Patni Computer Systems Ltd.

C Programming

Version 4.1

19th May 2005

SEI-CMM LEVEL 5 Company

PACE
Patni Academy for Competency Enhancement

Table of Contents

1 An Introduction to "C"

C is a programming language developed at AT & T's Bell Laboratories of USA in 1972. It was designed and written by Dennis M. Ritchie. In the late seventies, C began to replace the more familiar languages of that time like PL/1, ALGOL etc. Possibly, C seems so popular because it is reliable, simple and easy to use.

1.1 Evolution of "C"

An international committee developed ALGOL 60 language, which is used to program all type of applications such as commercial applications, scientific applications, system applications and so on. However, ALGOL 60 never became popular because it was too abstract and too general. To reduce this abstractness and generality, a new language called Combined Programming Language (CPL) was developed at Cambridge University. However, CPL turned out to be so big, having so many features, that it was hard to learn and difficult to implement.

Basic Combined programming Language (BCPL), developed by Martin Richards at Cambridge University to solve the problems of CPL. But unfortunately it turned out to be too less powerful and too specific. Around same time a language called "B" was written by Ken Thompson at AT & T's Bell labs, as a further simplification of CPL. But like BCPL, B is also too specific. Finally Ritchie inherited the features of B and BCPL, added some of his own stuff and developed "C".

1.1.1 Compilation and Execution

As like most high-level languages, C also uses compiler to convert its source code (files with the extension .c) to object code (files with the extension .obj) and the object code will be link edited by the linker to form the machine language also known as executable code (files with the extension .exe). The following figure (Fig. 1.1) explains the various stages of compilation.

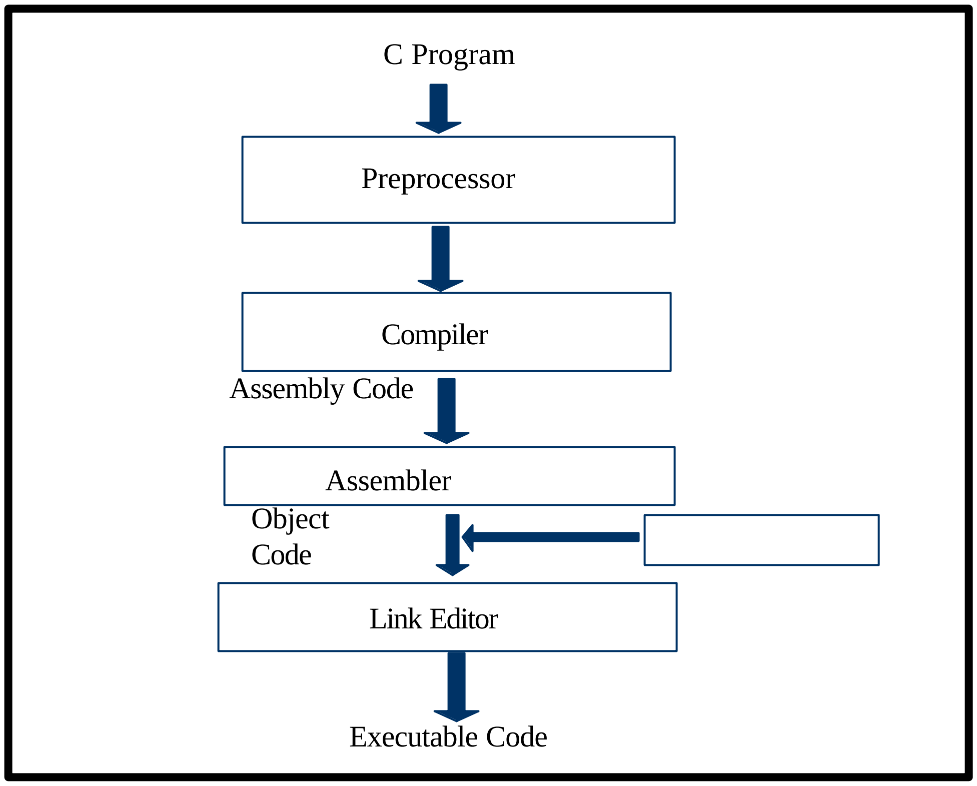


Fig 1.1: Various Stages of Compilation

1.2 Features of C

- Robust language, which can be used to write any complex program.
- Has rich set of built-in functions and operators.
- Well-suited for writing both system software and business applications.
- Efficient and faster in execution.
- Highly portable.
- Well-suited for structured programming.
- Dynamic Memory Allocation

1.3 Constants

A constant is an entity (memory location) whose value does not change during the program execution. Constants are either created literally or via the #define statement.

E.g.:

```
     58, 344          (Integer literal constants)
     'P', 'C', '7'    (Character literal constants)
     "Patni Computer Systems Ltd." (String constant)
```

A string constant is always stored in an array with multiple bytes and ends with a special character '\0' (Backslash Zero). This character, also called as **null character**, acts as a string terminator.

1.3.1 Symbolic constants

Symbolic constants are usually written in uppercase to differentiate them from variables.
E.g.:

```
    #define TRUE 1
    #define MAX_LINE 1000
    #define NULL '\0'
```

Expressions consisting entirely of constant values are called **constant expressions**.
Eg:

```
    128 + 7 – 17
```

1.4 Variables and Data Types

1.4.1 Variables

A variable is an entity used by the program to store values used in the computation. Variable names are the names (labels) given to the memory location where different constants are stored. The type of variable depends on the type of constant that it stores.

Rules for forming variable names:

- It should begin with a letter or underscore (_).
- Followed by any combination of letters, underscores or the digits 0-9.
  E.g.:

sum, piece_flag, _sys_flag.	**Valid names**
8name, price$, tel#	**Invalid names**

- The uppercase and lowercase letters are distinct in C; the variable names "Sum" and "SUM" refer to different variables.
- The length of a variable name depends on the compiler.
- No commas or blanks are allowed within a variable name.

1.4.2 Data types and sizes

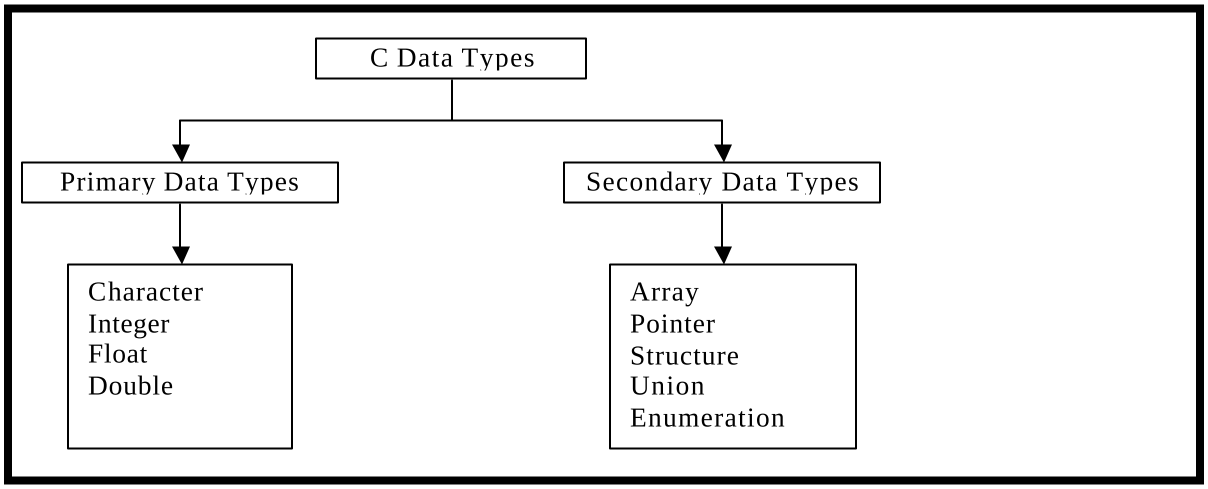


Fig 1.2: Data Types in C

1.4.2.1 Integers

The allowable range for integer (int) in a 16-bit (2 bytes) computer is -32768 to +32767. For a 32-bit (4 bytes) computer, of course, the range would be much larger. In Integer (2 bytes), the 16th bit is used to store the sign of the integer (1 - if the number is negative, 0 - if it is positive).

E.g.:

```
      int i ;
      int p = 320, r = -100;
```

There are a few qualifiers that can be applied to these basic types. short and long, which will vary the size of the variable, signed and unsigned, which varies the range.

A long integer (long int) would occupy 4 bytes of memory, which is double the size of int on a 16-bit environment. The value of long integer can vary from -2147483648 to +2147483647. short int will be same as int.

E.g.:

```
      short int i;
      long int abc;
      long xyz; /* same as long int xyz */
```

An unsigned integer is one, which cannot store negative values. The most significant bit will be utilized for storing the value and not used for storing the sign. The value will range from 0 to 65535 on a 16-bit environment. A signed int is same as int.

A long unsigned int, which has range of 0 to 4294967295, occupies 4 bytes of memory. By default, a long int is a signed long int.

E.g.:

```
     unsigned int ui;
     unsigned long ulMemAdd;
```

1.4.2.2 Floating Point or Real Numbers

Floating point numbers or Real numbers could be written in two forms, fractional form and exponential form. The value can be positive or negative. Default sign is positive. No commas or blanks are allowed. In the exponential form representation, the real constant is represented in two parts. The part appearing before ‘e’ is called mantissa, whereas the part following ‘e’ is called exponent.

The first type of floating point number is float, which is a single precision real number, occupies 4 bytes.

E.g:

```
      float p = 3.2e-5;
      float j =  4.1e98, k = 34.65F;
```

A double precision real number, double occupies 8 bytes. If situation demands usage of real numbers that lie even beyond the range offered by double data type, then there exists a long double that occupies 10 bytes.

E.g.:

```
      double d = 5.6e+34;
      long double dHigh = 3.4E-65;
```

1.4.2.3 Character

A character (char) data type stores a single alphabet, a single digit or a single special symbol enclosed within single inverted commas.

E.g:

```
      char chOld = ‘A’, chNew = ‘a’;
      char flag = ‘\n’, spec = ‘*’;
```

Character can be either signed or unsigned both occupying 1 byte each, but having different ranges. A signed char is same as ordinary char and has a range from -128 to +127, where as unsigned char has a range from 0 to 255.

1.4.2.4 String

String in “C” is a group or array of characters enclosed in double quotes. C compiler automatically puts a NULL character, ‘\0’ character, at the end of every string constant. The ‘\0’ is a string terminator. String containing no characters is a NULL string.

E.g:

char coName[] = “PCS”

P	C	S	\0

Data type	Range in environment		Usage
	16 bit	32 bit	
char	-128 to 127	-128 to 127	A single byte capable of holding one character.
short int	-2^{15} to $2^{15}-1$	-2^{31} to $2^{31}-1$	An integer, short range.
int	-2^{15} to $2^{15}-1$	-2^{31} to $2^{31}-1$	An integer.
long int	-2^{31} to $2^{31}-1$	-2^{31} to $2^{31}-1$	An integer, long range.
float	-3.4e38 to +3.4e38 (4 bytes)		Single-precision floating point.
double	-1.7e308 to +1.7e308 (8 bytes)		Double-precision floating point.
unsigned int	0 to $2^{16}-1$	0 to $2^{32}-1$	Only positive integers.
unsigned char	0 to 255	0 to 255	Only positive byte values.

Fig 1.3: Data types and their range.

Note: The size of an integer is considered as 4 bytes, in the further discussions till the scope of this book, assuming that you will be working on a 32-bit environment. If you are working on a 16-bit environment consider the size of an integer as 2 bytes.

1.4.2.5 Declarations

All the variables/constants must be declared before use. A declaration specifies a type, and contains a list of one or more variables of that type.

E.g.:

```
     int nCount, nLow, nHigh;
     char c;
```

1.5 Escape Characters

These are non-graphic characters including white spaces. These are non-printing characters and are represented by escape sequences consisting of a backslash (\) followed by a letter.

Character	Description
\b	Backspace
\n	New line
\a	Beep
\t	Tab
\”	”
\\	\
\’	’
\r	Carriage return

Table 1.1: Escape Characters.

1.6 Format Control Strings

Data Type	**Conversion Specifier**
signed char	%c
unsigned char	%c
short signed int	%d
short unsigned int	%u
long signed int	%ld
long unsigned int	%lu
float	%f
double	%lf
long double	%Lf

Table 1.2: Format Control Strings.

1.7 The Structure of a C program

- Preprocessor Directives
- Function declarations and definitions
- A function is a block of statement that breaks the entire program into smaller units.
- A C program must have a main function, which is the entry point to all the programs.
- This function can call other library functions or user-defined functions.

1.7.1 The preprocessor directive

Preprocessor is a part of the compiler. A C program may have the following preprocessor directive sections.

```
    # include <file-name>
```

The #include directive tells the preprocessor to treat the contents of a file, specified by *file-name*, as if those contents had appeared in the source program at the point where the directive appears. You can organize constant and macro definitions into include files and then use #include directives to add these definitions to any source file.

```
    # define  identifier   token-string
```

The #define directive gives a meaningful name to a constant (symbolic constant) in your program. This directive substitutes *token-string* for all subsequent occurrences of an *identifier* in the source file.

1.8 First C program

```
void main(void)
{
        char c;
        unsigned char d;
        int i;
        unsigned int j;
        long int k;
        unsigned long int m;
        float x;
        double y;
        long double z;
        scanf(“%c %c”, &c, &d);
        printf(“%c %c”, c, d);

        scanf(“%d %u”, &i, &j);
        printf(“%d %u”, i, j);

        scanf(“%ld %lu”, &k, &m);
        printf(“%ld %lu”, k, m);

        scanf(“%f %lf %lf”, &x, &y, &z);
        printf(“%f %lf %lf”, x, y, z);
}
```

Fig 1.4: First C Program

2 Operators and Type Conversion

2.1 Operators

An operator is a symbol which represents a particular operation that can be performed on some data. The data is called as operand. The operator thus operates on an operand. Operators could be classified as “unary”, “binary” or “ternary” depending on the number of operands i.e, one, two or three respectively.

2.1.1 Arithmetic operators

The binary arithmetic operators are +, -, *, / and the modulus operator %. Integer division truncates any fractional part. Modulus operator returns the remainder of the integer division. This operator is applicable only for integers and cannot be applied to float or double.

The operators *, / and % all have the same priority, which is higher than the priority of binary addition (+) and subtraction (-). In case of an expression containing the operators having the same precedence it gets evaluated from left to right. This default precedence can be overridden by using a set of parentheses. If there is more than one set of parentheses, the innermost parentheses will be performed first, followed by the operations with-in the second innermost pair and so on.

E.g.:

```
	34 + 5 = 39
	12 – 7 = 5
	15 * 5 = 75
	14 / 8 = 1
	17 % 6 = 5
```

2.1.2 Relational operators

Relational operators are used to compare two operands to check whether they are equal, unequal or one is greater than or less than the other.

Operator	Description
>	Greater than
>=	Greater than or equals to
<	Less than
<=	Less than or equals to
= =	Equality test.
!=	Non-equality test.

Table 2.1: Relational operators.

The value of the relational expression is of integer type and is 1, if the result of comparison is true and 0 if it is false.

E.g.:

```
	14 > 8     has the value 1, as it is true
	34 <= 19   has the value 0, as it is false
```

2.1.3 Logical operators

The logical operators && (AND), || (OR) allow two or more expressions to be combined to form a single expression. The expressions involving these operators are evaluated left to right, and evaluation stops as soon as the truth or the falsehood of the result is known.

Operator	Usage
&&	Logical AND. Returns 1 if both the expressions are non-zero.
||	Logical OR. Returns 1 if either of the expression is non-zero.
!	Unary negation. It converts a non-zero operand into 0 and a zero operand into 1.

Table 2.2: Logical operators.

Note: All the expressions, which are part of a compound expression, may not be evaluated, when they are connected by && or || operators.

Expr1	Expr2	Expr1 && Expr2	Expr1 || Expr2
0	0	0	0
0	non-zero	0	1
non-zero	0	0	1
non-zero	non-zero	1	1

Table 2.3: Operation of logical && and || operators

2.1.4 Unary Increment and Decrement operators

The unary increment operator (++) increments the value of the operand by 1. Similarly the unary decrement operator (--) decrements the value by 1.

E.g.:

```
int x = 0;
int p = 10;
x = p++ ;  ----------->    Result:   x = 10
// Now p will have a value 11. (Postfixing)
x = ++p;   ----------->    Result :  x = 12
// Now p will have a value 12. (Prefixing)
p = 11          ----------->    p = 11
```

Postfixing: The unary operators (increment or decrement) when used after the variable, as in p++, acts as a postfix operator. In the expression p++, p is incremented after its value has been used i.e., assigned to x.

Prefixing: The unary operators (increment or decrement) when used before the variable, as in ++p, acts as a prefix operator. The expression ++p increments p before its value has been used i.e., assigned to x.

The table below contains some more examples on unary operators.

Values before operations	Expression	Values after operations
a = 1	b = a++;	b = 1, a = 2
a = 1	b = ++a;	b = 2, a = 2
a = 1	b = a--;	b = 1, a = 0
a = 1	**b = --a;**	b = 0, a = 0
a = 1	b = 8 - ++a;	b = 6, a = 2
a = 1, c = 5	b = a++ + --c;	b = 5, a = 2, c = 4
a = 1, c = 5	b = ++a - c--	b = -3, a = 2, c = 4

Table 2.4: Examples for unary operators

2.1.5 Bitwise operators

The bitwise operators provided by C may only be applied to operands of type char, short, int and long, whether signed or unsigned.

```
      &     bitwise AND
      |     bitwise inclusive OR
      ^     bitwise exclusive OR
```

2.1.6 Ternary/Conditional Operator

The conditional expressions written with the ternary operator "?:" provides an alternate way to write the if conditional construct. This operator takes three arguments.

The syntax is:

```
      expression1 ? expression2 : expression3
```

If expression1 is true (i.e. Value is non-zero), then the value returned would be expression2 otherwise the value returned would be expression3.

Eg:

```
     int num, res;
     scanf("%d", &num);
     res = ( num >= 0 ? 1 : 0 );

res contains 1 if num is positive or zero, else it contains 0.

     int big, a, b, c;
     big = (a > b ? (a > c 3 : 4) : ( b  > c ? 6 : 8 ));

 big contains the highest of all the three numbers.
```

2.1.7 Compound Assignment operators

Most of the binary operators like +, * have a corresponding assignment operator of the form *op*= where *op* is one of +, -, *, /, %, &, |, ^. The explanation of these compound assignment operators is given below in the table 2.5.

Operator	Explanation
v + = expr	Value of the expression (expr) is added with the value of variable (v) and stores the sum in same variable (v).
v - = expr	Value of the expression (expr) is subtracted from the value of variable (v) and stores the balance in variable (v).
v * = expr	Value of the expression (expr) is multiplied with the value of variable (v) and stores the product in variable (v).
v / = expr	Value of the expression (expr) divides the value of (v) and stores the quotient in v.
v % =expr	Value of the expression (expr) divides the value of v and stores the remainder in v.
v &= expr	**Value of the expression (expr) is ANDed bitwise with the value of variable (v) and stores the result in variable (v).**
v |= expr	Value of the expression (expr) is ORed bitwise with the value of variable (v) and stores the result in variable (v).
v ^= expr	Value of the expression (expr) is XORed bitwise with the value of variable (v) and stores the result in variable (v).

Table 2.5: Explanation of Compound Assignment operators

Consider the value i = 15 for all the expressions given in the table below.

