+4 in stack. Now we can use this in assembly procedure. The assembly procedure for this will be as-

ADD.ASM

```

.MODEL SMALL
.CODE
    PUBLIC _ADDDNUM
._ADDDNUM PROC
    PUSH BP
    MOV BP, SP
    MOV AX, [BP+4]
    MOV BX, [BP+6]
    ADD AX, BX
    POP BP
    RET
._ADDDNUM ENDP
END

```

Now we can link assembly procedure with C program as-

```

d:\>tasm -mx ADDNUM.ASM
d:\> tcc MAIN.c ADDNUM.obj

```

Chapter 18

Library Functions

18.1 Mathematical Functions

The header file math.h contains declarations for several exponential, logarithmic, trigonometric, hyperbolic, and other mathematical functions. In all trigonometric functions, angles are taken in radians.

18.1.1 abs()

Declaration: `int abs(int x);`

It returns the absolute value of the argument. If the argument is positive then it returns the positive value, if the argument is negative then it negates the value and returns the positive value.

18.1.2 acos()

Declaration: `double acos(double x);`

This function returns the arc cosine of x i.e. $\cos^{-1}x$. The argument should be in the range -1 to 1 and the return value lies in the range 0 to π .

18.1.3 asin()

Declaration: `double asin(double x);`

This function returns the arc sine of x i.e. $\sin^{-1}x$. The argument should be in the range -1 to 1 and the return value lies in the range $-\pi/2$ to $\pi/2$.

18.1.4 atan()

Declaration: `double atan(double x);`

This function returns the arc tangent of x i.e. $\tan^{-1}x$. The argument should be in the range -1 to 1 and the return value lies in the range $-\pi/2$ to $\pi/2$.

18.1.5 atan2()

Declaration: `double atan2(double y, double x);`

This function is used for computing the arc tangent of y/x i.e. $\tan^{-1}(y/x)$. The return value lies in the range $-\pi$ to π .

18.1.6 cabs()

Declaration: `double cabs(struct complex x);`

This function returns the absolute value of complex number.

18.1.7 ceil()

Declaration: double ceil(double x);

This function finds the smallest integer not less than x and returns it as a double. For example-

x = 2.4	return value 3.0
x = -2.4	return value -2.0

18.1.8 cos()

Declaration: double cos(double x);

This function returns the trigonometric cosine of x. The value of the argument must be in radians and return value lies between -1 and 1.

18.1.9 cosh()

Declaration: double cosh(double x);

This function returns the hyperbolic cosine of x.

18.1.10 exp()

Declaration: double exp(double x);

This function returns the value of e^x .

18.1.11 fabs()

Declaration: double fabs(double x);

This function is same as abs() function but it computes the absolute value of floating point number.

18.1.12 floor()

Declaration: double floor(double x);

This function finds the largest integer not greater than x and returns it as a double. For example-

x = 2.4	return value 2.0
x = -2.4	return value -3.0

18.1.13 fmod()

Declaration: double fmod(double x, double y);

It returns the floating remainder of x/y. The sign of the result is the same as that of x. If the value of y is zero, the result is implementation defined.

x = 7.0, y = 4.0	return value 3.0
x = 4.6, y = 2.2	return value 0.2

18.1.14 frexp()

Declaration: double frexp(double arg, int *expr);

This function splits the first argument into mantissa and exponent such that

$$\text{arg} = \text{mantissa} \times 2^{\text{exponent}}$$

It returns the value of mantissa whose range is [0.5, 1), and the value of exponent is stored in the variable pointed to by expr i.e. *expr gives the value of exponent. If the value of arg is 0

then the function returns 0 and the value stored in *expr is also 0.

18.1.15 ldexp()

Declaration: double ldexp(double arg, int exp);

This function is used for obtaining the value of the expression $\text{arg} * 2^{\text{exp}}$.

18.1.16 log()

Declaration: double log(double arg);

This function returns the natural logarithm(base e) of argument.

18.1.17 log10()

Declaration: double log10(double arg);

This function returns the base 10 logarithm of argument.

18.1.18 modf()

Declaration: double modf(double arg , double *ptr);

This function splits the first argument arg into integer and fractional part. Both parts have the same sign as arg. The fractional part is returned by the function, and the integer part is stored in the variable pointed to by ptr .

18.1.19 pow()

Declaration: double pow(double x, double y);

This function returns the value of x^y .

18.1.20 sin()

Declaration: double sin(double arg);

This function returns the trigonometric sine of the argument, where the argument is in radians.

18.1.21 sinh()

Declaration: double sinh(double x);

This function returns the hyperbolic sine of x.

18.1.22 sqrt()

Declaration: double sqrt(double x);

This function returns the square root of x..

18.1.23 tan()

Declaration: double tan(double x);

This function returns the trigonometric tangent of x.

18.1.24 tanh()

Declaration: double tanh(double x);

This function returns the hyperbolic tangent of x.

18.2 Character Type Functions

Header File: ctype.h

18.2.1 isalnum()

Declaration: int isalnum(int arg);

This macro returns a nonzero value if argument is alphanumeric(a...z, A...Z, 0...9), otherwise it returns zero.

18.2.2 isalpha()

Declaration: int isalpha(int arg);

This macro returns a nonzero value if argument is alphabetic(a...z, A...Z), otherwise it returns zero.