Operator	Expression	Result
i + = 3	i = i + 3	i = 18
i - = 2	i = i – 2	i = 13
i * = 4	i = i * 4	i = 60
i / = 3	i = i / 3	i = 5
i % = 4	i = i % 4	i = 3

Table 2.6: Examples for Compound Assignment operators

2.1.8 The sizeof operator

The sizeof operator returns the number of bytes the operand occupies in memory. The operand may be a variable, a constant or a data type qualifier.

```
/* sample program using sizeof operator */
    # include <stdio.h>
    void main(void)
    {
        int sum;
        printf(“%d \n”, sizeof(float));
        printf(“%d \n”, sizeof(sum));
        printf(“%d \n”, sizeof(char));
        printf(“%d \n”, sizeof(‘G’));
    }
```

Fig 2.1: Sample Code using sizeof operator

The output of the above program will be compiler dependent.

The sizeof operator is generally used to determine the lengths of entities called arrays and structures when their sizes are not known. It is also used to allocate memory dynamically during program execution.

2.2 Precedence and order of evaluation

The hierarchy of commonly used operators is shown in the table 2.7 below.

Operators	Associativity
! ++ -- + - (unary)	right to left
* / %	left to right
+ - (binary)	left to right
< <= > >=	left to right
= = !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
? :	right to left
= += -= *= /= %= &= ^= |=	right to left

Table 2.7: Precedence and Associativity of operators

In case of a tie between operations of same priority then they are evaluated based on their associativity. You can use parentheses to change the order of evaluation. If there is more than one set of parentheses, the innermost parentheses will be performed first, followed by the operations with-in the second innermost pair and so on.

C, like most languages, does not specify the order in which the operands of an operator are evaluated. Similarly, the order in which function arguments are evaluated is also not specified. So the statement

```
     printf(“%d %d\n”, ++n, power(2, n));  /* AVOID */
```

can produce different results with different compilers, depending on whether n is incremented before power is called. The solution is to write

```
     ++n;
     printf(“%d %d\n”, n, power(2, n));
```

2.3 Type conversion

When an operator has operands of different types, they are converted to a common type according to a small number of rules. In general, the only automatic conversions are those that convert a "narrower" operand into a "wider" one without losing information, such as converting an integer to a floating-point value.

2.3.1 Implicit arithmetic conversions

If a binary operator like +, -, * or / that takes two operands of different types then the "lower" type is promoted to the "higher" type before the operation proceeds. The result is of the "higher" type.

Operator1	Operator2	Result	Operator1	Operator2	Result
char	Char	char	int	float	float
char	Int	int	int	double	double
char	Float	float	long int	float	float
char	Double	double	double	float	double

Table 2.8: The conversion rules for different data types.

An arithmetic operation between an integer and integer always yields an integer result. Operation between float and float always yields a float result. Operation between float and integer always yields a float result.

Operation	Result	Operation	Result
5/2	2	2*5	10
5.0/2	2.5	2.0+5	7.0
5/2.0	2.5	5.0/2.0	2.5
5.0*2	10.0	2/5	0

Table 2.9: Arithmetic operations.

2.3.2 Type conversion in Assignments

In certain cases the type of the expression and the type of the variable on the left-hand side of assignment operator may not be same. In such a case the value of the expression promoted or demoted depending on the type of the variable on the left-hand side of = operator.

E.g.:

```
int p, iNum = 30;
float b = 3.5;
p = b;
 b = iNum;
```

In above example, the first assignment will store 3 to the variable p, because p is an integer variable, it cannot store a float value. The float is demoted to an integer and its value is stored. Exactly opposite happens in the next statement. Here, 30 is promoted to 30.000000 and then stored in b, since b is a float variable.

2.3.3 Type casting

Explicit type conversions can be forced in any expression, with a unary operator called a *cast*. In the construction

```
(type-name) expression
```

The expression is converted to the named type by the conversion rules. The precise meaning of a cast is as if the expression were assigned to a variable of the specified type, which is then used in place of the whole construction.

E.g.:

```
            int iCount;
            float fVal = 34.8f;
            iCount = (int) fVal; /* iCount contains 34 */
```

3 Control Flow

The control flow statements of a language specify the order in which computations are performed. They determine the "Flow of Control" in a program.

C programming language provides three types of control statements.

1. **Sequence Control Statements**
   The sequence control statement ensures that the instructions in the program are executed in the same order in which they appear in the program.

2. **Selection or Decision Control Statements**
   The decision and case control statements allow selective processing of a statement of a group of statements. These are also called as Conditional Statements.

3. **Repetition or Loop Control Statements**
   The Loop control statement executes a group of statements repeatedly till a condition is satisfied.

3.1 Statements and blocks

An expression becomes a statement when a semicolon follows it. Braces { and } are used to group declarations and statements together into a compound statement, or block, so that they are syntactically equivalent to a single statement. There is no semicolon after the right brace that ends a block.

3.2 Selection or Decision Control Statements

The major decision making constructs of C language are:

1. The if statement
2. The if-else statement
3. The switch statement

3.3 The if statement

The if statement is used to specify conditional execution of a program statement, or a group of statements enclosed in braces.

The general format of if statement is:

```
if  (expression)
{
            statement-block;
}
program statement;
```

Fig 3.1: Format of IF statement

When an if statement is encountered, expression is evaluated and if its value is true, then statement-block is executed, and after the execution of the block, the statement following the if statement (program statement) is executed. If the value of the expression is false, the statement-block is not executed and the execution continues from the statement immediately after the if statement (program statement).

```
/* Program to print the maximum of the two given numbers
using if statement */
void main(void)
{
    int n1, n2, max;
    printf("Enter two numbers: ");
    scanf("%d%d", &n1, &n2);
    max = n1;
    if (n2 > n1)
        max = n2;
    printf("The Maximum of two numbers is: %d \n", max);
}
```

Fig 3.2: Program to print the maximum of the two numbers using if statement

3.4 The if ..else statement

The purpose of if-else statement is to carry out logical tests and then, take one of the two possible actions depending on the outcome of the test.

The general format of if-else statement is:

```
if (expression)
{
        /* if block */
    true-statement-block;
}
else
{
        /* else block */
    false-statement-block;
}
```

Fig 3.3: Format of if..else statement

If the expression is true, then the true-statement-block, which immediately follows the if is executed otherwise, the false-statement-block is executed.

```
/* Program to check whether the given number is even or odd
*/
void main(void)
{
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);
    if ((num % 2) = = 0)
       printf("%d is even \n", num);
    else
       printf("%d is odd \n", num);
}
```

Fig 3.4: Program to check whether the given number is even or odd

The group of statements after the if upto but not including the else is known as an if block. The statements after the else form the else block. When the if block or the else block contains more than one statements they have to be enclosed in a pair of { } braces. When the if or else block contain only one statement it need not be enclosed in braces as written in the example above.

Note: Its always a good practice to enclose the if, else or any loop blocks in the braces for maintainability of the code.

3.5 Nested conditional constructs

The if statement can be included within other if block, the else block or of another conditional statement.

```
if  (expression1)
{
      true-statement1-block;
      if (expression2)
      {
          true-statement2-block;
      }
}
else
{
      false-statement1-block;
}
```

Fig 3.5: Format of Nested if statement

3.6 The else if .. statement

This sequence of if statements is the most general way of writing a multi-way decision. The expressions are evaluated in order; if any expression is true, the statement associated with it is executed, and this terminates the whole chain. can be included within the if block, the else block or of another conditional statement.

```
if  (expression1)
{
        statement-block1;
}
else if (expression2)
{
        statement-block2;
}
else
{
         default-statement-block;
}
```

Fig 3.6: Format of else..if statement

The last else part handles the "none of the above" or default case where none of the other conditions is satisfied. If there is no explicit action for the default then the else block can be omitted.

```
/* Program to calculate and print telephone bill for customers by checking certain conditions
*/
void main(void)
{
            int units, custno;
            float charge;
            printf("Enter customer no and units consumed: ");
            scanf("%d%d", &custno, &units);
            if (units <= 200)
                    charge = 0.5 * units;
            else if (units <= 400)
                    charge = 100 + 0.65 * (units - 200);
            else if (units <= 600)
                    charge = 230 + 0.8 * (units - 400);
            else
                    charge = 390 + (units - 600);
            printf("Customer No: %d consumed %d units \n", custno, units);
            printf("The total bill amount is : %.2f \n", charge);
}
```

Fig 3.7: Program to calculate and print telephone bill for customers

3.7 The switch statement

The switch statement is a multi-way decision that tests whether an expression matches one of a number of constant integer values, and branches accordingly.

```
switch (expression)
{
        case value1:
                statement-block1;
                break;
        case value2:
                statement-block2:
                break;
                …
        default:
               default-block;
}
```

Fig 3.8: Format of switch statement

3.8 Repetition or Loop control statements

These statements are also called as **Iterative Structure** or **Program Loop**. This allows a sequence of program statements to be executed several times, either a specified number of times or until a particular condition is satisfied.

It consists of an entry point that may include initialization of loop variables, a loop continuation condition, a loop body and an exit point.

C has three major loop control methods:

1. **The while loop**
2. **The do-while loop**
3. **The for loop**

The loop continuation condition may be tested before the loop body is executed as in case of while and for loops. In such case, the loop is referred to as a **pre-test loop**.

The case in which the condition is tested after the execution of the loop body, as in case of do-while loop, such a loop is called as **post-test loop**.

3.8.1 The while Loop

The general format of a while loop is:

```
        initialization;
        while (expression)
        {
              statements;
        }
```

Fig 3.9: Format of while loop

The expression is evaluated first. If the expression evaluates to non-zero (true), the body of the loop is executed. After execution of the body, the expression is once again evaluated and if it is true, the body of the loop is executed once again.

This process continues until the result of the expression becomes zero (false). The iteration is then terminated and the control passes to the first statement that follows the body of the while loop. If the expression evaluates to zero (false) at the very first time, the body of the loop is not executed even once.

```
/* Program to print numbers 1 to 10 using while loop */
void main(void)
{
    int num = 1;
    while (num <= 10)
    {
        printf(“%d \n”, num);
        num++;
    }
}
```

Fig 3.10: Program to print numbers 1 to 10 using while loop

3.8.2 The do...while loop

The general format of a do…while loop is:

```
    initialization;
    do
     {
            statement-block;
     }
     while (expression);
```

Fig 3.11: Format of do...while loop

In case of do…while loop, the body of the loop is executed, followed by the evaluation of the expression. If the expression evaluates to non-zero (true) the body of the loop is again executed.

The iteration continues until the expression evaluates to zero (false). The iteration is then terminated. If the expression evaluates to zero (false) at the very first time, the body of the loop is already executed once.

```
/* Program to print numbers 1 to 10 using do…while loop */
void main(void)
{
    int num = 1;
    do
    {
        printf("%d \n", num++);
    }
    while (num <= 10);
}
```

Fig 3.12: Program to print numbers 1 to 10 using do...while loop

Note: Since the exit condition is evaluated at the bottom of the loop, in case of do…while, the body of the loop is executed at least once.

In case of while and do…while loops, the loop counter is initialized before the control enters the loop and it must be incremented/decremented within the body of the loop.

3.8.3 The for Loop

The for loop is very flexible and is preferable when there is a simple initialization and increment, as it keeps the loop control statements close together and visible at the top of the loop.

The general format of the for loop is:

```
for (expr1; expr2; expr3)
{
    statement-block;
}
```

Fig 3.13: Format of for loop

This is equivalent to

```
expr1;
while (expr2)
{
    statement-block;
    expr3;
}
```

The three components of for loop are expressions. Most commonly, expr1 (initialization) and expr3 (increment) are assignments or function calls and expr2 (test condition) is a relational expression.

The sequence of control flow or the evaluation of these three expressions is:

1. The initialization (expr1 is evaluated) is done only once at the beginning.
2. Then the condition (expr2) is tested. If satisfied (evaluates to non-zero) the body of the loop is executed, otherwise the loop is terminated.
3. When the expr2 evaluates to non-zero the body of the loop is executed. Upon reaching the closing braces of for, control is sent back to for statement, where the increment (expr3) is performed.
4. Again the condition is tested and will follow the path based on the results of the test condition.

```
/* Program to print numbers 1 to 10 using for loop */
void main(void)
{
    int num;
    for (num = 1; num <= 10; num++)
    {
        printf(“%d \n”, num);
    }
}
```

Fig 3.14: Program to print numbers 1 to 10 using for loop

3.8.3.1 The features of the for loop

One or more variables can be initialized (expr1) at a time in for loop.

for (p = 0, q = 1; p < 10; p++)

This has two parts in its initialization separated by a comma.

Similar to initialization, the increment section (expr3) may also have more than one part.

for (m = 0, n = 25; m < n; m++, n--)

This has two parts in increment section, m++ and n --, separated by a comma.

The test condition (expr2) may have any compound relation and testing need not be limited only to loop control variable.

for (i = 1, sum = 0; i < 10 && sum < 50; i++)

This loop uses the compound test condition with loop control variable i and sum.

Any of the three parts can be omitted, although the semi colon must remain.

for (; p < 100;)

Both initialization (expr1) and increment (expr3) sections are omitted.
If the test condition (expr2), is not present, it is taken as permanently true, so

```
for ( ; ; ) {
        statement-block;
}
```

is an “infinite” loop, presumably to be broken by other means, such as a break or return.

3.9 Loop Interruption

It is sometimes convenient to be able to exit from a loop other than by testing the loop termination condition at the top or bottom.

3.9.1 The break statement

The break statement provides an early exit from for, while, and do, just as from switch. A break causes the innermost enclosing loop or switch to be exited immediately.

```
/* Program to print sum of prime numbers between 10 and 100
*/
 void main(void)
 {
      int sum = 0, i, j;
      for (i = 10; i <= 100; i++)
      {
          for (j = 2; j <= sqrt(i); j++)
              if (i % j = = 0)
                   break;

          if (j > sqrt(i))
              sum += i;
      }
      printf (“%d \n”, sum);
 }
```

Fig 3.15: Program to print sum of prime numbers between 10 and 100

The break statement breaks the inner loop as soon as the first divisor is found, but the iteration continues in the outer loop for the next value of i.

3.9.2 The continue statement

The continue statement is used to bypass the remainder of the current pass through a loop. That is, it passes the flow of control to the next iteration within for, while or do loops.

In the while and do, this means that the test part is executed immediately; in the for, control passes to the increment step. The continue statement applies only to loops, not to switch.

```
 for (i = 0; i < n; i++)
 {
     if (arr[i] < 0)
          continue;
     sum += a[i];
 }
```

The above code fragment calculates the sum of only the positive elements in the array arr; negative values are skipped.

3.9.3 The exit function

The standard library function, **exit ( )** is used to terminate execution of the program. The difference between break statement and exit function is, break just terminates the execution of loop in which it appears, whereas exit ( ) terminates the execution of the program itself.

4 Functions

A function is a self-contained block of program that performs some specific, well-defined task. A C program consists of one or more functions rather than one large main() function. printf() and scanf() are two predefined functions that we have used so far.

Functions break large complicated computing tasks into smaller and simpler ones. Separating a program into functions also aids in maintenance and enhancement of programs by localizing the effects of changes. A C program may reside in one or more source files. Source files may be compiled separately and loaded together, along with previously compiled functions from libraries. This helps programmers to build on the existing functions by creating their own functions and tying them to the existing library.

4.1 Fundamentals of functions

There are basically two types of functions.

1. **Library functions**
2. **User-defined functions**

The commonly required functions written, compiled and placed in libraries are called as "Library Functions". Some examples of library functions are printf() and scanf() etc.

The functions written by the user, are termed as "User Defined Functions". In user-defined functions, user has freedom to choose the function name, return data type and the arguments (number and type). There is no conceptual difference between the user defined and library function. The method of calling both the functions is same.

4.2 Function declaration and prototype

The function can be declared with a prototype of its parameters.

The general form of a function declaration is:

```
return-type     function-name (argument declaration);
```

Fig 4.1: Format of a function declaration

where,

return-type: The data type of the value, which is returned.

function-name: The name of the function defined.

argument declaration: types and names of the parameters of the function, separated by commas.

Thus the declaration

```
	int Cube(int);
```

declares a function Cube that returns integer value with one argument of type integer.

Function declaration is also called as function prototype, since they provide model or blueprint of the function.

4.3 Function definition

A function definition introduces a new function by declaring the type of value it returns and its parameters, and specifying the statements that are executed when the function is called.

The general format of a function definition is:

```
return-type  function-name (parameters declaration)
{
        local variable declaration;
        statements;
}
```

Fig 4.2: Format of a function definition

where,

return-type: the data type of the value, which is returned
function-name: **Name of the function defined**
parameter declaration: Types and names of the parameters of the function, separated by commas.

Functions in C are used not only to determine values, but also to group together related actions, such as displaying the headers of a report.

A function, that does not return any value, but only performs some operation, is declared with the return-type as void. Similarly if the function does not take any parameters then it is declared with parameter declaration of type void. The specification of function type is optional for some compilers. The rules for naming functions and parameters are the same as for naming variables.

Thus the function

```
double area(int n, double d)
{
        // function body
}
```

Defines area to be a function that returns a value of type double, and has two parameters – n of type integer and d, of type double.

The function body consists of variable declarations followed by any valid C-statements, enclosed within the curly braces.

User may send as many parameters to the function as he wishes, but the function itself can return one and only one value.

4.4 Function call

A function call is an expression of the form:

```
function-name (argument-list);
```

Fig 4.3: Format of a function call

where,

Function-name: Name of the function called

Argument-list : A comma separated list of expressions that constitute the arguments to the function.

Thus the statement

AddValue (nIndex);

is a function call that invokes the function named AddValue with the argument nIndex.

```
/* Example of function usage */
    # include <stdio.h>
    main ( )
    {
         void sub (void); /* Function prototype */
         printf ("In main function, before function call.\n");
         sub ( );  /* Function call */
         printf ("In main function, after function call. \n");
    }
    void sub ( )
    {
         printf("Welcome to the function sub \n");
    }
```

Fig 4.4 : Example of function usage

The main( ) function gets executed first. As the control encounters the statement sub( );, the activity of main( ) is temporarily suspended and control passes to the sub( ). After execution of sub( ), the control again returns to main( ). main( ) resumes its execution from the statement after sub( ).

Thus main( ) becomes the "calling function" as it calls the function sub( ) and sub( ) becomes the "called function" as it is called in main( ).

If a program contains multiple functions, their definitions may appear in any order, though they must be independent of one another. That is, one function definition cannot be embedded within another.

There is no limit on the number of functions that might be present in a C program. Each of the function is called in the sequence specified by the function calls in the main( ).

4.5 The return statement

In the figure 4.4, the moment closing brace of the called function (sub) was encountered, the control returned to the calling function (main). No separate return statement was necessary to send back the control as the called function is not going to return any value to the calling function.

However, in functions, which are expected to return some values, it is necessary to use the return statement.
Syntax:

```
return (expression);
        or
return;
```

Fig 4.5: Syntax of return statement

On executing the return statement,
The value of the expression, which is just after the return keyword, is returned to the calling function.
Control is transferred back to the calling function.
If the expression is not present, it returns an integer or void depending on the compiler that you use.

The expression can be a constant, a variable, a user defined data structure, a general expression or a function call.

If the data type of the expression returned does not match the return type of the function, it is converted to the return type of the function.

For example, in the function

```
int convert()
{
    return 10.32;
}
```

Fig 4.6: Sample code for return statement

the return statement is equivalent to
    return (int 10.32) ;
and returns 10 to the calling function.

If you do not have a return statement in the function, the calling function will receive the control, but no value. Such a type of function is known as a **void function**.

More than one return statement can be used in the same function as shown below.

```
int factorial(int n)
{
    int i,result;
    if(n<0)
         return -1;
     if(n==0)
         return 1;
    for(i=1,result=1;i<=n;i++)
         result *=i;
    return result;
}
```

Fig 4.7: Sample code using more than one return statement

The first executed return statement terminates the execution of the function and the rest of the function body is not executed. Thus, if factorial is called with arguments 0, the function will return with the value 1 and for loop will not be executed.

4.6 Function arguments

The function parameters are the means of communication between the calling and the called functions. There is no limitation on the number of parameters passed to a function.

Formal parameters:

These, commonly called as parameters, are given in the function declaration and function definition.

Actual parameters:

These, commonly called as **arguments**, are specified in the function call.

The following conditions must be satisfied for a function call:

1. The list of arguments in the function call and function declaration must be the same.
2. The data type of each of the actual parameter must be same as that of formal parameter.
3. The order of the actual parameters must be same as the order in which the formal parameters are specified.

However, the names of the formal parameters in function declaration and definition are unrelated. They can be same or different.

```
#include <stdio.h>
void main(void)
{
    int calcsum(int, int, int);
     int a, b, c, sum;
    printf("Enter three numbers");
    scanf("%d%d%d", &a, &b, &c);
    sum = calcsum(a, b, c);
    printf("The sum is : %d", sum);
}
int calcsum(int x, int y, int z)
{
    int d;
    d = x + y + z;
    return (d);   ---> Integer value of d is returned
}
```

Fig 4.8: Sample code for function arguments

In this program, from the function main() the values of a, b and c are passed on to the function calcsum(), by making a call to the function calcsum() and passing a, b, and c in the parentheses:

```
     sum = calcsum(a, b, c);
```

In the calcsum() function these values get collected in three variables x, y, z.

```
     calcsum(int x, int y, int z);
```

The variables a, b and c are called ‘ actual parameters’, whereas the variables x, y and z are called ‘formal parameters’. Any number of arguments can be passed to a function being called. However, the type, order and number of the actual and formal arguments must always be the same.

4.6.1 Passing Arguments to a Function

C provides following two mechanisms to pass arguments to a function:

1. **Pass arguments by value (Call by value)**
2. **Pass arguments by address or by pointers (Call by reference)**

4.6.1.1 Call By Value

Functions in C pass all arguments by value. It means the contents of the arguments in the calling functions are not changed, even if they are changed in the called function. The contents of the variable are copied to the formal parameters of the function definition, thus preserving the contents of the argument in the calling function.

The following example illustrates the concept of passing arguments by value.

```
/* Program to demonstrate pass by value */
#include<stdio.h>
void main(void)
{
   int num = 100;
   void modify(int);
   printf("In main, the value of num is %d \n", num);
   modify(num);
   printf("Back in main, the value of num is %d \n", num);
}
void modify(int n)
{
   printf("In function value of num is %d \n", n);
   n = 200;
   printf("In function changed value of num is %d \n", n);
}
```

Output

```
In main, the value of num is 100
In function value of num is 100
In function changed value of num is 200
Back in main, the value of num is 100
```

Fig 4.9: Sample code for passing arguments by value

The variable num is assigned a value of 100 in the function main(). During execution of the function, the value of the variable n is changed to 200, but the value of the variable num in the function main remains the same as prior to the execution of the function call i.e., 100.

4.6.1.2 Call By Reference

Instead of passing the value of a variable, we can pass the memory address of the variable to the function. It is termed as Call by Reference. We will discuss call by reference when we learn pointers.

4.7 Scope Of Variables

The part of the program within which a variable/constant can be accessed is called as its scope.

By default the scope of a variable is local to the function in which it is defined. Local variables can only be accessed in the function in which they are defined; they are unknown to other functions in the same program.

Setting up variables that are available across function boundaries can change the default scope of variable. If a variable is defined outside any function at the same level as function definitions is called as **External variable**.

Scope of an external variable is the rest of the source file starting from its definition.

Scope of an external variable defined before any function definition, will be the whole program, and hence such variables are sometimes referred to as Global variables.

Following code uses external variables:

```
int i, j;
void input()
{
    scanf(“%d %d”, &i, &j);
}
int k;
void compute()
{
    k = power(i, j);
}
void output()
{
    printf(“i=%d j=%d k=%d”, i, j, k);
}
```

Fig 4.10: Sample code using external variables

4.8 Storage Classes

All variables have a data type; they also have a ‘Storage class’. The storage class determines the lifetime of the storage associated with the variable. If we don’t specify the storage class of a variable in its declaration, the compiler will assume a storage class depending on the context in which the variable is used.

From C compiler’s point of view, a variable name identifies some physical location within the computer where the string of bits representing the variables’ value is stored. Basically, there are two types of locations in a computer where such a value is kept. They are “Memory” and “CPU Registers“.

It is variable’s storage class, which determines in which of these two locations the value is stored.

A variable’s storage class gives the following information:

- Where the variable would be stored.
- What will be the default initial value
- What is the scope of the variable
- What is the life of the variable, i.e. how long would the variable exist.

There are four types of storage classes in C:

1. **Automatic Storage Class**
2. **Static Storage Class**
3. **Register Storage Class**
4. **External Storage Class**

4.8.1 Automatic Variables

A variable is said to be automatic, if it is allocated storage upon entry to a segment of code, and the storage is reallocated upon exit from this segment.

Features of a variable with an automatic storage class are as follows:

Storage	**Memory**
Default initial value	Garbage value
Scope	Local to the block, in which it is defined
Life	Till the control remains within the block, in which it is defined.

A variable is specified to be automatic by prefixing its type declaration with the storage class specifier – auto - in the following manner

```
    auto data-type variable-name;
```

By default, any variable declared in a function is of the `automatic storage class`. They are automatically initialized at run-time.

Thus the declarations of the variables *i* and *result* in

```
int num(int n)
{
        int i, result;
}
```

is equivalent to

```
int num(int n)
{
        auto int i, result;
}
```

and declare *i* and *result* to be automatic variables of type integer.

An automatic variable may be initialized at the time of its declaration by following its name with an equal sign and an expression. The expression is evaluated and its value is assigned to the automatic variable each time the block is entered.

Thus, the *auto* variable *result*, when initialized as

```
int num(int n)
{
        auto int i, result=1;
 }
```

will be set to 1 each time num is called. The function parameters can also be used in the initialization expression.

Thus, the *auto* variable *last* when initialized as

```
int num(int n)
{
        auto int i, result=n-1;
 }
```

is set to one less than the value of the actual argument supplied with a call to *num.*

Note: In the absence of explicit initialization, the initial value of an automatic variable is undefined.

4.8.2 Static Variables

A variable is said to be static, if it is allocated storage at the beginning of the program execution and the storage remains allocated until the program execution terminates. Variables declared outside all blocks at the same level as function definitions are always static.

Features of a variable with a static storage class are as follows

Storage	**Memory**
Default initial value	Zero
Scope	Local to the block, in which it is defined
Life	Value of the variable persists between different function calls.

Within a block, a variable can be specified to be static by prefixing its type declaration with the storage class specifier static in the following manner

```
static data-type variable-name;
```

Thus the declarations of the variable *i* in

```
int num(void)
{
        static int i;
}
```

declares i as a static variable of type integer.

Variables declared as static could be initialized only with constant expressions. The initialization takes place only once, when the block is entered for the first time.

The following program illustrates the difference between auto and static variables.

```
#include <stdio.h>
void main(void)
{
  void incr(void);
  int i;
  for(i = 0; i < 3; i++)
    incr();
}
void incr()
{
    int auto_i = 0;
    static int static_i = 0;
     printf(“auto=%d \t static=%d\n“, auto_i++,static_i++);
}
```

Fig 4.11: Sample code for function arguments

```
Output
         auto=0    static=0
         auto=0    static=1
         auto=0    static=2
```

The output shows the value of auto_i is 0 for each line of display, and that of static_i incremented by 1 from 0 through 2.

While auto_i is assigned the value 0, each time the function incr() is called, static_i is assigned the value 0 only once, when incr() is first executed and its value is retained from one function call to the next.

4.8.3 Register Variables

In case when faster computation is required, variables can be placed in the CPU’s internal registers, as accessing internal registers take much less time than accessing memory. Therefore, if a variable is used at many places in a program it is better to declare its storage class as register.

Features of a variable with a register storage class are as follows

Storage	CPU registers
Default initial value	Garbage value
Scope	Local to the block, in which it is defined
Life	Till the control remains within the block in which it is defined.

A variable can be specified to be in register by prefixing its type declaration with the storage class specifier register in the following manner

```
    register data-type variable-name;
```

But this is entirely upto the compiler to decide whether a variable is to be stored as a register variable or should be automatic.

4.8.4 External Variables

If the declared variable is needed in another file, or in the same file but at a point earlier than that at which it has been defined, it must be declared of storage class external.

Features of a variable with an external storage class are as follows

Storage	**Memory**
Default initial value	Zero
Scope	Global
Life	As long as the program's execution doesn't come to an end.

A variable has to be declared with the keyword extern before it can be used.

An extern variable declaration is of the form

```
    extern type-identifier;
```

The declaration of an external variable declares, for the rest of the source file, the type of the variable but does not allocate any storage for the variable.

The definition of an external variable, specified without the keyword extern, causes the storage to be allocated, and also serves as the declaration for the rest of that source file.

An initial variable can be initialized only at the time of its definition. There must be only one definition of an external variable; all other files that need access to this variable must contain an extern declaration for this variable.

All function names are considered global and are visible in any part of the program, be it the file in which the function has been defined or any other file that is part of the source for the program.

Thus a file need not contain extern declarations for functions external to it.

The following example shows the definition, declaration and use of external variables. The program consists of two modules *main.c* and *compute.c*.

main.c	compute.c

#include <stdio.h>
int add(void);
 	
/* Declaration of i & k */
int i, k;
 	
int main(void)
{
printf("Enter values for i and k");
scanf("%d %d", &i, &k);
printf("i=%d", add());
return 0;
}
	#include <stdio.h>
	#define MODULUS 10
	/* Extern declaration of i & k: No new variables are created */
	extern int i, k;
	/* Declaration and definition of j */
	int j = MODULUS;
	
	int add()
	{
	i += j + k;
	return(i);
	}

Fig 4.12: Sample code for the usage of external variables

The declarations common to more than one module are usually collected in a single file, known as the header file. These are then copied into the modules that use them by means of the #include directive. By convention, the names of the header files are suffixed with .h.

For instance, the preceding program can be rewritten by collecting constants and external declarations in a file named global.h as follows:

```
/*********global.h***************/
# include <stdio.h>
# define MODULUS 10
extern int i,k;
int j=MODULUS
```

main.c

```
#include <stdio.h>
int add(void);
int i,k;
int main(void)
{
        printf("Enter values for i and k"):
        scanf("%d%d",&i,&k);
        printf("i=%d",add());
        return 0;
}
```

compute.c

```
#include “global.h”
int add(void)
{
        i += j + k;
        return(i);
 }
```

User-defined header files are to be included by enclosing it within double quotes along with full path or otherwise it would be searched in the current directory

For a very large program, there may be more than one header file; each module then includes only those header files that contain information relevant to it.

Each module of a large program is separately compiled. Separate compilation speeds up debugging, as the whole program does not have to be recompiled if the changes are confined to one module.

4.9 Variable initialization

In the absence of explicit initialization, external and static variables are guaranteed to be initialized to zero; automatic and register variables have undefined (i.e., garbage) initial values.

For external and static variables, the initializer must be a constant expression; the initialization is done once, conceptually before the program begins execution. For automatic and register variables, it is done each time the function or block is entered. For automatic and register variables, the initializer is not restricted to being a constant: it may be any expression involving previously defined values, even function calls.

4.9.1 Scope rules

The functions and external variables that make up a C program need not all be compiled at the same time; the source text of the program may be kept in several files, and previously compiled routines may be loaded from libraries.

The scope of an identifier is the part of the program within which the identifier can be used. For an automatic variable declared at the beginning of a function, the scope is the function in which the identifier is declared. Local variables of the same name in different functions are unrelated. The same is true of the parameters of the functions, which are in effect local variables.

The scope of an external variable or a function lasts from the point at which it is declared to the end of the file. On the other hand, if an external variable is to be referred to before it is defined, or if it is defined in a different source file from the one where it is being used, then an extern declaration is mandatory.

4.10 Recursion

Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied. The process is used for repetitive computations in which each action is stated in terms of a previous result. Many iterative problems can be written in this form.

Functions may be defined recursively; that is, a function may directly or indirectly call itself in the course of execution, If the call to a function occurs inside the function itself, the recursion is said to be direct. If a function calls another function, which in turn makes a call to the first one, the recursion is said to be indirect. The chain of calls may be more involved; there may be several intermediate calls before the original function is called back.

```
/* To calculate factorial of an integer using recursion */
/* factorial of n is calculated as n! = n * (n-1)! */

#include <stdio.h>
long int factorial(int n)
{
    if((n == 0)||(n == 1))
         return 1;
    else
         return(n * factorial(n-1));
}
void main(void)
{
    int num;
    printf(“Enter a number : “);
    scanf(“%d”, &num);
    if(num >= 0)
         printf(“\n Factorial(%d)=%4d \n”, num,
                 factorial(num));
    else
         printf(“\n Invalid Input \n”);
}
```

Fig 4.12: Program To calculate factorial of an integer using recursion

5 Arrays

C language provides a capability called 'array' that enables the user to design a set of similar data types. Very often, one needs to process collections of related data items, such as addition of fifty numbers, test scores of students in a university, a set of measurements resulting from an experiment, income tax tables, etc. One way of handling such a situation would be to declare a new variable name for each of these data items. This approach obviously is quite cumbersome, if not altogether impossible.

A better way to solve the problem is to use an array of a corresponding data type. This enables the user to access any number of relative data type using a single name and subscript.

5.1 Definition

An ordered finite collection of data items, each of the same type, is called an array, and the individual data items are its elements.

Only one name is assigned to an array and specifying a subscript references individual elements.

A subscript is also called an index. In C, subscripts start at 0, rather than 1, and cannot be negative. The single group name and the subscript are associated by enclosing the subscript in square brackets to the right of the name.

Consider an example where marks of some students are stored in an array named mark, then mark[0] refers to the marks of first student, mark[1] to the marks of second student, mark[10] to the marks of eleventh student and mark[n-1] to the marks of nth student.

An array has the following properties:

- The type of an array is the data type of its elements.
- The location of an array is the location of its first element.
- The length of an array is the number of data elements in the array.
- The storage required for an array is the length of the array times the size of an element.

Arrays, whose elements are specified by one subscript, are called one-dimensional arrays. These are commonly known as **Vectors**.

Arrays, whose elements are specified by more than one subscript, are called multi-dimensional arrays. These are commonly known as **Matrix**.

5.2 Declaration of Single Dimensional Array (Vectors)

Arrays, like simple variables, need to be declared before use.

An array declaration is of the form:

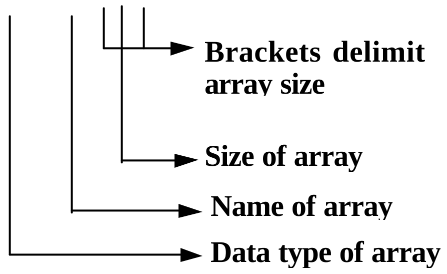
```
[storage class] data-type arrayname[size] ;
```

where,

storage class	Storage class of an array.
data-type	The type of data stored in the array.
arrayname	Name of the array.
Size	Maximum number of elements that the array can hold.

Hence, an array num of 50 integer elements can be declared as:

int num[50];



5.3 Initialization of Single Dimensional Array

Elements of an array can be assigned initial values by following the array definition with a list of initializers enclosed in braces and separated by commas.

For example, The declaration:

```
     int mark[5] = {40,97,91,88,100};
```

declares an array mark to contain five integer elements and initializes the elements of array as given below:

mark[0]	**40**
mark[1]	**97**
mark[2]	**91**
mark[3]	**88**
mark[4]	**100**

The declaration:

```
     char name[3] = {‘R’,’A’,’J’};
```

declares an array name to contain three character elements and initializes the elements of array as given below:

name[0]	**‘R’**
name[1]	**‘A’**
name[2]	**‘J’**

The declaration:

```
     float price[7] = {0.25, 15.5, 10.7, 26.8, 8.8, 2.8, 9.7};
```

declares an array price to contain seven float elements and initializes the elements of array as given below:

price[0]	**0.25**
price[1]	**15.5**
price[2]	**10.7**
price[3]	**26.8**
price[4]	**8.8**
price[5]	**2.8**
price[6]	**9.7**

Since any constant integral expression may be used to specify the number of elements in an array, symbolic constants or expressions involving symbolic constants may also appear in array declarations.

For example, The declaration:

```
     #define    UNIT_PRICE 80
     #define    TOT_PRICE       100
     int    sl_price[UNIT_PRICE] ;
     int nt_price[TOT_PRICE] ;
```

declare sl_price and nt_price to be one-dimensional integer array of 80 and 100 elements respectively.

The array size may be omitted during declaration.

Thus, the declaration,

```
    int mark[] = {40,97,91,88,100};
```

is equivalent to the

```
    int mark[5] = {40,97,91,88,100};
```

In such cases, the subscript is assumed to be equal to the number of elements in the array (5 in this case).

The elements, which are not explicitly initialized, are automatically set to zero.

E.g.:

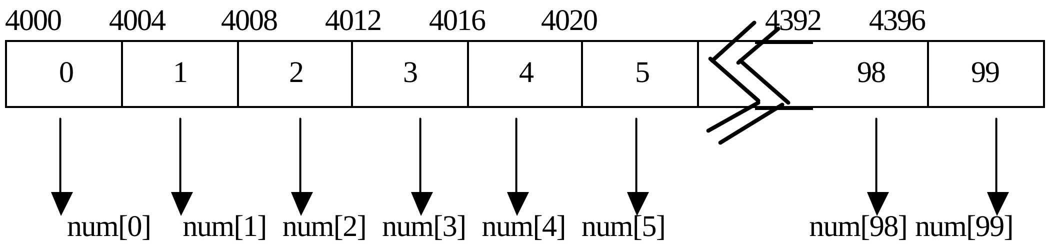
```
int x[4]={1,2}; implies
      x[0]=1
      x[1]=2
      x[2]=0
      x[3]=0
```

5.4 Array elements in memory

Consider the following array declaration:

```
   int num[100];
```

In the above declaration, 400 bytes get immediately reserved in memory, as each of the 100 integers would be of 4 bytes long. An array is a set of contiguous memory locations, first element starting at index zero. The allocation will be like this.



As seen above, array elements are always numbered (index) from 0 to (n-1) where n is the size of the array.

5.5 Array Processing

The capability to represent a collection of related data items by a single array enables the development of concise and efficient programs.

An individual array element can be used in a similar manner that a simple variable is used. That is user can assign a value, display it's value or perform arithmetic operations on it.

To access a particular element in an array, specify the array name, followed by square braces enclosing an integer, which is called the `Array Index`.

For example, The assignment statement

```
     num[5] = 2 ;
```

assigns 2 to 6th element of num.

```
     p = (net[1] + amount[9]) /2 ;
```

assigns the average value of **2nd element of net** and **10th element of amount** to p.

The statement

```
     --num[8] ;
```

decrements the content of **9th element of num** by 1**.**

The assignment statements

```
     i = 5;
     p = num[++i] ;
```

assigns the value of **num[6]** to p.

whereas the statements

```
     i = 5 ;
     p = num[i++] ;
```

assign the value of **num[5]** to p.

However, all operations involving entire arrays must be performed on an **element-by-element** basis. This is done using loops. The number of loop iterations will hence equal to the number of array elements to be processed.