18.2.3 iscntrl()

Declaration: int iscntrl(int arg);

The iscntrl() returns a non zero value if the argument is a control character, otherwise it returns zero.

18.2.4 isdigit()

Declaration: int isdigit(int arg);

This macro returns a non zero value if the argument is a digit (0...9), otherwise it returns zero.

18.2.5 isgraph()

Declaration: int isgraph(int arg);

This macro returns a non zero value if the argument is a graphic character (any printing character except a space), otherwise it returns zero.

18.2.6 islower()

Declaration: int islower(int arg);

This macro returns a non zero value if the argument is a lowercase character (a...z), otherwise it returns zero.

18.2.7 isprint()

Declaration: int isprint(int arg);

This macro returns a non zero value if the argument is printable character (including space), otherwise it returns zero.

18.2.8 ispunct()

Declaration: int ispunct(int arg);

This macro returns a nonzero value if the argument is punctuation character, otherwise it returns zero.

18.2.9 isspace()

Declaration: int isspace(int arg);

This macro returns a nonzero value if the argument is white space character (space, horizontal

tab, vertical tab, form feed, new line, carriage return), otherwise it returns zero.

18.2.10 isupper()

Declaration: int isupper(int arg);

This macro is used to check whether a given character is uppercase character (A....Z) or not. It returns nonzero if the argument is uppercase character, otherwise it returns zero.

18.2.11 isxdigit()

Declaration: int isxdigit(int arg);

This macro is used to check whether a given character is hexadecimal digit(A....F, a....f, 0....9) or not. It returns nonzero if the argument is hexadecimal digit, otherwise it returns zero.

18.2.12 tolower()

Declaration: int tolower(int arg);

This macro is used to convert an uppercase character(A....Z) into equivalent lowercase character(a....z). It returns lowercase character if the argument is an uppercase character, otherwise it returns unchanged value.

18.2.13 toupper()

Declaration: int toupper(int arg);

The macro toupper() is used to convert a lowercase character(a....z) into equivalent uppercase character(A....Z). It returns uppercase character if the argument is a lowercase character, otherwise it returns unchanged value.

18.3 String Manipulation Functions

Header file: string.h

18.3.1 strcat()

Declaration: char *strcat(char *str1, const char *str2);

This function is used to concatenate two strings. The null character from str1 is removed and str2 is concatenated at the end of str1. A pointer to the concatenated strings is returned by the function.

18.3.2 strchr()

Declaration: char *strchr(const char *str, int ch);

This function returns the pointer to the first occurrence of the character ch in the string str. If the character is not present in the string then it returns NULL.

```
#include<stdio.h>
#include<string.h>
main()
{
    char *p;
    p=strchr("Multinational", 'n');
    printf("%s\n", p);
}
```

Output:

```
national
```

18.3.3 strcmp()

Declaration: char *strcmp(const char *str1, const char *str2);

This function is used to compare two strings lexicographically.

strcmp(str1, str2) returns a value-

< 0	when str1 < str2
= 0	when str1 == str2
> 0	when str1 > str2

18.3.4 strcpy()

Declaration: char *strcpy(char *str1, char *str2);

This function copies string str2 to str1, including the null character, and returns a pointer to the first string.

18.3.5 strcspn()

Declaration: size_t strcspn(const char *str1, const char *str2)

This function returns the index of the first character of str1, which is matched with any character of str2. For example-

strcspn("abcmnop", "lmn");	will return 3
strcspn("abcmnop", "ln");	will return 4
strcspn("1234ABCD", "COT");	will return 6

18.3.6 strlen()

Declaration: size_t strlen(const char *str);

It returns the number of characters in the string, excluding the null character.

18.3.7 strncat()

Declaration: char *strncat(char *str1, const char *str2, size_t length);

This function is same as strcat() but it concatenates only a portion of a string to another string.

The null character from str1 is removed and *length* characters of str2 are appended at the end of str1, and at last a null character is added at the end of str1. This function returns str1.

```
#include<stdio.h>
#include<string.h>
main()
{
    char str1[15] = "ABC";
    strncat(str1, "DEFGHIJ", 4);
    printf("%s\n", str1);
}
```

Output:

```
ABCDEFG
```

18.3.8 strcmp()

Declaration: int *strcmp(const char *arr1, const char *arr2, size_t length);

Header file: string.h

This function is similar to strcmp() but it compares the characters of the strings upto specified length.

strcmp(str1, str2, len) returns a value-

- < 0 when str1 < str2
- = 0 when str1 == str2
- > 0 when str1 > str2

```
#include<stdio.h>
#include<string.h>
main()
{
    int l,m,n,p;
    l=strncmp("Deepali","Deepanjali",4);
    m=strncmp("Deepali","Deepanjali",6);
    n=strncmp("Suresh","Suraiya",3);
    p=strncmp("Suresh","Suraiya",5);
    printf("%d\t%d\t%d\t%d\n",l,m,n,p);
}
```

Output:

0 -2 0 4

18.3.9 strcpy()

Declaration: int *strcpy(char *str1, const char *str2, size_t length);

It is same as strcpy() but it copies the characters of the string to the specified length. Here str2 should be a null terminated string and str1 should be an array.

If str2 has more than *length* characters, then str1 might not be null terminated.

If str2 has less than *length* characters, then null characters will be added to str1 at the end, till the specified length of characters have been copied.

```
#include<stdio.h>
#include<string.h>
main()
{
    char str1[10];
    strcpy(str1,"Departmental",6);
    str1[6]='\0';
    printf("%s\n",str1);
}
```

Output:

Depart

18.3.10 strpbrk()

Declaration: char *strpbrk(const char *str1, const char *str2);

This function returns the pointer to first character of string str1, which matches with any character of str2. It returns NULL if there are no common characters in the two strings.

```
#include<stdio.h>
#include<string.h>
main()
{
    char *p1,*p2,*p3;
    p1=strpbrk("abcmnop","lmn");
    p2=strpbrk("abcmnop","ln");
    p3=strpbrk("1234ABCD","COT");
    printf("%s\t%s\t%s\n",p1,p2,p3);
}
```

Output:

mnop nop CD

18.3.11 strrchr()

Declaration: char *strrchr(const char *str, int ch);

This function returns a pointer to the last occurrence of the character ch in the string pointed to by str, otherwise it returns NULL.

```
#include<stdio.h>
#include<string.h>
main()
{
    char *p;
    p=strrchr("Multinational",'n');
    printf("%s\n",p);
}
```

Output:

nal

18.3.12 strspn()

Declaration: size_t strspn(const char *str1, const char *str2);

This function returns the index of the first character of str1, which does not match with any character from str2. For example-

strspn("cindepth", "datastructure")	will return 1
strspn("abcdefghijklm", "completedatabase")	will return 5
strspn("1234ABCD", "AB12")	will return 2

18.3.13 strstr()

Declaration: char *strstr(const char *arr1, const char *arr2);

This function is used to locate the first occurrence of a substring in another string. The sec

argument is a pointer to the substring and the first argument is a pointer to the string in which the substring is to be searched. This function returns a pointer to the beginning of the first occurrence of the substring in another string. If the substring is not present then it returns NULL.

```
#include<stdio.h>
#include<string.h>
main()
{
    char *ptr;
    ptr=strstr("placement section", "cement");
    printf("%s",ptr);
}
```

Output:

cement section

18.4 Input/Output Functions

18.4.1 access()

Declaration: int access(const char *fname, int mode);

Header file: io.h

This function is used to see the existence of file. It also checks the type of file. The possible values of mode are as given below-

mode	value
0	existence of file
1	executable file
2	write access
4	read access
6	read/write access

18.4.2 chmod()

Declaration: int chmod(const char *fname, int mode);

Header file: io.h

This function is used for changing the access mode of the file to the value of the mode. These modes are as given below-

mode	meaning
S_IREAD	read access
S_WRITE	write access
S_IREAD S_IWRITE	read and write access

18.4.3 clearerr()

Declaration: void clearerr(FILE *stream);

Header file: stdio.h

This function is used to set the end of file and error indicators to 0.

18.4.4 close()

Declaration: int close(int fd);

Header file: io.h

This function is called with a file descriptor which is created with a successful call to open() or creat(). It closes the file and flushes the write buffer. It returns zero on success otherwise it returns -1.

18.4.5 creat()

Declaration: int creat(const char *fname, int pmode);

Header file: io.h

This function creates a new file for writing. The file descriptor returns a positive value on success otherwise it returns -1. The permission mode determines the file access setting. The values of pmode are as:

permission mode	meaning
S_IREAD	read only
S_IWRITE	write only

18.4.6 fclose()

Declaration: int fclose(FILE *fptr);

Header file: stdio.h

This function is used to close a file that was opened by fopen(). It returns EOF on error and 0 on success.

18.4.7 feof()

Declaration: int feof(FILE *fptr);

Header file: stdio.h

The macro feof() is used to check the end of file condition. It returns a nonzero value if end of file has been reached otherwise it returns zero.

18.4.8 perror()

Declaration: int perror(FILE *fptr);

Header file: stdio.h

The macro perror() is used for detecting any error occurred during read or write operations on a file. It returns a nonzero value if an error occurs i.e. if the error indicator for that file is set otherwise it returns zero.

18.4.9 fflush()

Declaration: int fflush(FILE *fptr);

Header file: stdio.h

This function writes any buffered unwritten output to the file associated with fptr. On success it returns 0 and on error it returns EOF.

18.4.10 fgetc()

Declaration: int fgetc(FILE *fptr);

Header file: stdio.h

This function reads a single character from a given file and increments the file pointer position. On success it returns the character after converting it to an int without sign extension. On end of file or error it returns EOF.

18.4.11 fgets()

Declaration: char fgets(char *str, int n, FILE *fptr);

Header file: stdio.h

This function is used to read characters from a file and these characters are stored in the string pointed to by str. It reads at most n-1 characters from the file where n is the second argument. fptr is a file pointer which points to the file from which characters are read. This function returns the string pointed to by str on success, and on error or end of file it returns NULL.

18.4.12 fileno()

Declaration: int fileno(FILE *stream);

Header file: stdio.h

The function fileno() is used for returning the file descriptor of the specified file pointer.