As an illustration of the use of arrays, consider the following program.

```
/* Program to find average marks obtained by 25 students in a test by accepting marks of
each student */
        # include <stdio.h>
        void main(void)
        {
                  int i;
                  float sum=0;
                  float mark[25];
                  for(i=0;i<25;i++)
                  {
                            printf("Enter marks :  ");
                            scanf("%f",&mark[i]);
                            sum += mark[i];
                  }
                  printf("\n Average marks : %.2f \n",sum/25);
        }
```

Fig 5.1: Sample code using Arrays

5.6 Multidimensional Arrays

Multidimensional Arrays are defined in much the same manner as single dimensional arrays, except that a separate pair of square brackets is required for each subscript (dimension).

5.6.1 Declaration of multi-dimensional arrays

Declaration of multi-dimensional arrays:

Syntax:

[storage class] data-type arrayname[expr-1][expr-2] ... [expr-n];

Where,

storage class	Storage class of an array.
data-type	The type of data stored in the array.
arrayname	Name of the array.
expr-1	A constant integral expression specifying the number of elements in the 1st dimension of the array.
expr-n	A constant integral expression specifying the number of elements in the nth dimension of the array.

The scope of this course will limit the discussion to 2-dimensional arrays only.

Thus a two-dimensional array will require two pairs of square brackets. One subscript denotes the row and the other the column. All subscripts i.e. row and column start with 0.

So, a **two-dimensional array** can be declared as

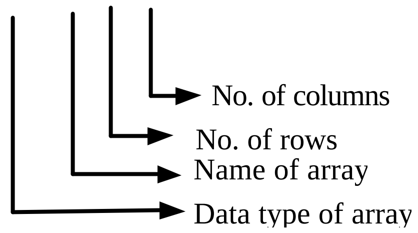
```
   [storage class] data-type arrayname[expr1][expr2];
```

where,

expr1	Maximum number of rows that the array can hold.
expr2	Maximum number of columns that the array can hold.

Hence, an array num of integer type holding 5 rows and 10 columns can be declared as

int num[5][10];



5.6.2 Initialization of two-dimensional arrays

Two-dimensional arrays are initialized analogously, with initializers listed by rows. A pair of braces is used to separate the list of initializers for one row from the next, and commas are placed after each pair of brace except for the last row that closes off a row.

E.g.

```
     int no[3][4] = {
                     {1,2,3,4},
                     {5,6,7,8},
                     {9,10,11,12}
                    };
```

declares an array no of integer type to contain 3 rows and 4 columns. The inner pairs of braces are optional. Thus the above declaration can equivalently be written as

```
     int no[3][4] = {1,2,3,4,5,6,7,8,9,10,11,12};
```

Above array initializes the elements of array as given below:

no[0, 0] = 1	**no[0, 1] = 2**	**no[0, 2] = 3**	**no[0, 3] = 4**
no[1, 0] = 5	**no[1, 1] = 6**	**no[1, 2] = 4**	**no[1, 3] = 5**
no[2, 0] = 9	**no[2, 1] = 10**	**no[2, 2] = 11**	**no[2, 3] = 12**

Note: It is important to remember that while initializing a two-dimensional array it is necessary to mention the second (column) dimension, whereas the first dimension (row) is optional.

Hence the declaration,

```
     int arr[2][3] = {12,34,56,78};        valid
     int arr[ ][3] = {12,34,56,78};        valid
    int arr[2][ ] = {12,34,56,78};         invalid
    int arr[ ][ ] = {12,34,56,78};         invalid
```

5.6.3 Memory Representation of Two-dimensional Arrays

A two-dimensional array a[i][j] can be visualized as a table or a matrix of i rows and j columns as shown below:

	col 1	**col 2**	**col 3**		**col j-1**	**col j**
row 1	**a[0][0]**	**a[0][1]**	**a[0][2]**	**----**	**a[0][j-2]**	**a[0][j-1]**
row 2	**a[1][0]**	**a[1][1]**	**a[1][2]**	**----**	**a[1][j-2]**	**a[1][j-1]**
row i-1	**a[i-2][0]**	**a[i-2][1]**	**a[i-2][2]**	**----**	**a[i-2][j-2]**	**a[i-2][j-1]**
row i	**a[i-1][0]**	**a[[i-1][1]**	**a[i-1][2]**	**----**	**a[i-1][j-2]**	**a[i-1][j-1]**

All the elements in a row are placed in contiguous memory locations.

Consider the statement

```
     char OS[2][4] = {“DOS”,”ABC”};
```

Internally in memory, it is represented as:

D	**O**	**S**	**‘\0’**	**A**	**B**	**C**	**‘\0’**
OS [0, 0]	**OS [0, 1]**	**OS [0, 2]**	**OS [0, 3]**	**OS [1, 0]**	**OS [1, 1]**	**OS [1, 2]**	**OS [1, 3]**

5.6.4 Two-Dimensional Array Processing

Processing of two-dimensional array is same as that of single dimensional arrays.
As an illustration of the use of two-dimensional arrays, consider the following program.

```
/* Program to find average marks obtained by a class of 25 students in a test by accepting
roll number and marks of each student */
# include <stdio.h>
void main(void)
{
        int i;
        float sum=0;
        int student[25][2];
        for(i=0; i<25; i++)
        {
                printf("Enter Roll no and marks : ");
                scanf("%d%d", &student[i][0], &students[i][1]);
                 /* Roll no will get stored in students[i][0] and marks in students[i][1] */
                sum += students[i][1];
         }
        printf("\n Average marks : %.2f\n", sum/25);
}
```

Fig 5.2: Sample code for Two dimensional Array processing

5.7 What are strings?

A string constant is one-dimensional array of characters terminated by a null ('\0') character. Strings are used to store text information and to perform manipulations on them. Strings are declared in the same manner as other arrays.

For Example

```
    char fruit[10];
```

5.7.1 Initializing Character Arrays

Character arrays can be initialized in two ways as individual characters or as a single string.

```
    char name[ ] = {'P', 'a', 't', 'n', 'i', '\0'};
```

Each character in the array occupies one byte of memory and the last character is always '\0', which is a single character. The null character acts as a string terminator. Hence a string of n elements can hold (n-1) characters.

```
    char fruit[ ] = "Apple";
```

Note that, in this declaration '\0' is not necessary, C inserts the null character automatically, when the array is initialized with a double quoted string constant.

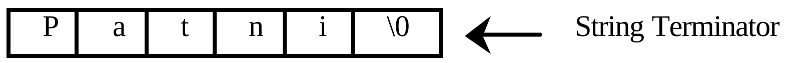
When initializing a character array, the length may be omitted. The compiler automatically allocates the storage depending on the length of the value given.

E.g.:

```
      char name[ ] = "Patni";
```

The above declaration automatically assigns storage equivalent to 6 characters including '\0' to the character array name.

Memory representation of above array is shown in figure below:



```
/* Program to accept and print a string */
void main(void)
{
        char name[20];
        scanf("%s", name);
        printf("%s", name);
}
```

The %s used in printf() is a format specification for printing out a string. The same specification can be used with scanf() also. In both cases we are supplying the base address to the functions. The scanf() function , after the enter is pressed automatically inserts a '\0' at the end of the string. The scanf() function is not capable of receiving multi-word strings separated by space. In that case use the gets() and puts() functions.

```
/* Program that accepts and prints a string using gets and puts functions */
#include <stdio.h>
#include <string.h>
main()
{
      char name[20];
      gets(name);
      puts(name);
}
```

Following are some examples given using strings.

```
/* Program to compute the length of a given string */
#include <stdio.h>
void main(void)
{
        char str[10];
        int len;
        printf("\n Enter string :");
        scanf("%[^\n]", arr1);
        for(len = 0; str[len] != '\0'; len++);
        printf("\nThe length of the string is %d\n", len);
}
```

5.8 Built-in String Functions

The header file `string.h` provides useful set of string functions. These functions help in manipulating strings. To use these functions, the header file `string.h` must be included in the program with the statement:

```
    # include <string.h>
```

5.8.1 strcat (target, source)

The strcat() function accepts two strings as parameters and concatenates them, i.e. it appends the source string at the end of the target.

```
/* Sample program using strcat() */
#include <stdio.h>
#include <string.h>
void main(void)
{
        char name1[]= "Ash";
        char name2[]= "wini";
        strcat(name1, name2);
        printf("\n");
        puts(name1);
}
```

Output:

```
        Ashwini
```

5.8.2 strcmp (string1, string2)

The function strcmp() is used to compare two strings. This function is useful while writing program for ordering or searching strings.

The function accepts two strings as parameters and returns an integer value, depending upon the relative order of the two strings.

Return value	**Description**
Less than 0	If string1 is less than string2
Equal to 0	If string1 and string2 are identical
Greater than 0	If string1 is greater than string2

Table 5.1: strcmp() function return values

```
/* Sample program to test equality of two strings  using strcmp() */
# include <stdio.h>
# include <string.h>
void main(void)
{
        char str1[10];
        char str2[10];
        int result;
        printf("\n*** Comparing two strings ***\n");
        fflush(stdin); /* flush the input buffer */
        printf("Enter first string\n");
        scanf("%s", str1);
        fflush(stdin);
        printf("\nEnter second string\n");
        scanf("%s", str2);
        result = strcmp(str1, str2);
        if(result < 0)
                printf("\nString2 is greater than String1 ...");
        else if(result == 0)
                printf("\nBoth the Strings are equal..");
        else
                printf("\nString1 is greater than String2 ...");
}
```

The function strcmp() compares the two strings, character by character, to decide the greater one. Whenever two characters in the string differ, the string that has the character with a higher ASCII value is greater.

E.g. consider the strings *hello* and *Hello!*

The first character itself differs. The ASCII code for h is 104, while that for H is 72. Since the ASCII code of h is greater, the string hello is greater than Hello!. Once a difference is found, there is no need to compare the other characters of the strings; hence, function returns the result.

5.8.3 strcpy(target, source)

The strcpy() function copies one string to another. This function accepts two strings as parameters and copies the source string character by character into the target string, up to and including the null character of the source string.

```
/* Sample program using strcpy() function */
# include <stdio.h>
# include <string.h>
void main(void)
{
        char name1[]= "Ash";
        char name2[]= "win";
        printf("\n** Before Copying two strings are **\v");
        printf("%s\t%s", name1, name2);
        strcpy(name1, name2);
        printf("\n** After Copying two strings are **\v");
        printf("%s\t%s\n", name1, name2);
}
Output
        ** Before Copying two strings are **
                     Ash             win
        ** After Copying two strings are **
                     win             win
```

5.8.4 strlen(string)

The strlen() function returns an integer value, which corresponds, to the length of the string passed. The length of a string is the number of characters present in it, excluding the terminating null character.

```
/* Sample Program using strlen() function() */
# include <stdio.h>
# include <string.h>
void main(void)
{
        char arr1[10];
        int i, len;
        printf("\nEnter string :\n");
        scanf("%[^\n]", arr1);
        printf("\nThe length of the string is %d", strlen(arr1));
}
```

5.9 Two Dimensional Arrays of Characters

```
main()
{
        char namelist[3][10] ={
                            "akshay",
                            "parag",
                            "raman"
                            };
}
```

Instead of initializing the names, had these names been supplied from the keyboard, the program segment would have looked like this...

```
        for (i = 0; i < 3; i++)
            scanf("%s", namelist[i]);
```

The memory representation of the above array is given below

1001	a	k	s	h	a	y	\0	▨	▨	▨
1011	p	a	r	a	g	\0	▨	▨	▨	▨
1021	r	a	m	a	n	\0	▨	▨	▨	▨

Even though 10 bytes are reserved for storing the name 'akshay', it occupies only 7 bytes. Thus 3 bytes go waste.

5.10 Standard Library String Functions

Function	**Description**
strlen	Finds the length of a string
strlwr	Converts a string to lowercase
strupr	Converts a string to uppercase
strcat	Appends one string at the end of another
strncat	Append first n character of a string at the end of another
strcpy	Copies a string into another
strncpy	Copies first n character of one string into another
strcmp	Compares two strings
strncmp	compares first n characters of two strings

Table 5.2: String built-in functions.

6 Pointers

The significance of pointers in C is the flexibility it offers in the programming. Pointers enable us to achieve parameter passing by reference, deal concisely and effectively either arrays, represent complex data structures, and work with dynamically allocated memory.

Although, a lot of programming can be done without the use of pointers, their usage enhances the capability of the language to manipulate data. Pointers are also used for accessing array elements, passing arrays and strings to functions, creating data structures such as linked lists, trees, graphs, and so on.

6.1 What is a pointer variable?

Memory can be visualized as an ordered sequence of consecutively numbered storage locations. A data item is stored in memory in one or more adjacent storage locations depending upon its type. The address of a data item is the address of its first storage location. This address can be stored in another data item and manipulated in a program. The address of a data item is called a **pointer** to the data item and a variable that holds an address is called a **pointer variable**.

Uses of Pointers

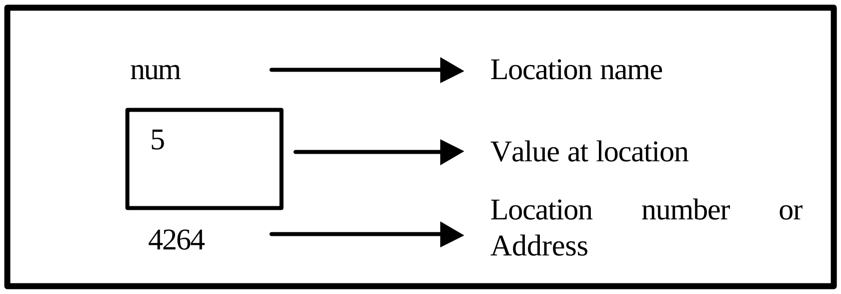
1. Keep track of address of memory locations.
2. By changing the address in pointer type variable you can manipulate data in different memory locations.
3. Allocation of memory can be done dynamically.

6.2 Address and Dereferencing (& and *) Operators

Consider the declaration

```
int num = 5;
```

The compiler will automatically assign memory for this data item. The data item can be accessed if we know the location (i.e., the *address*) of the first memory cell.



The address of **num**'s memory location can be determined by the expression **&num**, where **&** is unary operator, called the **'address of'** operator. It evaluates the address of its operand.

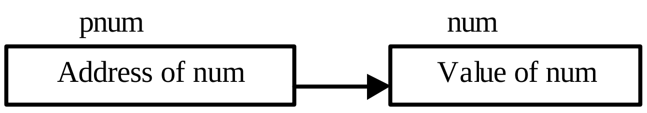
We can assign the address of **num** to another variable, *pnum* as:

```
pnum = &num;
```

This new variable *pnum* is called **a pointer to num**, since it **points** to the location where *num* is stored in memory. Thus *pnum* is referred to as a **pointer variable**.

The data item represented by *num*, can be accessed by the expression **pnum*, where ***** is unary operator, called **'the value at the address'** operator. It operates only on a pointer variable.

It can be illustrated as below:



Relationship between pnum and num (where *pnum = &num* and num = **pnum*).

Therefore, **pnum* and num both represent the same data item.

Accessing a data item through a pointer is called `Dereferencing`, and the operator asterisk (`*`) is called the '`dereferencing or indirection operator`'.

6.3 Pointer type Declaration

Pointers are also variables and hence, must be defined in a program like any other variable. The rules for declaring pointer variable names are the same as ordinary variables.

The declaration of a pointer is of the following form

```
type *varibale_name;
```

where,

type	Data type of the variable pointed by the pointer variable.
variable_name	Name of the pointer variable
***(asterisk)**	Signifies to the compiler that this variable has to be considered a pointer to the data type indicated by *type*.

For example,

int *int_ptr	*int_ptr* is a pointer to data of type integer
char *ch_ptr	*ch_ptr* is a pointer to data of type character
double *db_ptr	*db_ptr* is a pointer to data of type double

Note: All the pointer variables will occupy 4 bytes of memory regardless of the type they point to.

6.4 Pointer Assignment

The address of **(&)** operator, when used as a prefix to the variable name, gives the address of that variable.

Thus,

```
      ptr = &i;
```

assigns address of variable *i* to ptr.

```
/* Example of '&' - address of operator */
#include <stdio.h>
void main(void)
{
        int a=100;
        int b=200;
        int c=300;
        printf("Address:%u contains value :%d\n", &a, a);
        printf("Address:%u contains value :%d\n", &b, b);
        printf("Address:%u contains value :%d\n", &c, c);
}
Output:
                Address:65524 contains value :100
                Address:65520 contains value :200
                Address:65516 contains value :300
```

Fig. 6.1: Sample Code for '&' operator

A pointer value may be assigned to another pointer of the same type.

For example, in the program below

```
      int i=1, j, *ip;
      ip=&i;
      j=*ip;
      *ip=0;
```

The first assignment assigns the address of variable *i* to *ip*.

The second assigns the value at address *ip*, that is, *1* to *j*, and finally to the third assigns *0* to *i* since **ip* is the same as *i*.

The two statements

```
ip=&i;
j=*ip;
```

are equivalent to the single assignment

```
j=*(&i);
```

or to the assignment

```
j=i;
```

i.e., the address of operator **&** is the inverse of the dereferencing operator *****.

Consider the following segment of code

```
#include <stdio.h>
void main(void)
{
        char *ch;
        char b = 'A';
        ch = &b;        /* assign address of b to ch */
        printf("%c", *ch);
}
Output:                    A
```

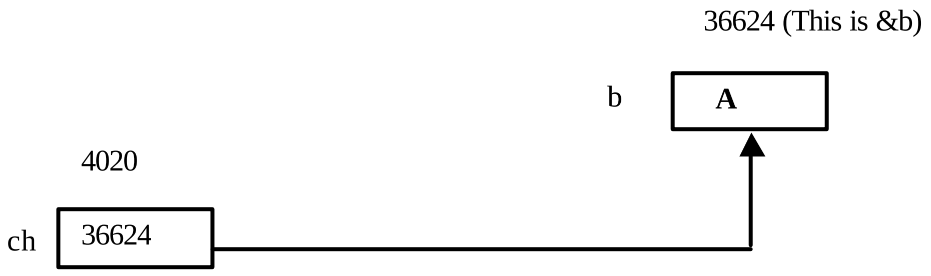


Fig. 6.2 Memory representation of pointer

In the above example,

b	value of b, which is 'A'
&b	address of b, i.e., 36624
ch	value of ch, which is 36624
&ch	address of ch, i.e., 4020 (arbitrary)
*ch	contents of ch, => value at 36624, i.e., A This is same as *(&b)

6.5 Pointer Initialization

The declaration of a pointer variable may be accompanied by an initializer. The form of an initialization of a pointer variable is

```
        type *identifier=initializer;
```

The initializer must either evaluate to an address of previously defined data of appropriate type or it can be NULL pointer.

For example, the declaration

```
        float *fp=null;
```

The declarations

```
      short s;
      short *sp;
      sp=&s;
```

initialize *sp* to the address of *s*.

The declarations

```
      char c[10];
      char *cp=&c[4];
```

initialize *cp* to the address of the fifth element of the array *c*.

```
        char *cfp=&c[0];
```

initialize cf*p* to the address of the first element of the array *c*. It can also be written as

```
        char *cfp=c;
```

Address of first ele ment of an array is also called as **base address** of array.

Following program illustrates declaration, initialization, assignment and dereferencing of pointers.

```
/* Example : Usage of Pointers */
# include <stdio.h>
void main(void)
{
        int i, j=1;
        int *jp1, *jp2=&j;      /* jp2 points to j */
        jp1 = jp2;              /* jp1 also points to j */
        i = *jp1;               /* i gets the value of j */
        *jp2 = *jp1 + i;        /* i is added to j */
        printf("i=%d j=%d *jp1=%d *jp2=%d\n", i, j, *jp1, *jp2);
}
Output:
              i=1 j=2 *jp1=2 *jp2=2
```

6.6 Pointer Arithmetic

Arithmetic can be performed on pointers. However, in pointer arithmetic, a pointer is a valid operand only for the addition(+) and subtraction(-) operators.