18.4.13 fopen()

Declaration: FILE *fopen(const char *fname, const char *mode);

Header file: stdio.h

The function fopen() is used to open a file. It takes two strings as arguments, the first one is the name of the file to be opened and the second one is the mode that decides which operations (read, write, append etc) are to be performed on the file. On success, fopen() returns a pointer of type FILE and on error it returns NULL.

18.4.14 fprintf()

Declaration: fprintf (FILE *fptr, const char *format [, argument, ...]);

Header file: stdio.h

This function is same as the printf() function but it writes formatted data into the file instead of the standard output. This function has same parameters as in printf() but it has one additional parameter which is a pointer of FILE type, that points to the file to which the output is to be written. It returns the number of characters output to the file on success, and EOF on error.

18.4.15 fputc()

Declaration: int fputc(int ch, FILE *fptr);

Header file : stdio.h

This function writes a character to the specified file at the current file position and then increments the file position pointer. On success it returns an integer representing the character written, and on error it returns EOF.

18.4.16 fputs()

Declaration: int fputs(const char *str, FILE *fptr);

Header file: stdio.h

This function writes the null terminated string pointed to by str to a file. The null character that marks the end of string is not written to the file. On success it returns the last character written and on error it returns EOF.

18.4.17 fread()

Declaration: size_t fread(void *ptr, size_t size, size_t n, FILE *fptr);

Header file: stdio.h

This function is used to read an entire block from a given file. Here ptr is a pointer which points to the block of memory which receives the data read from the file, size is the length of each item in bytes, n is the number of items to be read from the file and fptr is a file pointer which points to the file from which data is read. On success it reads n items from the file and returns n, if error or end of file occurs then it returns a value less than n.

18.4.18 fputchar()

Declaration: int fputchar(int arg);

Header file: stdio.h

This function is used for writing a character to standard output.

18.4.19 fscanf()

Declaration: fscanf (FILE *fptr, const char *format [, address, ...]);

Header file: stdio.h

This function is similar to the scanf () function but it reads data from file instead of standard input , so it has one more parameter which is a pointer of FILE type and it points to the file from which data will be read. It returns the number of arguments that were assigned some values on success, and EOF at the end of file.

18.4.20 fseek()

Declaration: int fseek(FILE *fp , long disp, int position);

Header file: stdio.h

This function is used for setting the file position pointer at the specified byte. Here fp is a file pointer, displacement is a long integer which can be positive or negative and it denotes the number of bytes which are skipped backward (if negative) or forward (if positive) from the position specified in the third argument. The third argument can take one of these three values.

Constant	Value	Position
SEEK_SET	0	Beginning of file
SEEK_CURRENT	1	Current position
SEEK_END	2	End of file

18.4.21 fstat()

Declaration: int fstat(int fd, struct stat *finfo);

Header file: sys/stat.h

This function is used for filling the file information in structure finfo. The structure of finfo is same as stat which is defined in sys/stat.h.

18.4.22 ftell()

Declaration: int fputc(FILE *stream);

Header file: stdio.h

This function returns the current position of the file position pointer. The value is counted from the beginning of the file.

18.4.23 isatty()

Declaration: int isatty(int handle);

Header file: io.h

This function is used for checking whether the handle is associated with character device or not.

The character device may be a terminal, console, printer or serial port. It returns nonzero on success, otherwise it returns zero.

18.4.24 open()

Declaration: int open(const char *fname, int access, unsigned mode);

Header file: io.h

This function is used for opening a file. The access is the base mode with modifier. modifier is taken by applying OR operator with base mode.

Base mode	Meaning
O_WRONLY	write only
O_RDONLY	read only
O_RDWR	read / write

Modifiers	Meaning
O_APPEND	For appending
O_BINARY	binary file
O_CREAT	creates the file, if the file doesn't exist and sets the attribute with mode

O_EXCL	when it is used with O_CREAT it doesn't create file, if the file already exists.
O_NDELAY	Unix compatible
O_TEXT	text file
O_TRUNC	Truncates to length 0 if file doesn't exist

The parameter mode is used with only O_CREAT modifier.

Mode	Meaning
S_IREAD	Read
S_IWRITE	Write
S_READ / S_IWRITE	Read / Write

18.4.25 read()

Declaration: int read(int fd, void *arr, unsigned num);

Header file: io.h

This function is used for reading the files. Here fd is the file descriptor, num is the number of bytes read from the file and arr points to the array which has the number of bytes read from file. The file pointer position is incremented by the number of bytes read from file. If end of file occurs then number of bytes read will be smaller than num.

18.4.26 remove()

Declaration: int remove(const char *fname);

Header file: stdio.h

This function is used for deleting the file. It returns 0 on success and -1 on error.

18.4.27 rename()

Declaration: int rename(const char *old, const char *new)

Header file: stdio.h

This function is used to rename a file. We can give full path also in the argument, but the drives should be the same although directories may differ. On success it returns 0 and on error it returns -1.

18.4.28 setbuf()

Declaration: void setbuf(FILE *fptr, char *buffer);

Header file: stdio.h

The function setbuf() specifies the buffer that will be used by a file. The size of the buffer should be BUFSIZ bytes, which is defined in stdio.h. If setbuf() is called with null, then the I/O will be unbuffered.

18.4.29 `sopen()`

Declaration: `int sopen(const char *fname, int access, int shmode, int mode)`

Header file: io.h

The macro `sopen()` is used for opening a file in shared mode. This is used in network environment. The access is the base mode with modifier. Modifier is taken by applying OR operator with base mode.

Base mode	Meaning
<code>O_WRONLY</code>	write only
<code>O_RDONLY</code>	read only
<code>O_RDWR</code>	read / write

Modifiers	Meaning
<code>O_APPEND</code>	For appending
<code>O_BINARY</code>	binary file
<code>O_CREAT</code>	creates the file, if the file doesn't exist and sets the attributes with mode
<code>O_EXCL</code>	when it is used with <code>O_CREAT</code> it doesn't create file, if the file already exists.
<code>O_NDELAY</code>	
<code>O_TEXT</code>	
<code>O_TRUNC</code>	text file

`shmode` is the share mode with the file. These are-

shmode	Meaning
<code>SH_COMPACT</code>	Compatibility mode
<code>SH_DENYRD</code>	Not allowed for read
<code>SH_DENYWR</code>	Not allowed for write
<code>SH_DENYRW</code>	Not allowed for read / write
<code>SH_DENYNO</code>	Allowed for read / write
<code>SH_DENYNONE</code>	Allowed for read / write

The parameter mode is used with only `O_CREAT` modifier. The modes are-

Base mode	Meaning
S_IREAD	Read
S_IWRITE	Write
S_READ / S_IWRITE	Read / Write

18.4.30 stat()

Declaration: int stat(char *fname, struct stat *finfo);

Header file: sys\stat.h

This function is used for filling the file information in structure finfo. The structure of finfo is same as stat which is defined in sys\stat.h.

18.4.31 sprintf()

Declaration: int sprintf (char *str, const char *controlstring [, arg1, arg2,]);

Header file: stdio.h

This function is same as printf () function except that instead of sending the formatted output to the screen, it stores the formatted output in a string . This function returns the number of characters output to the string excluding the null character and on failure it returns EOF.

18.4.32 sscanf()

Declaration: int sscanf (const char *str, const char *controlstring [, address1, address2,.....]);

Header file: stdio.h

This function is same as the scanf () function except that data is read from a string rather than the standard input. We can read the formatted text from a string and convert it into variables of different data types.

18.4.33 tell()

Declaration: long tell(int fd)

Header file: io.h

This is same as ftell(). It is used for getting the current file pointer position.

18.4.34 tmpfile()

Declaration: FILE *tmpfile(void)

Header file: stdio.h

tmpfile() creates a temporary binary file and opens it in update mode(wb+). On success it returns a pointer of type FILE that points to the temporary file and on error it returns NULL. The file created by tmpfile() is temporary because it is automatically deleted when closed or when the program terminates.

18.4.35 tmpnam()

char *tmpnam(char *arr)

Header file: stdio.h

This function is used for generating a unique file name. The number of different file names that

can be generated by tmpnam() during the execution of program is specified by TMP_MAX, which is defined in stdio.h file. This file name is stored in the array, which is pointed to by arr.

18.4.36 unlink()

Declaration: int unlink(const char *fname)

Header file: stdio.h

This function is used for deleting a file from the directory. We can give full pathname also as the argument. The file should be closed before deleting. On success it returns 0 and on error it returns -1 and errno is set to one of these values:

ENOENT Path or file name not found

EACCES Permission denied

Appendix A

ASCII Characters

ASCII value	Character	ASCII value	Character	ASCII value	Character
000	NUL	049	1	098	b
001	SOH	050	2	099	c
002	STX	051	3	100	d
003	ETX	052	4	101	e
004	EOT	053	5	102	f
005	ENQ	054	6	103	g
006	ACK	055	7	104	h
007	BEL	056	8	105	i
008	BS	057	9	106	j
009	HT	058	:	107	k
010	LF	059	;	108	l
011	VT	060	<	109	m
012	FF	061	=	110	n
013	CR	062	>	111	o
014	SO	063	?	112	p
015	SI	064	@	113	q
016	DLE	065	A	114	r
017	DC1	066	B	115	s
018	DC2	067	C	116	t
019	DC3	068	D	117	u
020	DC4	069	E	118	v
021	NAK	070	F	119	w
022	SYN	071	G	120	x
023	ETB	072	H	121	y
024	CAN	073	I	122	z
025	EM	074	J	123	{
026	SUB	075	K	124	
027	ESC	076	L	125	}
028	FS	077	M	126	~
029	GS	078	N	127	DEL
030	RS	079	O		
031	US	080	P		
032	blank	081	Q		
033	!	082	R		

ASCII Characters

ASCII value	Character	ASCII value	Character
034	"	083	S
035	#	084	T
036	\$	085	U
037	%	086	V
038	&	087	W
039	,	088	X
040	(089	Y
041)	090	Z
042	*	091	[
043	+	092	\
044	,	093]
045	-	094	^
046	.	095	_
047	/	096	~
048	0	097	a

* Characters from 1 – 32 and character 127 are control characters.