An integral value n may be added to or subtracted from a pointer ptr. Assuming that the data item that ptr points to lies within an array of such data items. The result is a pointer to the data item that lays n data items after or before the one p points to respectively.

The value of ptr±n is the storage location ptr±n*sizeof(*ptr), where sizeof is an operator that yields the size in bytes of its operand.

Consider following example

```
/* Example of Pointer arithmetic */
#include <stdio.h>
void main(void)
{
        int i=3, *x;
        float j=1.5, *y;
        char k='C', *z;
        printf("Value of i=%d\n", i);
        printf("Value of j=%f\n", j);
        printf("Value of k=%c\n", k);
        x=&i;
        y=&j;
        z=&k;
        printf("Original Value in x=%u\n", x);
        printf("Original Value in y=%u\n", y);
        printf("Original Value in z=%u\n", z);
        x++;
        y++;
        z++;
        printf("New Value in x=%u\n", x);
        printf("New Value in y=%u\n", y);
        printf("New Value in z=%u\n", z);
}
Output:
         Value of i=3
         Value of j=1.500000
         Value of k=C
         Original Value in x=1002
         Original Value in y=2004
         Original Value in z=5006
        New Value in x=1006
        New Value in y=2008
        New Value in z=5007
```

In the above example, New value in x is 1002(original value)+4, New value in y is 2004(original value)+4, New value in z is 5006(original value)+1.

This happens because every time a pointer is incremented it points to the immediately next location of its type. That is why, when the integer pointer x is incremented, it points to an address four locations after the current location, since an int is always 4 bytes long. Similarly, y points to an address 4 locations after the current locations and z points 1 location after the current location.

Some valid pointer arithmetics are

- Addition of a number to a pointer.
- Subtraction of a number from a pointer.

For example, if p1 and p2 are properly declared pointers, then the following statements are valid.

```
        y=*p1**p2; /*same as (*p1)*(*p2) */
        sum=sum+*p1;
        z=5*-*p2/*p1; /* same as (5*(-(*p2)))/(*p1) */
        *p2=*p1+10;
```

C allows subtracting one pointer from another. The resulting value indicates the number of bytes separating the corresponding array elements. This is illustrated in the following example:

```
 # include <stdio.h>
 void main(void)
 {
        static int ar[]={10, 20, 30, 40, 50};
        int *i, *j;
        i=&ar[1];   /* assign address of second element to i */
        j=&ar[3];   /* assign address of fourth element to j */
        printf("%d %d", j-i, *j-*i);
}
 Output:
              2       20
```

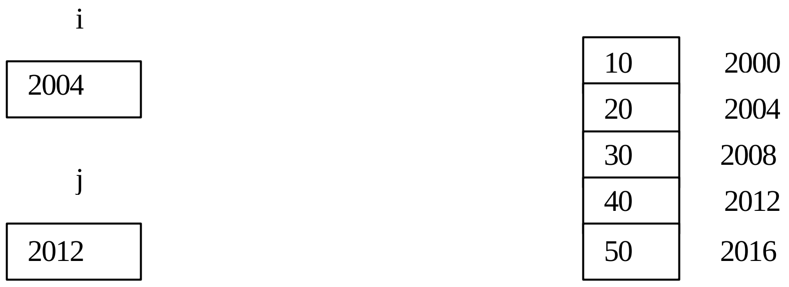


Fig 6.3: Memory Representation of Pointer Arithmetic

The result of expression (j-i) is not 8 as expected (2012-2004) but 2.

This is because when a pointer is decremented (or incremented) it is done so by the length of the data type it points to, called the scale factor

```
        (j-i) = (2012-2004) /4 = 2
```

as size of int is 4.

This is called **reference by address.**

Some invalid pointer arithmetics are

- Addition two pointers.
- Multiplication of a number with a pointer.
- Division of a pointer with a number.

6.7 Pointer Comparison

The relational comparisons ==, != are permitted between pointers of the same type.

The relational comparisons <, <=, >, >= are permitted between pointers of the same type and the result depends on the relative location of the two data items pointed to.

For example,

```
        int a[10], *ap;
```

the expression

```
        ap==&a[9];
```

is true if ap is pointing to the last element of the array a, and the expression

```
        ap<&a[10];
```

is true as long as ap is pointing to one of the elements of a.

6.8 Pointers and Functions

A function can take a pointer to any data type, as argument and can return a pointer to any data type.

For example, the function definition

```
double *maxp(double *xp, double *yp)
{
  return *xp >= *yp ? x;
}
```

specifies that the function *maxp*() return a pointer to a double variable, and expects two arguments, both of which are pointers to double variables. The function de-references the two argument pointers to get the values of the corresponding variables, and returns the pointer to the variable that has the larger of the two values. Thus given that,

```
double u=1, v=2, *mp;
```

the statement

```
mp = maxp(&u, &v);
```

makes *mp* point to *v*.

6.8.1 Call by Value

In a call by value, values of the arguments are used to initialize parameters of the called function, but the addresses of the arguments are not provided to the called function. Therefore, any change in the value of a parameter in the called function is not reflected in the variable supplied as argument in the calling function.

```
/* Example: Function parameters passed by Value */
#include <stdio.h>
void main(void)
{
        int a=5, b=7;
        void swap(int, int);
        printf("Before function call: a=%d b=%d", a, b);
        swap(a, b); /* Variables a and b are passed by value */
        printf("After function call: a=%d b=%d", a, b);
}
void swap(int x, int y)
{
        int temp;
        temp=x;
        x=y;
        y=temp;
}
Output :
        Before function call: a=5 b=7
        After function call: a=5 b=7
```

6.8.2 Call by Reference

In contrast, in a call by reference, addresses of the variables are supplied to the called function and changes to the parameter values in the called function cause changes in the values of the variable in the calling function.

Call by reference can be implemented by passing pointers to the variables as arguments to the function. These pointers can then be used by the called function to access the argument variables and change them.

```
/* Example : Arguments as pointers */
#include <stdio.h>
void main(void)
{
        int a=5, b=7;
        void swap(int*, int*);
        printf("Before function call: a=%d b=%d", a, b);
        swap(&a, &b); /* Address of variable a and b is passed */
        printf("After function call: a=%d b=%d", a, b);
}
void swap(int *x, int *y)
{
        int temp;
        /* The contents of memory location are changed */
        temp=*x;
        *x=*y;
        *y=temp;
}
Output :
          Before function call: a=5 b=7
          After function call: a=7 b=5
```

Steps involved for using pointers in a function are

1. Pass address of the variable (Using the ampersand (&) or direct pointer variables).
2. Declare the variable as pointers within the routine.
3. Refer to the values contained in a memory location via asterisk (*).

Using call by reference, we can make a function return more than one value at a time, as shown in the program below:

/* Returning more than one values from a function through arguments */

```
# include <stdio.h>
void main(void)
{
    float radius;
    float area, peri;
    void areaperi(float, float*, float*);
    printf("Enter radius : ");
    scanf("%f", &radius);
    areaperi(radius, &area, &peri);
    printf("\nArea = %.2f \n", area);
    printf("Perimeter = %.2f", peri);
}
void areaperi(float r, float *a, float *p)
{
    *a = 3.14 * r * r;
    *p = 2 * 3.14 * r;
}
```

Output :

```
Enter radius of a circle : 5
Area=78.50
Perimeter=31.40
```

6.9 Pointers to Functions

Functions have addresses just like data items. A pointer to a function can be defined as the address of the code executed when the function is called. A function's address is the starting address of the machine language code of the function stored in the memory.

Pointers to functions are used in

- writing memory resident programs
- writing viruses, or vaccines to remove the viruses.

Address of a Function

The address of a function can be obtained by only specifying the name of the function without the trailing parentheses.
For example, if CalcArea() is a function already defined, then CalcArea is the address of the function CalcArea().

Declaration of a Pointer to a Function

The declaration of a pointer to a function requires the function’s return type and the function’s argument list to be specified along with the pointer variable.

The general syntax for declaring a pointer to a function is as follows:

return-type (*pointer variable)(function’s argument list);

Thus, the declaration

int (*fp)(int i, int j);

declares *fp* to be a variable of type “pointer to a function that takes two integer arguments and return an integer as its value.” The identifiers *i* and *j* are written for descriptive purposes only.

The preceding declaration can, therefore also be written as

int (*fp)(int, int);

Thus, declarations

int i(void);	declares *i* to be a function with no parameters that return an int.
int* pi(void);	declares *pi* to be a function with no parameters that returns a pointer to an int.
int (*ip)(void);	declares *ip* to be a pointer to a function that returns an integer value and takes no arguments.

```
/* Example: Pointer to Function */
#include <stdio.h>
int func1(int i)
{
     return(i);
}
float func2(float f)
{
      return(f);
}
void main(void)
{
     int (*p)(int); /* declaring pointer to function */
     float (*q)(float);
     int i=5;
     float f = 1.5;
     p=func1; /* assigning address of function func1 to p */
     q=func2; /* assigning address of function func2 to q */
     printf("i = %d f= %f\n", p(i), q(f));
}
```

After declaring the function prototypes and two pointers p and q to the functions; p is assigned the address of function func1 and q is assigned the address of function func2.

Invoking a Function by using Pointers

In the pointer declaration to functions, the pointer variable along with the operator (*) plays the role of the function name. Hence, while invoking function by using pointers, the function name is replaced by the pointer variable.

/* Example: Invoking function using pointers */

```
# include <stdio.h>
void main(void)
{
    unsigned int fact(int);
    unsigned int ft, (*ptr)(int);
    int n;
    ptr=fact; /* assigning address of fact() to ptr */
    printf("Enter integer whose factorial is to be found:");
    scanf("%d", &n);
    ft=ptr(n); /* call to function fact using pointer ptr */
    printf("Factorial of %d is %u \n", n, ft);
}
unsigned int fact(int m)
{
    unsigned int i, ans;
    if (m == 0)
         return(1);
    else
    {    for(i=m, ans=1; i>1 ; ans *= i--);
         return(ans);
     }
}
```

pointer to function
with prototype of fact()

Output:

```
Enter integer whose factorial is to be found: 8
Factorial of 8 is 40320
```

6.9.1 Functions returning Pointers

We have already learnt that a function can return an int, a double or any other data type. Similarly it can return a pointer. However, to make a function return a pointer it has to be explicitly mentioned in the calling function as well as in the function declaration.

While retaining pointers, return the pointer to global variables or static or dynamically allocated address. Do not return any addresses of local variables because stop to exit after the function call.

/* Example: Function returning pointers */

```
/* Program to accept two numbers and find greater number */

# include <stdio.h>
void main(void)
{
    int a, b, *c;
    int* check(int, int);
    printf("Enter two numbers : ");
    scanf("%d%d", &a, &b);
    c=check(&a, &b);
    printf("\n Greater numbers : %d", *c);
}

int* check(int *p, int *q)
{
    if(*p >= *q)
         return(p);
    else
         return(q);
}
```

check function takes two integers as arguments and returns a pointer to an integer

- **The address of integers being passed to check() are collected in p and q.**
- **Then in the next statement the conditional operators test the value of *p and *q and return either the address stored in p or the address stored in q.**
- **This address gets collected in c in main().**

6.10 Pointers and Arrays

In C, there is a close correspondence between arrays and pointers that results not only in notational convenience but also in code that uses less memory and runs faster. Any operation that can be achieved by array subscripting can also be done with pointers.

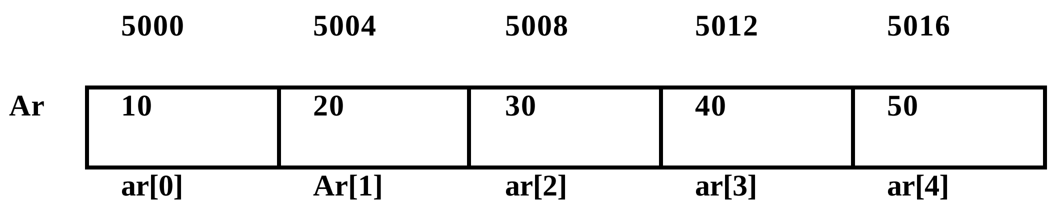
6.10.1 Pointer to Array

Arrays are internally stored as pointers. A pointer can efficiently access the elements of an array.

```
/* Program to access array elements using pointers */
#include <stdio.h>
void main(void)
{
static int ar[5]={10, 20, 30, 40, 50};
int i, *ptr;
ptr = &ar[0]; /* same as ptr = ar */
  for(i=0; i<5; i++)
  {
  printf("%d-%d\n", ptr, *ptr);
        ptr++;
  }
}
Output:
                      5000-10
                      5004-20
                      5008-30
                      5012-40
                      5016-50
```

increments the pointer to point to the next element and not to the next memory location

An integer pointer, *ptr* is explicitly declared and assigned the starting address. The memory representation of above declared array *ar* (assuming an integer takes 4 bytes of storage) is shown below:



Recall that an array name is really a pointer to the first element in that array. Therefore, address of the first array element can be expressed as either **&ar[0]** or simply **ar**.

i.e. ar=&ar[0]=5000

Hence,

***ar=*(&ar[0])**

i.e. *ar=ar[0] or *(ar+0)=ar[0]

To make the above statement more general, we can write

***(ar+i)=ar[i];**

Where, i=0,1,2,3,...

Hence any array element can be accessed using pointer notation, and vice versa.

It will be clear from following table:

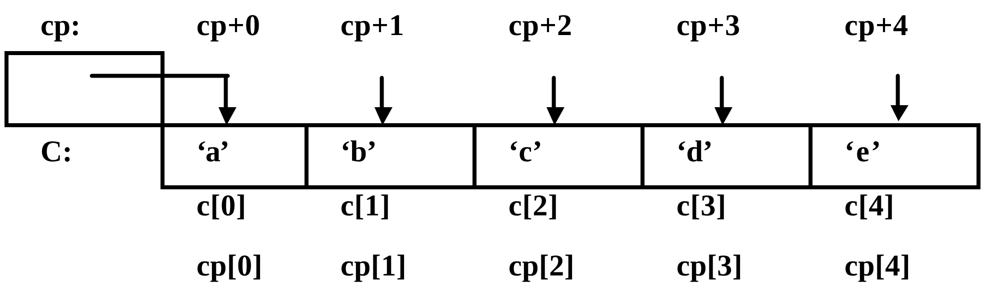
char c[10], int i;	
Array Notation	**Pointer Notation**
&c[0]	**c**
c[i]	***(c+i)**
&c[i]	**c+i**

For example, given that

```
char c[5] =  {‘a’, ’b’, ’c’, ’d’, ’e’} ;
char *cp ;
```

and

```
cp = c;
```



and

c[0]	**‘a’**	***cp**	**cp[0]**
c[1]	**‘b’**	***(cp+1)**	**cp[1]**
c[2]	**‘c’**	***(cp+2)**	**cp[2]**
c[3]	**‘d’**	***(cp+3)**	**cp[3]**
c[4]	**‘e’**	***(cp+4)**	**cp[4]**

Using this concept, we can write the above program as shown below.

```
# include <stdio.h>
void main(void)
{
     static int ar[5]={10,20,30,40,50};
     int i;
     for(i=0; i<5; i++)
          printf(“%d-%d\n”, (ar+i), *(ar+i));
}
```

Note: C does not allow to assign an address to an array.

For example,

```
ar=&a; is invalid.
```

The main difference between an array and a pointer is that an array name is a constant, (a constant pointer to be more specific), whereas a pointer is a variable.

6.10.2 Arrays as Function Arguments

An array name can be passed as an argument to a function. A formal parameter declared to be of type “array of T” is treated as if it were declared to be of type “pointer to T”.

Thus, the declaration,

```
void fun_arr(double x[], int length);
```

can equivalently be written as

```
void fun_arr(double *x, int length);
```

When an array is passed as an argument to a function, we actually pass a pointer to the zeroth element of the array i.e. a[0]. Since arrays are stored in contiguous memory locations, we can perform indexing on the starting location of the array.

Following is an example of a function that finds the value of the largest element in an integer array.

```
/* Function to return the largest number of an array */
int max(int *a, int length)
{
        int i, maxv;
        for(i=1, maxv=*a; i<length; i++)
        {
                if(*(a+i)>maxv)
                        maxv = *(a+i);
        }
        return maxv;
}
```

```
/* Program to display array elements by passing array to a function */
#include <stdio.h>
void main(void)
{
          static int num[5]={25,60,74,50,39};
          void display(int*, int);
          display(num, 5);         /* base address of array is passed */
}
void display(int *j, int n)
{
           int i=1;
           printf("Array elements are :\v");
           while(i<=n)
           {
                      printf("%d\t", *j);
                      i++;
                      j++;   /*increment pointer to point to the next location */
            }
}
Output:
                              Array elements are :
                                        25       60       74       50       39
```

6.10.3 Pointers and character arrays

All string manipulators use pointers. When a string is created, it is stored contiguously and a NULL (' \ 0') character is automatically appended to it at the end. This Null signifies the end of the string.

A character-type pointer variable can be assigned to an entire string as a part of the variable declaration. Thus, a string can conveniently be represented by either a one-dimensional character array or by a character pointer.
Shown below is simple C program in which two strings are represented as one-dimensional character arrays.

```
#include <stdio.h>
char x[]="This string is declared externally \n\n";
void main(void)
{
          char y[]="This string is declared within main";
          printf("%s", x);
          printf("%s", y);
}
```

The first string is assigned to the external array *x[]*.
The second string is assigned to the array *y[]*.

```
/* Here is a different version of the same program.  The strings are now assigned to
pointer variables rather than to conventional one-dimensional arrays. */
#include <stdio.h>
char *x = “This string is declared externally\n”;
void main(void)
{
        char *y = “This string is declared within main”;
        printf(“%s”, x);
        printf(“%s”, y);
}
Output :
          This string is declared externally
          This string is declared within main
```

The external pointer variable *x* points to the beginning of the first string, whereas the pointer variable *y*, declared within main, points to the beginning of the second string.

6.10.4 Pointers and multidimensional arrays

A two-dimensional array is actually a one-dimensional array, whose elements are themselves arrays.

A two-dimensional array declaration can be written as

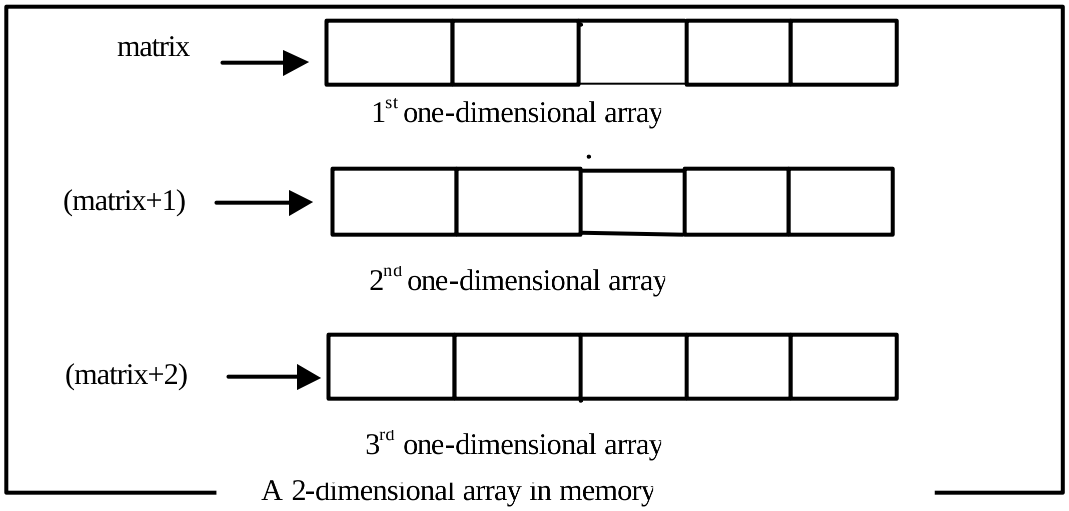
```
     data-type  array[expression 1] [expression 2];
```

In this declaration, *data-type* refers to the data type of the array, *array* is the corresponding array name, and *expression1* and *expression2* are positive integer expressions. The first subscript refers to rows and the second subscript refers to columns.