Value	Character
b	
c	
d	
e	
f	
g	
h	
i	
j	
k	
l	
m	
n	
o	
p	
q	
r	
s	
t	
u	
v	
w	
x	
y	
z	
{	
}	
~	
DEL	

Appendix B

Precedence and Associativity

Operator	Description	Precedence
()	Function call	
[]	Array subscript	1
→	Arrow operator	
.	Dot operator	
+	Unary plus	
-	Unary minus	
++	Increment	
--	Decrement	
!	Logical NOT	2
~	One's complement	
*	Indirection	
&	Address	
(datatype)	Type cast	
sizeof	Size in bytes	
*	Multiplication	
/	Division	3
%	Modulus	
+	Addition	4
-	Subtraction	
<<	Left shift	5
>>	Right shift	
<	Less than	
<=	Less than or equal to	6
>	Greater than	
>=	Greater than or equal to	
= =	Equal to	7
!=	Not equal to	
&	Bitwise AND	8
^	Bitwise XOR	9
	Bitwise OR	10
&&	Logical AND	11
	Logical OR	12
? :	Conditional operator	13

= *= /= %= += -= & = ^ = =<<= >>= ,	Assignment operators Comma operator	14 15	Right to Left Left to Right
---	--	--------------	------------------------------------

Appendix C

Conversion Specifications

%c	- a single character
%d	- a decimal integer
%f	- a floating point number
%e	- a floating point number
%g	- a floating point number
%lf	- long range of floating point number(for double data type)
%h	- a short integer
%o	- an octal integer
%x	- a hexadecimal integer
%i	- a decimal, octal or hexadecimal integer
%s	- a string
%u	- an unsigned decimal integer

The modifier h can be used before conversion specifications d, i, o, u, x to specify short integer and the modifier l can be used before them to specify a long integer.

The modifier l can be used before conversion specifications f,e,g to specify double while modifier L can be used before them to specify a long double.

Key	Normal	Shift	Ctrl	Alt
A	1E61	1E41	1E01	1E00
B	3062	3042	3002	3000
C	2E63	2E42	2E03	2E00
D	2064	2044	2004	2000
E	1265	1245	1205	1200
F	2166	2146	2106	2100
G	2267	2247	2207	2200
H	2368	2348	2308	2300
I	1769	1749	1709	1700
J	246A	244A	240A	2400
K	256B	254B	250B	2500
L	266C	264C	260C	2600
M	326D	324D	320D	3200
N	316E	314E	310E	3100
O	186F	184F	180F	1800
P	1970	1950	1910	1900
Q	1071	1051	1011	1000
R	1372	1352	1312	1300
S	1F73	1F53	1F13	1F00
T	1474	1454	1414	1400
U	1675	1655	1615	1600
V	2F76	2F56	2F16	2F00
W	1177	1157	1117	1100
X	2D78	2D58	2D18	2D00
Y	1579	1559	1519	1500
Z	2C7A	2C5A	2C1A	2C00
1	0231	0221	7800	
2	0332	0340	0300	7900
3	0433	0423	7A00	
4	0534	0524	7B00	
5	0635	0625	7C00	
6	0736	075E	071E	
7	0837	0826	7E00	
8	0938	092A	7F00	

9	0A39	0A28	8000	
0	0B30	0B29	8100	
-	0C2D	0C5F	0C1F	8200
=	0D3D	0D2B	8300	
[1A5B	1A7B	1A1B	1A00
]	1B5D	1B7D	1B1D	1B00
;	273B	273A	2700	
,	2827	2822		
'	2960	297E		
\	2B5C	2B7C	2B1C	2600 (same as Alt L)
,	332C	333C		
:	342E	343E		
/	352F	353F		
F1	3B00	5400	5E00	6800
F2	3C00	5500	5F00	6900
F3	3D00	5600	6000	6A00
F4	3E00	5700	6100	6B00
F5	3F00	5800	6200	6C00
F6	4000	5900	6300	6D00
F7	4100	5A00	6400	6E00
F8	4200	5B00	6500	6F00
F9	4300	5C00	6600	7000
F10	4400	5D00	6700	7100
F11	8500	8700	8900	8B00
F12	8600	8800	8A00	8C00
BackSpace	0E08	0E08	0E7F	0E00
Del	5300	532E	9300	A300
Down Arrow	5000	5032	9100	A000
End	4F00	4F31	7500	9F00
Enter	1C0D	1C0D	1C0A	A600
Esc	011B	011B	011B	0100
Home	4700	4737	7700	9700
Ins	5200	5230	9200	A200
Keypad 5	4C35	8F00		
Keypad *	372A	9600	3700	
Keypad -	4A2D	4A2D	8E00	4A00
Keypad +	4E2B	4E2B	4E00	
Keypad /	352F	352F	9500	A400
LeftArrow	4B00	4B34	7300	9B00
PgDn	5100	5133	7600	A100
PgUp	4900	4939	8400	9900
PrtSc	7200			
RightArrow	4D00	4D36	7400	9D00
SpaceBar	3920	3920	3920	3920
Tab	0F09	0F00	9400	A500
Up Arrow	4800	4838	8D00	9800

Index

!, 40
#define, 378
#else and #elif, 390
#error, 399
#if And #endif, 390
#ifdef and #ifndef, 393
#line, 399
#pragma, 400
#undef, 387
&, 408
&&, 39
^, 409
|, 408
||, 40
~, 410
<<, 411
>>, 411

A

abs(), 505
access(), 519
acos(), 505
actual arguments, 118
address operator, 197
AND (&&) Operator, 39
argc, 367
argv[], 367
arithmetic operators, 32
 binary arithmetic operators, 32
 unary arithmetic operators, 32
arrays, 158
 arrays and functions, 165, 174
 more than two dimensions, 173
 one dimensional array, 158
 two dimensional array, 167
asin(), 505
assembly and C interaction, 495
 inline assembly language, 495
 linking of two assembly Files, 498

linking assembly procedure in C Program, 502
assignment operators, 35
associativity of operators, 47
atan(), 505
atan2(), 505
automatic type conversion, 44
automatic, 437

B

binary search, 176
bit fields, 426
bitwise operators, 407
 bitwise AND (&), 408
 bitwise left shift (<<), 411
 bitwise OR (|), 408
 bitwise right shift (>>), 411
 bitwise XOR (^), 409
 one's complement (~), 410
break statement, 78
bubble sort, 181
buffer, 335

C

C character set, 7
cabs(), 505
calloc(), 233
ceil(), 506
character I/O, 339
character type functions, 508
characteristics of C, 4
chmod(), 513
clearerr(), 364, 514
close(), 514
code optimization, 487
comma operator, 42
command line arguments, 367
comments, 16
compilation of C programs, 457
compound statement or block, 58

conditional compilation, 389
 conditional operator, 41
 const, 447
 constants, 10
 character constants, 13
 numeric constants, 11
 string constants, 13
 symbolic constants, 13
 continue statement, 80
 control statements, 58
 conversion specifications, 17
`cos()`, 506
`cosh()`, 506
`creat()`, 514

D

data types, 10
 delimiters, 9
 design methods, 1
 bottom-up design, 2
 modular approach, 2
 top-down design, 1
 do...while loop, 69
 dynamic memory allocation, 231

E

enumeration, 433
 environment for C, 5
 MS-DOS Environment, 5
 Unix Environment, 5
 escape sequences, 8
 evaluation of operands, 53
 execution characters, 8
`exp()`, 506
 expressions, 15
 external, 439

F

`fabs()`, 506
`fclose()`, 514
`feof()`, 362,514
`ferror()`, 363,514
`fflush()`, 366,515
`fgetc()`, 340,515
`fgets()`, 342,515
`fileno()`, 515
 files, 334

block read / write, 345
 character I/O, 339
 closing a file, 337
 end of file, 338
 formatted I/O, 343
 integer I/O, 341
 opening a file, 335
 other file functions, 362
 predefined file pointers, 339
 random access to file, 348
 string I/O, 342
 structure of a general file program, 338
 text and binary modes, 334
 floating-point arithmetic, 34
`floor()`, 506
`fmod()`, 506
`fopen()`, 515
 for loop, 71
 formatted input and output, 24
 floating point numeric input, 26
 floating point numeric output, 27
 format for integer input, 24
 format for integer output, 25
 format for string input, 27
 format for string output, 28
`fprintf()`, 343,515
`fputc()`, 339,516
`fputchar()`, 516
`fputs()`, 342,516
`fread()`, 347,516
`free()`, 234
`freopen()`, 367
`frexp()`, 506
`fscanf()`, 344,516
`fseek()`, 349,516
`fstat()`, 517
`ftell()`, 350,517
 functions, 110
 advantages, 110
 library functions, 110
 user-defined functions, 111
 definition, 112
 call, 113
 declaration, 114,124, 126
 arguments, 118, 125
 types, 120
 variable number of arguments, 450
`fwrite()`, 345

G

`getc()` and `putc()`, 340
`getchar()`, 29
`getw()`, 341
global Variables, 131
`goto`, 82

H

high level languages, 3
history of C, 3

I

identifiers, 9
`if...else`, 59
 Nesting of `if...else`, 61
 else if Ladder, 63
including files, 389
increment and decrement operators, 35
 postfix increment / decrement, 36
 prefix increment / decrement, 36
input/output functions, 513
insertion sort, 184
integer arithmetic, 33
`isalnum()`, 508
`isalpha()`, 508
`isatty()`, 517
`iscntrl()`, 508
`isdigit()`, 508
`isgraph()`, 508
`islower()`, 508
`isprint()`, 508
`ispunct()`, 508
`isspace()`, 508
`isupper()`, 509
`isxdigit()`, 509

K

keywords, 9

L

`ldexp()`, 507
library creation, 484
 creation of library in turbo C, 482
 creation of library in Unix, 486
library functions, 110, 126
linkage, 445

linked list, 309

 creation of list, 314
deletion from a linked list, 313
deletion of a node in between, 314
deletion of first node, 313
insertion in between or at the end, 312
insertion in the beginning, 312
insertion into a linked list, 311
reversing a linked list, 318
searching in a linked list, 311
traversing a linked list, 311
local variables, 130
`log()`, 507
`log10()`, 507
logical or boolean operators, 39
 AND (`&&`) operator, 39
 OR (`||`) operator, 40
 NOT (`!`) operator, 40
loops, 65
 while loop, 65
 do...while loop, 69
 for loop, 71
 Nesting Of Loops, 75
 Infinite Loops, 77
lvalue, 456