For example, the declaration

```
     int matrix[3][5];
```

specifies that the array matrix consists of three elements, each of which is an array of five integer elements, and that the name matrix is a pointer to the first row of the matrix.


matrix
1st one-dimensional array
(matrix+1)
2nd one-dimensional array
(matrix+2)
3rd one-dimensional array

A 2-dimensional array in memory

Since a one-dimensional array can be represented in terms of a pointer (the array name), it is reasonable to expect that a multidimensional array can also be represented with an equivalent pointer notation.

For example, the element matrix[i][j] can be referenced using the pointer expression

```
      *(*(matrix+i)+j)
```

since

```
             matrix  - Pointer to the first row.
           matrix+i  - Pointer to the ith row.
        *(matrix+i)  - Pointer to the first element of the ith row.
      *(matrix+i)+j  - Pointer to the jth element of the ith row.
   *(*(matrix+i)+j)  - matrix[i][j]; the jth element of the ith row.
```

The same can be represented in the following figure.

For example, the element in third row and in fifth column can be expressed by

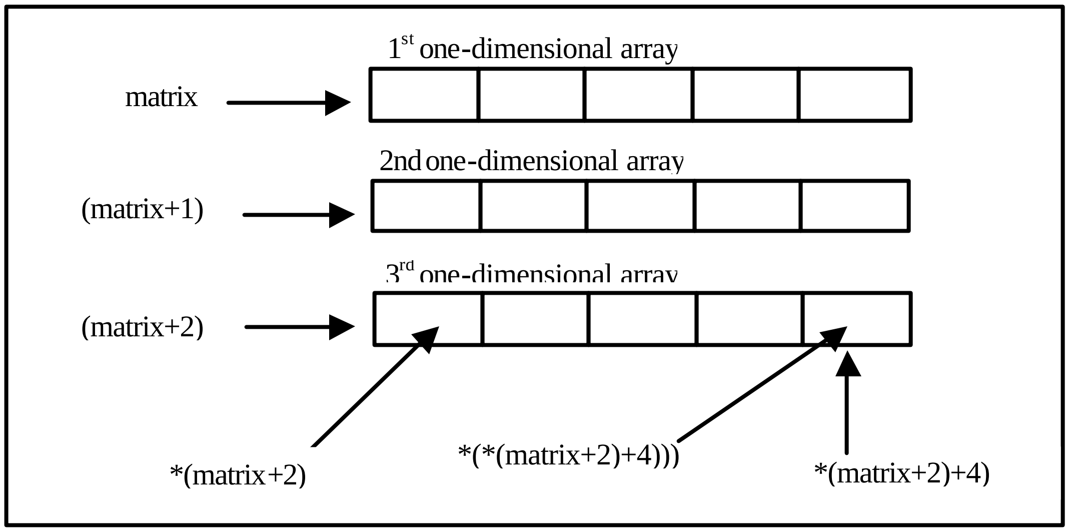

1st one-dimensional array
matrix
2nd one-dimensional array
(matrix+1)
3rd one-dimensional array
(matrix+2)
*(matrix+2)
((matrix+2)+4)))
*(matrix+2)+4)

Fig 6.4: Accessing elements of a table using indirection operator *

Assume the array matrix[3][5] is populated with the values below:

Matrix	**0**	**1**	**2**	**3**	**4**
0	**10**	**20**	**12**	**15**	**22**
1	**24**	**14**	**25**	**66**	**45**
2	**20**	**28**	**13**	**11**	**23**

Fig 6.5: Matrix populated with values

The following code is written to illustrate how a multi-dimensional array can be processed using pointers by writing a function column_total that calculates the sum of elements in a given column of the above matrix declaration.

Since pointer arithmetic works with pointers for any data type, a two-dimensional array can be traversed by initializing the pointer to the first row of the array and then incrementing the pointer to get the next row.

Let *rowptr* be the pointer to the rows of matrix. Now, the pointer *rowptr* is declared and initialized to the first row of the matrix as

```
int (*rowptr)[5] = matrix;
```

The above declaration specifies that *rowptr* is a pointer to an array of 5 integers.

Note that the parentheses around *rowptr* are necessary because the dereferencing operator * has low precedence than the indexing operator [].

Having declared *rowptr* to be a pointer to a row of matrix,

```
(*rowptr)[j]
```

refers to (j+1)th element of this row.

The function column_total is as follows:

```
int column_total(int (*matrix)[5],int rows,int col)
{
     int (*rowptr)[5]=matrix;
     int i, sum;
     for(i=0,sum=0;i<rows;i++)
     {
          sum += (*rowptr)[col];
          rowptr++;
     }
     return sum;
}
```

Note that the parameter declaration

```
    int (*matrix)[5]
```

specifies that *matrix* is a pointer to an array of 5 integer elements. This declaration is equivalent to

```
    int matrix[][5]
```

The function call

```
    column_total(matrix,3,2)
```

produces 50 as the sum of third column for the above given matrix (Refer Fig. 6.x)

Now, the function row_total is written to find the sum of the particular row for the above given matrix. As discussed earlier,

```
    *(matrix+i)
```

is the pointer to the first element of the row i of matrix. Thus, if colptr points to elements of matrix in row i, it can be initialized to point to the first element of row i by the following declaration.

```
int (*colptr) = *(matrix+i);
```

The function row_total is as follows:

```
    int row_total(int (*matrix)[5],int columns, int row)
    {
         int (*colptr)=*(matrix+row);
         int j, sum;
         for(j=0,sum=0;j<columns;j++)
              sum += *colptr++;
         return sum;
    }
```

With the same above given matrix (Fig 6.x), the function call

```
    row_total(matrix,5,2)
```

produces 95 as the sum of the third row.

6.10.5 Arrays of Pointers

As we have already seen, an array is an ordered collection of data items, each of the same type, and type of an array is the type of its data items. When the data items are of pointer type, is it known as a pointer array or an array of pointers.

Since a pointer variable always contains an address, an array of pointers is collection of addresses. The addresses present in the array of pointers can be address of isolated variables or addresses of array elements or any other addresses.

For example, the declaration

```
        char *day[7];
```

defines day to be an array consisting of seven character pointers.

The elements of a pointer array, can be assigned values by following the array definition with a list of comma-separated initializers enclosed in braces.

For example, in the declaration

```
char *days[7] = {“Sunday”, “Monday”, “Tuesday”, “Wednesday”, “Thursday”, “Friday”,
“Saturday”}
```

The necessary storage is allocated for the individual strings, and pointers to them are stored in array elements.

```
/* Example: Array of Pointers */
#include <stdio.h>
void main(void)
{
        int m = 25, n = 50, x = 60, y = 74;
        int i, ar[4] = {&m, &n, &x, &y}
        for(i=0; i<4; i++)
                printf(“%d\t”, *(ar+i));
}
```

ar contains addresses of isolated integer variables *m*, n, x and y. The for loop in the program picks up the addresses present in ar and prints the values present at these addresses.

Memory representation is shown below:

int variables	m	n	x	y
values stored in variables	25	50	60	74
addresses of variables	4002	5013	3056	9860

array of pointers	ar[0]	ar[1]	ar[2]	ar[3]
Elements of an array of pointers	4002	5013	3056	9860

An array of pointers can contain addresses of other arrays.

Multidimensional Array as Array of Pointers

A multidimensional array can be expressed in terms of an array of pointers rather than as a pointer to a group of contiguous arrays.

In general terms, a two-dimensional array can be defined as a one-dimensional array of pointers by writing –

```
data-type     *array[expression 1];
```

rather than the conventional array definition

```
data-type array[expression 1][expression 2];
```

Notice that the array name and its preceding asterisk are not enclosed in parentheses in this type of declaration. Thus, a right-to-left rule first associates the pairs of square brackets with array, defining the named data item as an array. the preceding asterisk then establishes that the array will contain pointers.

Moreover, note that the last (the rightmost) expression is omitted when defining an array of pointers, whereas the first (the leftmost) expression is omitted when defining a pointer to a group of arrays.

Example:

Suppose that *x* is a two-dimensional char array having 10 rows and 20 columns. We can define *x* as a one-dimensional array of pointers by writing -

```
char *x[10];
```

Hence, x[0] points to the beginning of the first row, x[1] points to the beginning of the second row, and so on. Note that the number of elements within each row is not explicitly specified.

In individual array element, such as x [2] [5], can be accessed by writing

```
*(x[2]+5);
```

In this expression, x[2] is a pointer to the first element in row 3, so that (x[2]+5) points to index 5 (actually, the sixth element) within row 3. The data item of this pointer, *(x[2]+5), therefore refers to x[2][5]. These relationships are illustrated in figure 6.4.

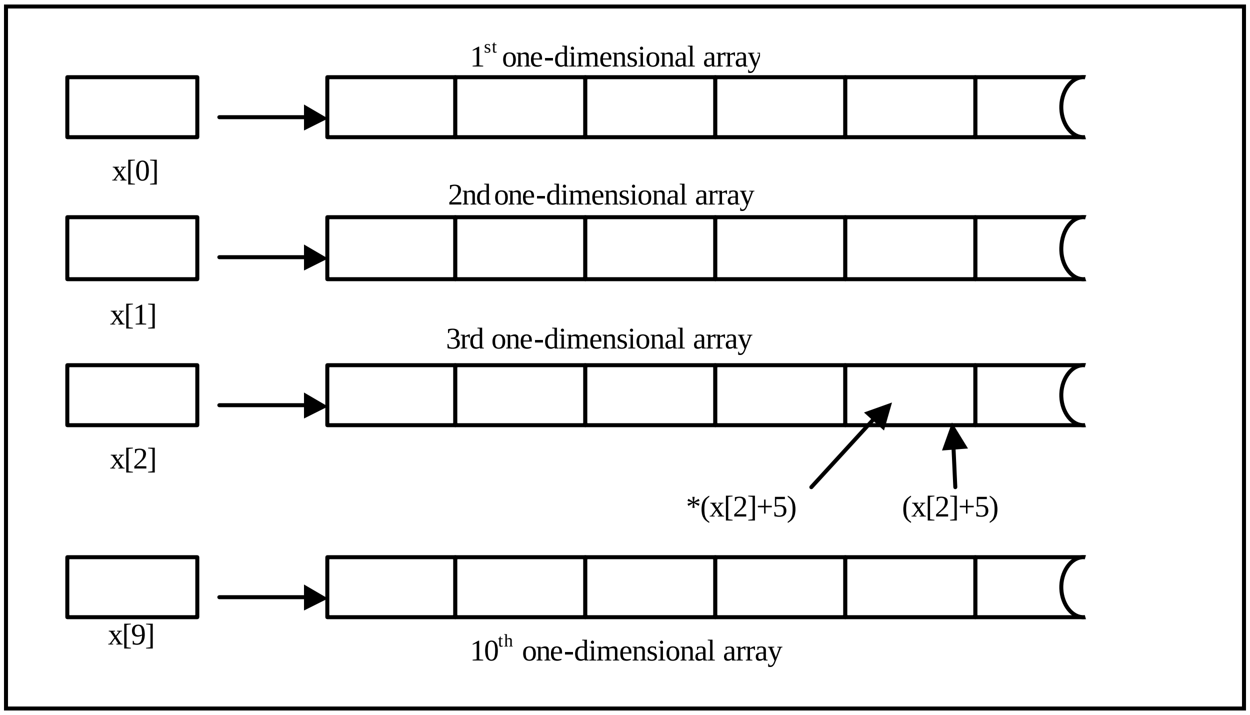

1st one-dimensional array
x[0]
2nd one-dimensional array
x[1]
3rd one-dimensional array
x[2]
*(x[2]+5)
(x[2]+5)
x[9]
10th one-dimensional array

Fig 6.6: Memory representation of pointer expression *(x[2]+5)

Here *x* is an array of 10 pointers (x[0] to x[9]). Memory from the heap has to be allocated for each pointer so that valid data can be stored in it. Now x[0] to x[9] can be treated as normal character pointers.

For example, the statement

```
        puts (x[i]) ;
```

will print the string to which x[i] points to.

6.11 Pointers to Pointers

A pointer provides the address of the data item pointed to by it. The data item pointed to by a pointer can be an address of another data item. Thus , a given pointer can be a pointer to a pointer to a data item. Accessing this data item from the given pointer then requires two levels of indirection. First, the given pointer is dereferenced to get the pointer to the given data item, and then this later pointer is dereferenced to get to the data item.

The general format for declaring a pointer to pointer is

```
        data-type **ptr_to_ptr;
```

The declaration implies that the variable *ptr_to_ptr* is a pointer to a pointer pointing to a data item of the type *data_type*.

For example, the declarations

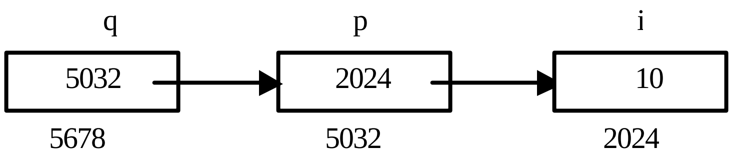
```
int i=10;
int *p;
```

declare i as an integer and p, a pointer to an integer.

We can assign the address as follows

```
p=&i;          /* p points to i */
q=&p;          /* q points to p */
```

The relationship between *i*, *p* and *q* is pictorially depicted below



This imply that *q* is the pointer to *p*, which in turn points to the integer *i*. We can indicate this by writing

```
**q;
```

which means “apply the dereferencing operator to *q* twice”.

The variable *q* is declared as :

```
int **q;
```

To get the value of *i*, starting from *q*, we go through two levels of indirection. The value of *q is the content of *p* which is the address of *i*, and the value of ***q* is **(&i)* which is 1. It can be written in different ways as

```
*(*q) = i =1
*(p) = i =1
*(*(&p)) = i =1
**(q) = i =1
```

Thus, the each of the expressions

```
i+1;
*p+1;
**q+1;
```

has the value 2.

There is no limit on the number of levels of indirection, and a declaration such as

```
int ***p;
```

Thus,

***p	is an integer.
**p	is a pointer to an integer.
*p	is a pointer to a pointer to an integer.
p	is a pointer to a pointer to a pointer to an integer.

```
/* Program to demonstrate the use of pointer to pointer */
#include <stdio.h>
main()
{
        int i=3, *j, **k;
        j = &i;
        k = &j;

        printf("The value of i = %d\n", i);
        printf("The value of j = %u\n", j);
        printf("The value of k = %u\n", k);

        printf("\nAddress of i : \n");
        printf("Using i (&i) = %u \tUsing j (j) = %u \tUsing k (*k) = %u \n", &i, j, *k);
        printf("Value of i : \n");
        printf("Using i (i) = %d \tUsing j (*j) = %d \tUsing k (**k) = %d \n", i, *j, **k);

        printf("\nAddress of j : \n");
        printf("Using j (&j) = %u \tUsing k (k) = %u \n", &j, k);
        printf("Value of j : \n");
        printf("Using j (j) = %u \tUsing k (*k) = %u \n", j, *k);

        printf("\nAddress of k : \n");
        printf("Using k (&k) = %u \n", &k);
}
```

Pointers to pointers offer flexibility in handling arrays, passing pointer variables to functions, etc.

```
/* Example :Pointers to Pointers */
#include <stdio.h>
void main(void)
{
          int data;
          int *iptr;                  /* pointer to an integer data */
          int **ptriptr;              /* pointer to int pointer */
          iptr = &data;              /* iptr points to data */
          ptriptr = &iptr;           /* ptriptr points to iptr */
          *iptr = 100;              /* same as data=100 */
          printf("Variable data :%d \n", data);
          **ptriptr = 200;          /* same as data=200 */
          printf("variable data :%d \n", data);
          data = 300;
          printf("ptriptr is pointing to :%d \n", **ptriptr);
}
Output :
        Variable data :100
        Variable data :200
        ptriptr is pointing to :300
```

Following program illustrates use of **pointer to pointers**.

The program allows the user to enter number of rows and columns of a double-dimensional array at runtime with **malloc()** function. It then asks the user to enter numbers to store in different subscripts.

```
# include <stdio.h>
# include <malloc.h>
void main(void)
{
          int j, i, row, col, **ptr;
          printf("Enter Number of Rows : ");
          scanf("%d", &row);
          ptr=(int **)malloc(sizeof(int *) * row);
          printf("Enter Number of Columns : ");
          scanf("%d", &col);
          for(i=0; i<=col; i++)
          {
                    ptr[i]=(int *)malloc(sizeof(int)* col);
          }
```

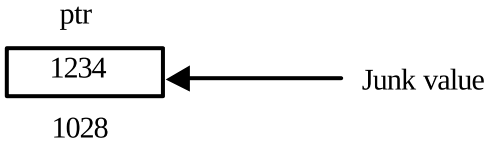
memory allocation for rows

memory allocation for columns

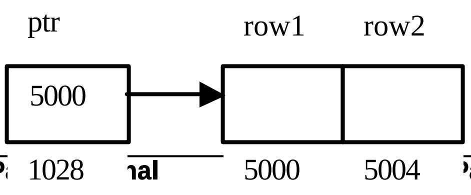
```
        printf(“\n\n”);

        /* accepting elements of the array */
        printf(“Enter no.s :\n”);
        for(i=0; i < row; i++)
        {
                for(j=0; j<col; j++)
                {
                        scanf(“%d”, ptr[i]+j);
                        fflush(stdin);
                }
        }
        /* displaying elements of the array */
        printf(“Numbers :\n”);
        for(i=0; i < row; i++)
        {
                for(j=0; j<col; j++)
                {
                  printf(“%d\t”, ptr[i][j]);
                }
                printf(“\n”);
        }
}
Output:
                Enter Number of Rows :2
                Enter Number of Columns :2
                Enter no.s :
                        10 12 14 16
                Numbers :
                                10 12
                                14 16
```

Pictorial Representation of above code:

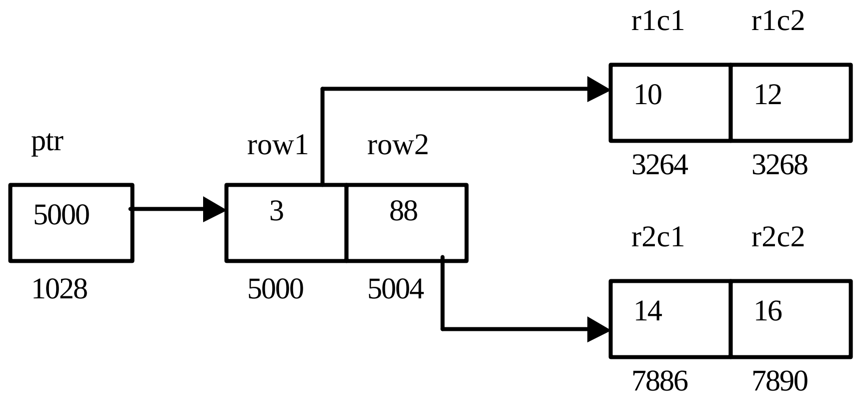
At Declaration



After allocating memory for row



After allocating memory for columns



***(*(ptr+0)+0)**	**Value at r1c1**	**10**
***(*(ptr+0)+1)**	**Value at r1c2**	**12**
***(*(ptr+1)+0)**	**Value at r2c1**	**14**
***(*(ptr+1)+1)**	**Value at r1c1**	**16**

6.12 Dynamic Memory Allocation

In many programs, the number of data items to be processed by the program and their sizes are not known.