M

macros, 379
 generic functions, 386
macros vs functions, 385
nesting in macros, 381
predefined macro names, 398
problems with macros, 382
magic matrix, 188
`main()` function, 125
`malloc()`, 231
masking, 413
 masking using bitwise AND, 413
 masking using bitwise OR, 415
 masking using bitwise XOR, 415
switching off bits using Bitwise AND and Complement Operator, 416
memory during program execution, 445
memory models, 500
memory, 196
merging of arrays, 185
mixed mode arithmetic, 34
`modf()`, 507

N

NOT (!) Operator, 40
 null directive, 400
 numeric constant, 11

O

one dimensional array, 158
 open(), 517
 OR (||) Operator, 40

P

pascal triangle, 187
 perror(), 364
 pointers, 196
 array of pointers, 227
 function returning pointer, 222
 pointer arithmetic, 201
 pointer comparisons, 206
 pointer to an array, 212
 pointer to pointer, 206
 pointers and functions, 219
 pointers and one dimensional arrays, 208
 pointers and three dimensional arrays, 217
 pointers and two dimensional arrays, 213
 pointers to functions, 238
 pointers variables, 197
 subscripting pointer to an array, 216
 subscripting pointer variables, 211

pow(), 507
 precedence of operators, 47
 preprocessor, 377
 programming languages, 2
 high-level languages, 3
 low level languages, 2

project building, 464
 building project in turbo C, 481
 building project on unix, 483
 coding, 468
 detail design, 465
 requirement analysis, 464
 testing, 481
 top level design, 465

putchar(), 29
 putw (), 341

R

read(), 518

realloc(), 233

recursion, 132

 tower of hanoi, 136

 advantages and disadvantages of recursion, 139
 local variables in recursion, 139

register, 444

relational operators, 37

remove(), 365,518

rename(), 364,518

reserved words, 9

return statement, 116

rewind(), 351

rvalue, 456

S

selection sort, 180

self referential structures, 309

setbuf(), 518

sin(), 507

sinh(), 507

sizeof operator, 43

spiral matrix, 189

sprintf(), 272,520

sqrt(), 507

sscanf(), 273,520

stat(), 520

statements, 15

static Variables, 132,442

storage classes in functions, 445

storage classes, 437

strcat(), 261,509

strchr(), 509

strcmp(), 258,510

strcpy(), 259,510

strcspn(), 510

string manipulation functions, 509

stringizing operator (#), 387

strings, 175, 253

 array of pointers to strings, 267

 array of strings or two dimensional array of characters, 264

 string constant or string literal, 253

 string library functions, 257

 string pointers, 262

 string variables, 255

strlen(), 257,510

strncat(), 510

strncmp(), 511

strncpy(), 511

stripbrk(), 512
strrchr(), 512
strspn(), 512
strstr(), 512
structure, 288
 accessing members of a structure, 291
 array of structures, 293
 arrays within structures, 295
 assignment of structure variables, 292
 declaring structure variables, 289
 defining a structure, 288
 initialization of structure variables, 290
 nested structures, 296
 pointers to structures, 298
 pointers within structures, 299
 size of structure, 293
 storage of structures in memory, 292
 structures and functions, 299
suppression character in scanf(), 28
switch, 84

T

tan(), 507
tanh(), 507
tell(), 520
tmpfile(), 366,522
tmpnam(), 366,520
token pasting operator(##), 388
tolower(), 509
toupper(), 509
translators, 3

trigraph characters, 8
two dimensional array, 167
type conversion, 44
 implicit type conversions, 44
 automatic conversions, 44
 type conversion in assignment, 45
 explicit type conversion, 46
typedef, 326

U

union, 321
unlink(), 365,521
user-defined functions, 111

V

va_arg, 450
va_end, 450
va_list, 450
va_start, 450
variable number of arguments, 450
variables, 14
 declaration of variables, 14
 initialisation of variables, 14
vfprintf(), 455
void pointers, 229
volatile, 449
vprintf(), 455
vsprintf(), 455

W

while loop, 65

C in Depth

The book explains each topic in depth without compromising over the lucidity of the text and programs. This approach makes this book suitable for both novices and advanced programmers. The well - structured programs are easily understandable by the beginners and useful for the experienced programmers. The book contains about 300 programs, 210 exercises and 80 programming exercises with solutions of exercises and hints to solve programming exercises. Five new chapters have been included in this edition of the book. The chapter on project development and library creation can help students in implementing their knowledge and become a perfect C programmer.

ABOUT THE AUTHOR

Suresh Kumar Srivastava has been working in software industry for last 7 years. He has done B level from DOEACC Society. He has worked on development of device driver, debugger and system software area. He likes to work on system side as well as some creative work for development of software tools. He has authored a book on Data Structures titled “ Data Structures through C in Depth”. He is planning to write some more books on different topics, which can be useful for students to work in system software development.

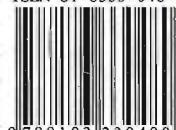
Deepali Srivastava has done MSc. In Mathematics and Advanced PGDCA from MJP Rohilkhand University. Her areas of interest are C and Data Structures. She also likes to learn and work on systems software. She has authored a book on Data structures titled “Data Structures through C in Depth”. She is planning to work on some other topics and use her creativity in system software development.

ISBN 81-8333-048-7

www.bpbonline.com



BPB PUBLICATIONS
B-14, CONNAUGHT PLACE, NEW DELHI-110001



Rs. 360/-