C provides a collection of dynamic memory management functions that enable storage to be allocated as needed and released when no longer required. Their prototypes are declared in **alloc.h** header file (under Borland C) and in **malloc.h** header file (under Unix and Windows).

The allocation of memory in this manner, as it is required, is known as 'Dynamic Memory Allocation'.

6.12.1 void* malloc(size)

malloc() is used to obtain storage for a data item. The allocation of storage by calling this function yields a pointer to the beginning of the storage allocated and is suitably aligned, so that it may be assigned to a pointer to any type of data item.

Suppose that *x* is to be defined as a one-dimensional, 10-element array of integers. It is possible to define x as a pointer variable rather than as an array. Thus, we can write

```
        int *x;
```

instead of

```
int x[10];
```

or instead of

```
# define SIZE 10
int x[SIZE];
```

However, *x* is not automatically assigned a memory block when it is defined as a pointer variable, where as block of memory large enough to store 10 integer quantities will be reserved in advance when x is defined as an array.

To assign sufficient memory for x, we can make use of the library function **malloc()**, as follows

```
x = (int *)malloc(10 * sizeof(int));
```

Format

```
void* malloc(size);
```

Where size is the number of bytes required.

Note: Since malloc() returns a pointer to void data type it needs to be typecasted. void* return type can be used to make general purpose functions.

To be consistent with the definition of *x*, we really want a pointer to an integer. Hence, we include a type cast to be on the safe side, that is,

```
x = (int *) malloc(10 * sizeof(int));
```

For example, if

```
float *fp, ma[10];
```

then

```
fp=(float *)malloc(sizeof(ma));
```

allocates the storage to hold an array of 10 floating point elements and assigns the pointer to this storage to *fp*.

6.12.2 void* calloc(nitems, size)

```
void *calloc(nitems, size)
```

calloc() function work exactly similar to malloc() except for the fact that it needs two arguments as against the one argument required by malloc().

General format for memory allocation using **calloc()** is

```
void *calloc(nitems, size);
```

where,

nitems	The number of items to allocate
Size	Size of each item

For example,

```
ar=(int *)calloc(10, sizeof(int));
```

allocates the storage to hold an array of 10 integers and assigns the pointer to this storage to *ar*.
Note: While allocating memory using calloc(), the number of items which are allocated, are initialized.

6.12.3 void* realloc(void *block, size)

```
     void *realloc(void *block, size)
```

General format for memory allocation using **realloc()** is

```
     void *realloc(void *block, size);
```

where,

block	Points to a memory block previously obtained by calling malloc(), calloc() or realloc().
size	New size for allocated block.

realloc() returns a pointer to the new storage and NULL if it not possible to resize the data item, in which case the data item (*block) remains unchanged. The new size may be larger or smaller than the original size. If the new size is larger, the original contents are preserved and the remaining space is uninitialized; if smaller, the contents are unchanged up to the new size.

Note: The function realloc() works like malloc() for the specified size if block is a null pointer.

For example, if

```
     char *cp;
     cp=(char *)malloc(sizeof("computer");
     strcpy(cp, "computer");
```

then cp points to an array of 9 characters containing string "computer".

The function call,

```
     cp=(char *)realloc(cp,sizeof("compute");
```

discards the trailing '\0' and makes *cp* point to an array of 8 characters containing the string "compute".
whereas the call

```
     cp=(char *)realloc(cp, sizeof("computerization");
```

makes cp point to an array of 16 characters.

6.12.4 free(ptr)

The memory allocated by the malloc(), calloc or realloc() function is not destroyed automatically. It has to be cleared by using the function **free()**.

```
     free(ptr);
```

where,

ptr is a pointer variable to which memory was allocated, free (ptr) clears the memory to which ptr points to.

```
/* Example : function returning void*   */
 #include <stdio.h>
 #include <malloc.h>
void main(void)
{
     void *message(void);
     int *int_ptr;
     char *char_ptr;
     int_ptr = (int *)message();
     char_ptr = (char *)message();
     printf("int= %d , char=%5.2s\n", *(int_ptr), char_ptr);
 }
void *message(void)
{
     int *i;
     i = (int *)malloc(sizeof(int));
     *i = 16707;
     return i; /* returning pointer to int */
 }
```

allocating memory for int pointer

This function reserves a block of memory whose size (in bytes) is equivalent to the size of an integer quantity. The function returns a pointer to void type, which can safely be converted to a pointer of any type.

```
/* Example using malloc(), realloc() and free() */
#include <stdio.h>
#include <stdlib.h>
void main(void)
{
        int *marks = (int*)malloc(4 * sizeof(int));
        int mark, i, n=0, siz=3;
        /* mark is current student's mark,
           n is number of marks input upto now,
           siz is integer which had current size of the array */
        printf("\nEnter marks(Enter -1 to stop) : \n");
        scanf("%d", &mark);
        while(mark != -1)
        {
                if(n >= siz)
                {
                        siz += 4;
                        printf("Reallocate 4 more integers...Success\n");
                        marks=(int*)realloc(marks, siz * sizeof(int));
                if(marks==(int *)NULL)
               {
                       printf("Not enough memory! \n");
                         exit(1);
              }
             else
                       printf("Enter marks for 4 students(-1 to stop) \n");
          }
          marks[n]=mark;
          fflush(stdin);
          scanf("%d",&mark);
          n++;
   }
   /* Output the marks */
   printf("The marks entered are.. \n");
   for (i=0; i<n; i++)
           printf("%d ", marks[i]);
}
Output:
           Enter marks(Enter –1 to stop):
           10
           20
           30
           40
           Reallocate 4 more integers...Success
           Enter marks for 4 students(-1 to stop)
           50
           60
           70
     80
           Reallocate 4 more integers...Success!
           Enter marks for 4 students(-1 to stop)
           -1
           The marks entered are..
           10 20 30 40 50 60 70 80
```

A buffer of 4 integers is allocated first, and then increase its size by 4 integers (16 bytes) every time the buffer overflows.

6.13 Pointer Declarations

int *p ;	p is pointer to integer.
int *p[10] ;	p is a 10-element array of pointers to integer
int (*p)[10] ;	pointer to 10-element integer array.
int *p(void) ;	p is a function that returns pointer to integer, argument is void
int p(char *a);	p is a function, which returns integer, argument, is char pointer.
int *p(char *a) ;	p is a function which returns pointer to integer argument is char pointer.
int (*P)(char *a) ;	P is a pointer to a function that returns integer, argument is char pointer
int (*P(char *a))[10] ;	P is function that returns pointer to 10-element integer array, the argument is char pointer.
int p(char (*a[]) ;	P is function that returns integer, accepts pointer to char array.
int p(char *a[]);	P is a function that returns integer, accepts array of pointers to character as argument.
int *p(char(*a[]) ;	p is a function returns pointer to integer, argument is pointer to char array.
int *(*p)(char (*a[]) ;	p is a pointer to function, returns int pointer accepts pointer to char array as argument

Table 6.1: Pointer declarations.

6.14 Command Line Arguments

Parameters or Values can be passed to a program from the command line which are received and processed in the main function. Since the arguments are passed from the command line hence they are called as command line arguments. This concept is used frequently to create command files. All commands on the Unix Operating System use this concept.

```
Eg:
    C:\>CommLineTest.exe arg1 arg2 arg3 ….
```

where,

CommLineTest.exe	Executable file of the program
arg1, arg2, arg3…	Actual parameters for the program

Two built in formal parameters are used to accept parameters in main.

argc : **contains number of command line arguments. It is of type int.**
argv : **A pointer to an array of strings where each string represents a token of the arguments passed. It is a character array of pointers.**

Eg:

```
C:\>Tokens.exe abc 10 xyz
```

The value of argc will be 4.
The contents of argv will be

argv[0]	"Tokens.exe"
argv[1]	"abc"
argv[2]	"10"
argv[3]	"xyz"

Fig 6.7: argv

The main in a command line argument program will look as follows

```
main(int argc, char *argv[])
```

All data types int, float or char are accepted in argv as strings. So to perform mathematical operations with them, they must be converted to int. This can be done using the function atoi(argv[2]). This will convert the argument to int. Similarly there are functions atof, atol. The header file stdlib.h must be included while using these functions.

7 Structures

Arrays provide the facility for grouping related data items of the same type into a single object. However, sometimes we need to group related data items of different types. An example is the inventory record of a stock item that groups together its item number, price, quantity in stock, reorder level etc. In order to handle such situations, C provides a data type, called `structures`, that allows a fixed number of data items, possibly of different types to be treated as a single object. It is used to group all related information into one variable.

7.1 Basics of Structures

`Structure` is a collection of logically related data items grouped together under a single name, called a `structure tag`.

The data items that make up a structure are called its `members` or `fields`, and can be of different types.

The general format for defining a structure is:

```
struct tag_name
{
     data_type member1;
     data_type member2;
     ....
};
```

Fig 7.1: Format for defining a structure

where,

struct	A keyword that introduces a structure definition.
Tag_name	The name of the structure
member1, member2	Set of type of declarations for the member data items that make up the structure.

For example, the structure for the inventory record of a stock item may be defined as

```
struct item
{
     int itemno;
     float price;
     float quantity;
     int reorderlevel;
};
```

Consider another example, of a book database consisting of book name, author, number of pages and price.

To hold the book information, the structure can be defined as follows

```
struct book_bank
{
     char title[15];
     char author[10];
     int pages;
     float price;
};
```

The above declaration does not declare any variables. It simply describes a format called template to represent information as shown below:

struct book_bank

title	array of 15 characters
author	array of 10 characters
pages	integer
price	float

Following figure illustrates the composition of this book database schematically.

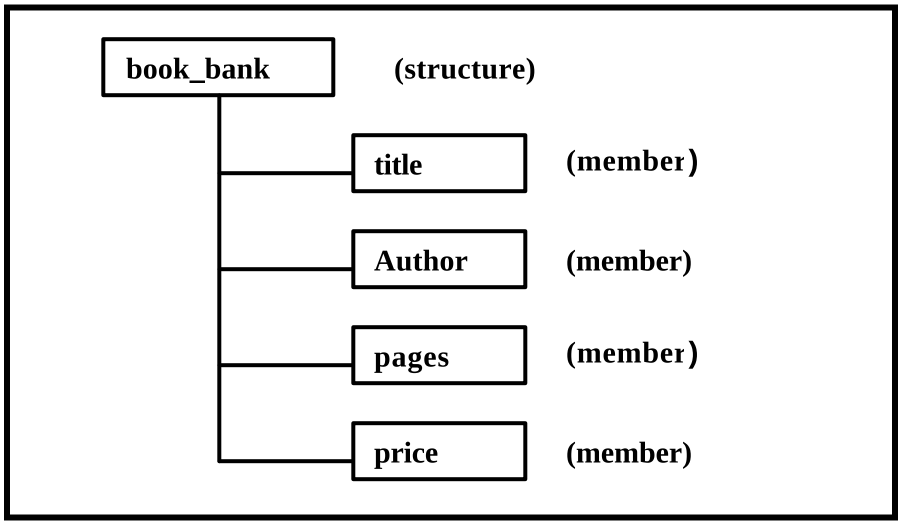


Fig 7.2: Structure for a book

All the members of a structure can be of the same type, as in the following definition of the structure date

```
struct date
{
     int day,month,year;
};
```

7.1.1 Declaration of Individual Members of a Structure

The individual members of a structure may be any of the common data types (such as int, float, etc.), pointers, arrays or even other structures.

All member names within a particular structure must be different. However, member names may be the same as those of the variables declared outside the structure.

Individual members cannot be initialized inside the structure declaration.

7.1.2 Structure Variables

A structure definition defines a new type, and variables of this type can be declared in the following ways

In the structure declaration: By including a list of variable names between the right brace and the termination semicolon in the structure definition.

For example, the declaration

```
struct student
{
     int rollno;
     char subject[10];
     float marks;
} student1, student2;
```

declares *student1*, *student2* to be variables of type struct *student.*

If other variables of the structure are not required, the tag name student can be omitted as shown below

```
struct
{
     int rollno;
     char name[10];
     float marks;
} student1, student2;
```

Using the structure tag

The structure tag can be thought of as the name of the type introduced by the structure definition and variables can also be declared to be of a particular structure type by a declaration of the form:

```
struct tag variable-list;
```

For example,

```
struct student student1,student2;
```

declares *student1* and *student2* to be variables of type struct *student.*

7.1.3 Structure Initialization

A variable of particular structure type can be initialized by following its definition with an initializer for the corresponding structure type. Initializer contains initial values for components of the structure, placed within curly braces and separated by commas.

Thus, the declaration

```
struct date
{
    int day,month,year;
}independence={15,8,1947};
```

initializes the member variables day, month and year of the structure variable *independence* to 15, 8 and 1947 respectively.

The declaration

```
     struct date republic ={26,1,1950};
```

initializes the member variables day, month and year of the structure variable republic to *26*, *1* and *1950* respectively.

Considering the structure definition student (defined in 8.1.2), the declaration

```
     struct student student1={1,”Ashwini”,98.5};
```

initializes the member variables rollno, name and marks of the structure variable student1 to 1, “Ashwini” and 98.5 respectively.

If there are fewer initializers than that of member variables in the structure, the remaining member variables are initialized to zero.

Thus the initialization

```
     struct date newyear={1,1};
```

is same as

```
     struct date newyear={1,1,0};
```

7.1.4 Accessing Structure Members

With the help of `dot operator(.)`, individual elements of a structure can be accessed and the syntax is of the form

```
     structure-variable.member-name;
```

Thus to refer to name of the structure student, we can use

```
     student1.name;
```

The statements,

```
struct date emp;                    (date is defined in 8.1.3)
emp.day=28;
emp.month=7;
emp.year=1969;
```

set the values of the member variables *day*, *month* and *year* within the variable *emp* to 28, 7 and 1969 respectively and the statement

```
struct date today;
if(today.day==1&&today.month==1)
        printf("Happy New Year");
```

tests the values of *day* and *month* to check if both are 1 and if so, prints the message.

The elements of a structure are always stored in contiguous memory locations. It is shown below

emp.day	emp.month	emp.year
28	7	1969

Following are some example given using structures

```
/* Program to print the date using structure variable */
# include<stdio.h>
void main(void)
{
struct date
{
     char month[15];
     int day,year;
 };
 struct date today;
 today.day=11;

printf("Enter Month : ");
 scanf("%[^\n]",today.month);
 today.year=1998;
 printf("\nToday's date is %d-%s-%d \n",
     today.day,today.month,today.year);
}
```

Structure definition

defining a structure variable

accessing and initializing structure member

```
          *** str.h ***
struct date
{
int month,day,year;
};
```

```
*** prog.c***
/* Program prompts the user for today’s date and prints
   tomorrow’s date */
 # include<stdio.h>
 # include “str.h”
 void main(void)
 {
     struct date today;
     struct date tomorrow;
     static int day_month[12]=
        {31,28,31,30,31,30,31,31,30,31,30,31};
     printf(“Enter Today’s date (dd:mm:yy): ”);
     scanf(“%d%d%d”,&today.day,&today.month,&today.year);
     if(today.day > day_month[today.month-1])
     {
          printf(“\n Invalid Date \n”);
          exit(0);
     }
     if(today.day!=day_month[today.month-1])
     {
          tomorrow.day=today.day+1;
          tomorrow.month=today.month;
          tomorrow.year=today.year;
     }
     else if(today.month==12)
     {
          tomorrow.day=1;
          tomorrow.month=1;
          tomorrow.year=today.year+1;
     }
     else
     {
          tomorrow.day=1;
          tomorrow.month= today.month+1;
          tomorrow.year=today.year;
     }
     printf(“\n Tomorrow’s date is %d-%d-%d \n”,
     tomorrow.day,tomorrow.month,tomorrow.year);
 }
```

One structure can be copied to another structure of same type directly using the assignment operator as well as element by element basis like arrays.

In this case, the values of members of a structure variable get assigned to members of another structure variable of the same type.

It is illustrated in the following example.

```
    *** strdef.h ***
 struct date
 {
     char month[5];
     int day,year;
 };
/* Example - To copy a structure to another structure */
 # include <stdio.h>
 # include <string.h>
 # include “strdef.h”
 void main(void)
 {
     struct date today={“March”,1,98};
     struct date day1,day2;
 /* copying element by element basis */
     strcpy(day1.month,today.month);
     day1.day=today.day;
     day1.year=today.year;
 /* copying entire structure to another structure */
     day2=day1;
     printf(“\n Date is %d %s %d \n”,
          today.day,today.month,today.year);
     printf(“\nDate is %d %s %d \n”,
          day1.day,day1.month,day1.year);
     printf(“\n Date is %d %s %d \n”,
          day2.day,day2.month,day2.year);
 }
```

accessing structure date defined in strdef.h

7.2 Nested Structures

The individual members of a structure can be other structures as well. It is termed as `Nested Structures`. We will include a new member *date* which itself is a structure. It can be done in two ways.

The first way is by declaring

```
struct date
{
     int day,month,year;
};
struct emp
{
     char name[15];
     struct date birthday;
     float salary;
};
```

The embedded structure *date* must be declared before its use within the containing structure.

The second way is by declaring

```
struct emp
{
     char name[15];
     struct date
     {
          int day,month,year;
     }birthday;
     float salary;
};
```

In this method, we combine the two structure declarations. The embedded structure *date* is defined within enclosing structure definition.

In the first case, where the *date* structure is declared outside the *emp* structure, it can be used directly in other places, as an ordinary structure. This is not possible in second case.

Variables of a nested structure type can be defined as usual. They may also be initialized at the time of declaration as

```
struct emp
{
     char name[15];
     struct date
     {
          int day,month,year;
     }birthday;
     float salary;
}person = {“Ashwini”,{28,7,1969},5000.65};
```

The inner pair of braces is optional.

A particular member inside a nested structure can be accessed by repeatedly applying the dot operator. Thus the statement

```
     person.birthday.day=28;
```

sets the day variable in the birthday structure within person to 28.

The statement

```
     printf(“%d-%d-%d”,person.birthday.day,person.birthday.month,
                              person.birthday.year);
```

prints date of birth of a person.

However, a structure can not be nested within itself.

7.3 Structures and Arrays

Arrays and structures can be freely intermixed to create arrays of structures, structures containing arrays.

7.3.1 Arrays of Structures

In the array of structures array contains individual structures as its elements. These are commonly used when a large number of similar records are required to be processed together.

For example, the data of motor containing 1000 parts can be organized in an array of structure as

```
        struct item motor[1000];
```

This statement declares *motor* to be an array containing 1000 elements of the type struct *item.*

An array of structures can be declared in two ways as illustrated below.

The first way is by declaring

```
struct person
{
     char name[10];
     struct date birthday;
     float salary;
}emprec[15];
```

In this case, *emprec* is an array of 15 *person* structures. Each element of the array *emprec* will contain the structure of type *person*. The *person* structure consists of 3 individual members : an array *name*, *salary* and another structure *date*.

The embedded structure *date* must be declared before its use within the containing structure.

The second approach to the same problem involves the use of the structure tag as below.

```
struct person
{
     char name[10];
     struct date birthday;
     float salary;
};
struct person emprec[15];
```

Following program explains how to use an array of structures.

```
/* Example- An array of structures */
# include<stdio.h>
void main(void)
{
    struct book
    {
         char name[15];
         int pages;
         float price;
    };
    struct book b[10];
    int i;
    printf(“\n Enter name, pages and price of the book\n”);
/* accessing elements of array of structures */
    for(i=0;i<9;i++)
    {
         scanf(“%s%d%f”,b[i].name,&b[i].pages,&b[i].price);
         printf(“\n”);
    }
    printf(“\n Name, Pages and Price of the book :\n”);
    for(i=0;i<=9;i++)
    {
         printf(“%s %d %f”,b[i].name,b[i].pages,b[i].price);
    }
}
```

7.3.2 Arrays within Structures

A structure may contain arrays as members. This feature is frequently used when a string needs to be included in a structure. For example, the structure date (declared in 8.1.3) can be expanded to also include the names of the day of the week and month as

```
struct date
{
    char weekday[10];
    int day;
    int month;
    char monthname[10];
    int year;
};
```

A structure variable *ndate* can be declared and initialized as –

```
struct date ndate={”Sunday”,21,11,”November”,2004};
```

An element of an array contained in a structure can be accessed using the dot and array subscript operators.

Thus the statement,

```
printf("%c",ndate.monthname[2]);
```

prints **v**.

7.4 Structures and Pointers

7.4.1 Pointers to Structures

The beginning address of a structure can be accessed in the same manner as any other address, through the use of the address of (**&**) operator.

Thus, if variable represents a structure-type variable, then

```
&variable
```

represents the starting address of that variable. Moreover, we can declare a pointer variable for a structure by writing

```
type *ptvar;
```

where,

type	A data type that identifies the composition of the structure
ptvar	The name of the pointer variable

Pointer variable holding address of structure is called **Structure Pointers**.

For example, the declaration

```
struct date ndate,*ptrndate;
```

declares *ndate* to be a variable of type *struct date* and the variable ptrndate to be a pointer to a ***struct date*** **variable.**

Consider the following example

Consider the following structure declaration, in

```
typedef struct
{
     int acct_no;
     char acct_type;
     char name[20];
     float balance;
     date lastpayment;
}account;
account customer,*pc;
```

In this example, *customer* is a structure variable of type *account*, and *pc* is a pointer variable whose object is a structure of type *account.*

The address operator (&) is applied to a structure variable to obtain the beginning address of *customer*. It can be assigned to *pc* by writing

```
pc=&customer;
```

The variable and pointer declarations can be combined with the structure declaration by writing

```
struct
{
     member 1;
     member 2;
     ...
     member n;
}variable,*ptvar;
```

Where,

variable	A structure type variable
ptvar	The name of a pointer variable

The following single declaration is equivalent to the two declarations presented in the previous example

```
struct
{
     int acct_no;
     char acct_type;
     char name[20];
     float balance;
     date lastpayment;
}customer,*pc;
```

The pointer variable *pc* can now be used to access the member variables of *customer* using the dot operator as

```
     (*pc).acct_no;
     (*pc).acct_type;
     (*pc).name;
```

The parentheses are necessary because the dot operator(**.**) has higher precedence than that of the dereferencing operator(*****).

The members can also be accessed by using a special operator called **the structure pointer or arrow operator (->)**.

The general form for the use of the operator **->** is

```
        printer_name->member_name;
```

Thus,

```
        if  pc=&customer
                pc->balance=(*pc).balance=customer.balance
```

where, balance is member of structure *customer*.

It is possible to take addresses of the member variables of a structure variable.

For example, the statement

```
        float *ptrbal=&customer.balance;
```

defines *ptrbal* to be a floating point pointer and initializes it to point to the member variable *balance* within the structure variable *customer*.

The pointer expression *&customer.balance* is interpreted as *&(customer.balance)* since, the precedence of the dot operator is higher than that of the address operator.

```
/* Example - structure pointers */
# include <stdio.h>
# include "str.h"
struct
{
        int acct_no;
        char acct_type;
        char *name;
        float balance;
        struct date *lastpayment;
}customer, *pc = &customer;
struct date PaymentDate ;
void main(void)
{
        PaymentDate.day = 26 ;
        PaymentDate.month = 1 ;
        PaymentDate.year = 1999 ;
        customer.acct_no=55;
        customer.acct_type='A';
        customer.name="Ashwini";
        customer.balance=99.99;
        customer.lastpayment = &PaymentDate ;
        printf("Account:%d\n",pc->acct_no);printf("Acc_Type  : %c \n",pc->acct_type);
        printf("Name       : %s \n",pc->name);
        printf("Balance    : %.2f \n",pc->balance);
        printf("LastPayment : %2d-%2d-%4d \n",
                pc->lastpayment->day,pc->lastpayment->month,
                pc->lastpayment->year);
}
```

Within the second structure, the members acct_no, acct_type, name and balance are written as pointers. Thus, the value to which acct_no points can be accessed by writing either *customer.acct_no or *p->acct_no. Same in case of acct_type and balance.

A string can be assigned directly to a character type pointer. Therefore, if name points to the beginning of a string, then the string can be accessed by writing either customer.name or pc->name.

Allocating Memory for Pointer to a Structure

Memory from the heap is to be allocated for a pointer to a structure if you want to store some data, this is done by using **malloc()** function.

Example:

```
typedef struct
{
        char name[20];
        char address[20];
        int empid;
}emp,*empptr;
```

The memory to store information about 10 employees can be allocated by the statement

```
empptr=(emp*)malloc(10*sizeof(emp));
```

After the memory is allocated you can use the pointer to get the information as follows

```
for(i=0;i<10;i++)
{
        scanf("%s%s%d",empptr[i].name,empptr[i].address, & empptr[i].empid);
}
```

7.4.2 Structures Containing Pointers

A structure can contain pointers as member variables.

For example, the structure definition

```
  struct location
 {
        char *name;
        char *addr;
  };
```

defines a structure location that contains two character pointers, name and addr as member variables. Variables of type struct location can now be defined and manipulated as in:

```
struct location att={"Ashwini","Boston's Computer Institute"};
struct location ibm;
ibm.name="R&D";
ibm.addr="Bell Labs,California";
```

```
          *** str1.h ***
# include “str.h”
struct person
{
    char name[20];
    char *lastname;
    struct date birthday;
    float *salary;
}emprec;
```

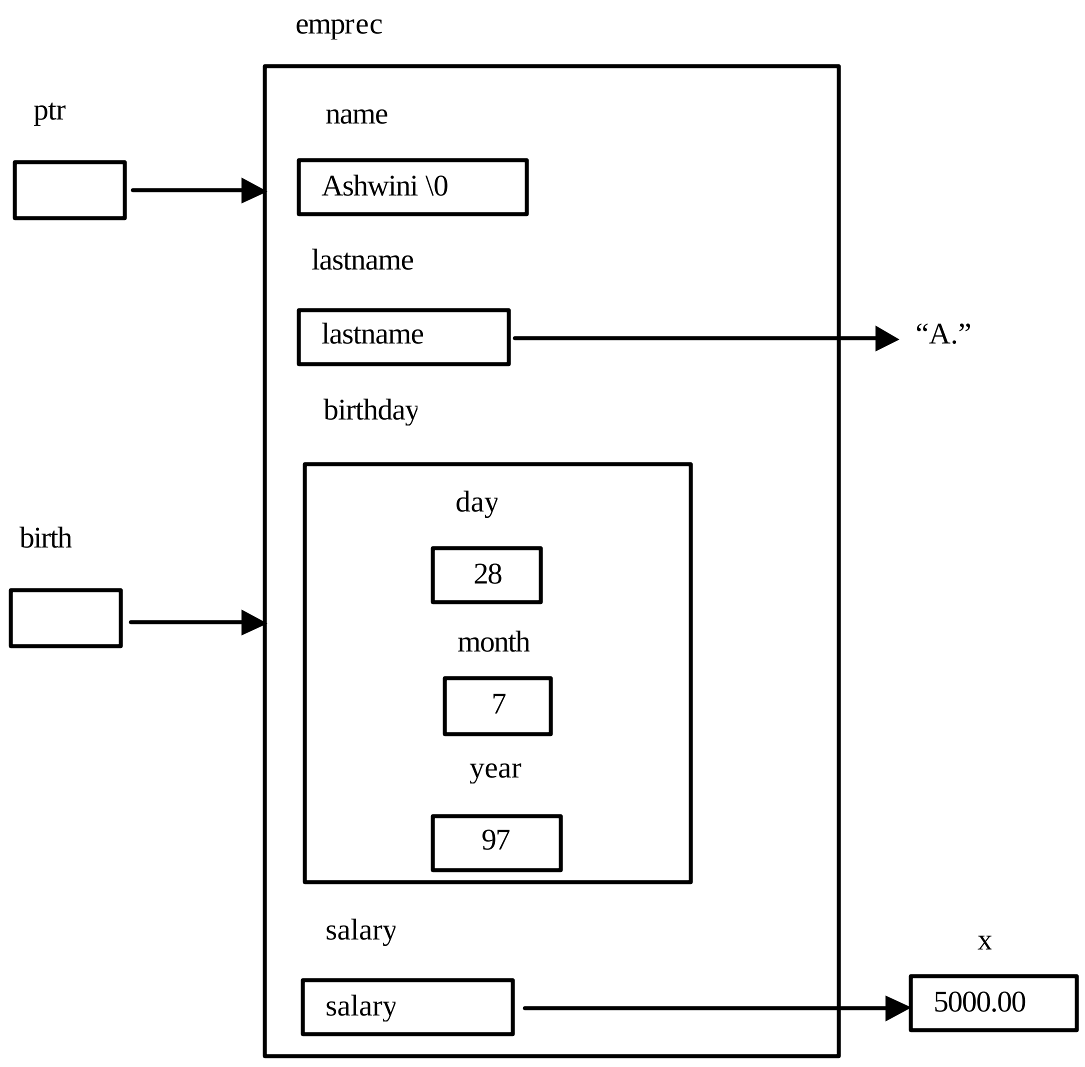
accessing structure date defined in str.h

*** strptr.c***

```
/* Example- structure containing pointers */
# include<stdio.h>
# include “str1.h”
void main(void)
{
        float x;
        struct person *ptr = &emprec;
        struct date *birth = &emprec.birthday;
        strcpy(emprec.name,”Ashwini”);
        emprec.lastname = ”A.”;
        ptr->birthday.day = 28;
        emprec.birthday.month = 7;
        birth->year = 97;
        x=5000;
        ptr->salary = &x;
        printf(“ *** Employee Details *** \n”);
        printf(“Name :%s %s \n”,ptr->name,ptr->lastname);
        printf(“Birthdate: %d:%d:%d \n”,(*ptr).birthday.day,
                        birth->month,emprec.birthday.year);

        printf(“Salary :%6.2f”,emprec.salary);
}
```

```
Output:
      *** Employee Details ***
            Name: Ashwini A.
            Birthday: 28:7:97
            Salary: 5000.00
```

7.5 Pictorial Representation of Above Code



Line 3 includes definition of struct person as well as the variable emprec as part of the program.
Line 7 declares a pointer to emprec.
Line 8 declares a pointer to the structure birthday which is part of emprec.
Note the different methods of accessing structure elements in lines 11, 12, 13.
Line 15 initializes salary to point to x.

Allocating Memory for Structure containing Pointer

When there is member of a structure, which is pointer to a structure, it is not enough to allocate memory for the pointer to the structure but you have to allocate memory for member pointer too.

```
Example:
typedef struct
{
     char* name;
     char* address;
     int empid;
}emp,*empptr;
```

- Following program illustrates memory allocation of pointer within structure. The program allows the user to enter total number of employees and size of name at runtime.

```
# include <stdio.h>
# include <alloc.h>
# include <string.h>
void main(void)
{
    int n,i,j;
    typedef struct
    {
          int empno;
          char *name;
    }emp;
    emp *empptr;
    char name[80] ;
    printf("Enter total no. of employees:");
    scanf("%d",&n);
    fflush(stdin);
    empptr = (emp *) malloc(n * sizeof(emp));
    for(i = 0 ; i < n ; i++)
    {
          printf("\n Enter empno of employee (%d) :",i+1);
          scanf("%d",&empptr[i].empno);
          fflush(stdin);
          printf("\n Enter name :");
          scanf("%[^\n]",name);
          fflush(stdin);
          empptr[i].name = (char *) malloc(strlen(name) *
                                 sizeof(char) + 1 );
          strcpy(empptr[i].name, name) ;
    }
    for(i=0;i < n ; i++)
    {
          printf("\nno-%d \tname-%s",empptr[i].empno,
                         empptr[i].name);
    }
}
```

7.6 Structures and Functions

A structure type definition may be local to a function or it may be external to any function. Structures may be passed as function arguments and functions may return structures.

7.6.1 Structures as Function Arguments

C provides three methods of passing structures to a function. They are explained below:

Passing Structure Member to Function

This method involves supplying structure members as the arguments in a function call. These arguments are then treated as separate non-structure values, unless they themselves are structures.

To illustrate this method, an example is given below

```
    *** str2.h ***

# include “str.h”
typedef struct
{
    char name[20];
    date birthday;
    float salary;
}person, emprec;
```

accessing structure date defined in str.h

```
     *** strfun.c ***

/* Example- structure member as function arguments */

# include <stdio.h>
# include “str2.h”
# define CURRENT_YEAR 98
float increment(float sal,int year,int inc)
{
    if(CURRENT_YEAR - year > 30)
         sal += inc;
    return(sal);
}
void main(void)
{
    int n=500;
/* give increments to employees if age is grater than 30 */
    emprec per={"Rohit Tamhane",5,9,79,4000.50};
    printf(" *** Employee Details ***\n");
    printf("Name :%s \n",per.name);
    printf("Birthdate: %d:%d:%d\n",per.birthday.day,
                 per.birthday.month,per.birthday.year);
    printf("Salary :%6.2f \n\n",per.salary);
    per.salary=increment(per.salary,per.birthday.year,n);
    printf(" *** Employee Details *** \n");
    printf("Name :%s \n",per.name);
    printf("Birthdate: %d:%d:%3d \n",per.birthday.day,
                  per.birthday.month,per.birthday.year);
```

```
        printf("Salary :%6.2f \n",per.salary);
}

Output:
     *** Employee Details ***
           Name: Rohit Tamhane
           Birthday: 5:9:67
           Salary: 4000.00
     *** Employee Details ***
           Name: Rohit Tamhane
           Birthday: 5:9:67
           Salary: 4500.00
```

Structure members per.salary, per.birthday.year and n are passed to the function increment(). The parameter sal is manipulated and returned from the function increment().

The function increment() checks the age of the employee and gives an increment of 500, if his age is above 30. The amount to be incremented is also passed to increment().

The disadvantage of this method is that the relationship between the member variables encapsulated in a structure is lost in the called function. This method should only be used if a few structure members need to be passed to the called function.

Passing Entire Structure to Function

Second method involves passing the complete structure to a function by simply providing the name of the structure variable as the argument in the function call. The corresponding parameter in the called function must be of the same structure type.

To illustrate this method, an example is given below

```
/* Example- Entire structure as function arguments */
# include<stdio.h>
struct book
{
    char name[20];
    char author[10];
    int pages;
};
void main(void)
{
    void display(struct book);
    static struct book b1={"Programming in C","Stephen",300};
    display(b1);
}
void display(struct book b)
{
    printf("Name :%s\n Author :%s\nPages :%d\n",
                           b.name,b.author,b.pages);
}
```

```
Output :
          Name: Programming in C
          Author: Stephen
          Pages: 300
```

Structure book has made global by defining it outside main(), so all the functions can access the structure book.

When a structure variable b1 is passed directly as an argument to the function, it is passed by value like an ordinary variable.

Passing Structure Pointers to Functions

The third method involves passing pointers to the structure variables as the function arguments.

In the situations where, more than one member variable is computed in the function, pointers to structures are used.

If the pointer to a structure is passed as an argument to a function, then any change that are made in the function are visible in the caller.

```
/* Example- structure pointers as function arguments */
# include <stdio.h>
# include “str2.h”
# define CURRENT_YEAR 98
void increment(emprec *x)
{
     if(CURRENT_YEAR - x->birthday.year > 30)
          x->salary += 500;
}
void main(void)
{
     emprec per={"Aniruddha",27,10,62,5500};
     printf(" *** Employee Details ***\n");
     printf("Name :%s \n",per.name);
     printf("Birthdate: %d:%d:%d \n", per.birthday.day,
                    per.birthday.month,per.birthday.year);
     printf("Salary :%6.2f\n \n",per.salary);
   /* give increments to employees if age is grater than 30 */
     increment(&per);
     printf(" *** Employee Details ***\n");
     printf("Name :%s \n",per.name);
     printf("Birthdate: %d:%d:%d\n",per.birthday.day,
                    per.birthday.month,per.birthday.year);
     printf("Salary :%6.2f\n",per.salary);
}
```

```
Output :
     *** Employee Details ***
          Name: Aniruddha
          Birthday: 27:10:62
          Salary: 5500.00
     *** Employee Details ***
          Name: Aniruddha
          Birthday: 27:10:62
          Salary: 6000.00
```

The address of the per structure variable is passed to increment().

This method becomes particularly attractive when large structures have to be passed as function arguments because it avoids copying overhead.

This is illustrated in the following program.

```
/* Example - structure as function arguments */
# include<stdio.h>
# include “str2.h” /* str2.h defined above */
void main(void)
{
    void printout(emprec);
    emprec record;
    printf("Enter Name:");
    scanf("%s",record.name);
    fflush(stdin);
    printf("\n Enter Date of Birth:");
    scanf("%d%d%d",&record.birthday.day,
          &record.birthday.month,&record.birthday.year);
    fflush(stdin);
    printf("\n Enter Salary: ");
    scanf("%f",&record.salary);
    printout(record);
}
void printout(emprec per)
{
    printf(" *** Employee Details ***\n");
    printf("Name: %s \n",per.name);
    printf("Date of Birth : %d:%d:%d\n",per.birthday.day,
                    per.birthday.month,per.birthday.year);
    printf("Salary: %6.2f \n\n",per.salary);
}
Output:
     Enter Name:Anuradha
     Enter Date of Birth:3 12 46
     Enter Salary:6500
     *** Employee Details ***
          Name:Anuradha
          Birthday:3:12:46
          Salary:6500.00
```

7.6.2 Structures as Function Values

Structures can be returned from functions just as variables of any other type. Instead of accepting a pointer to a structure, it can construct a structure by itself and return this structure variable.

```
/* Example- structures and functions */
# include<stdio.h>
struct time
{
     int min,hr,sec;
};
void main(void)
{
     struct time time_udt(struct time);
     struct time or_time,nx_time;
     printf("Enter time(hh:mm:ss):");
     scanf("%d:%d:%d",&or_time.hr,&or_time.min,&or_time.sec);
     nx_time = time_udt(or_time);
     printf("Updted time is :%2d:%2d:%2d\n", nx_time.hr,
            nx_time.min,nx_time.sec);
}
struct time time_udt(struct time now)
{
     struct time new_time;
     new_time=now;
     ++new_time.sec;
     if(new_time.sec==60)
     {
           new_time.sec=0;
           ++new_time.min;
           if(new_time.min==60)
           {
                new_time.min=0;
                ++new_time.hr;
                if(new_time.hr==24)
                        new_time.hr=0;
           }
     }
     return(new_time);
}
```

A structure or_time is passed as an argument to the function *time_udt()*.

The function returns a value of type struct time.

The program prompts the user for the current time, updates the time by one second and prints it.

8 Data Structures

8.1 Linked Lists

A List refers to a set of items organized sequentially. An array is an example of list. In an array, the sequential organization is provided implicitly by its index. The major problem with the arrays is that the size of an array must be specified precisely at the beginning, which is difficult in many practical applications.

Linked list is a linked list of structures (called nodes) for which memory is allotted dynamically. It is necessary to include an additional member that is pointer to structure.

Eg :

```
struct node
{
        data_type info;
         struct node * next ;
};
```

Fig 8.1: Structure of a node in Singly linked list.

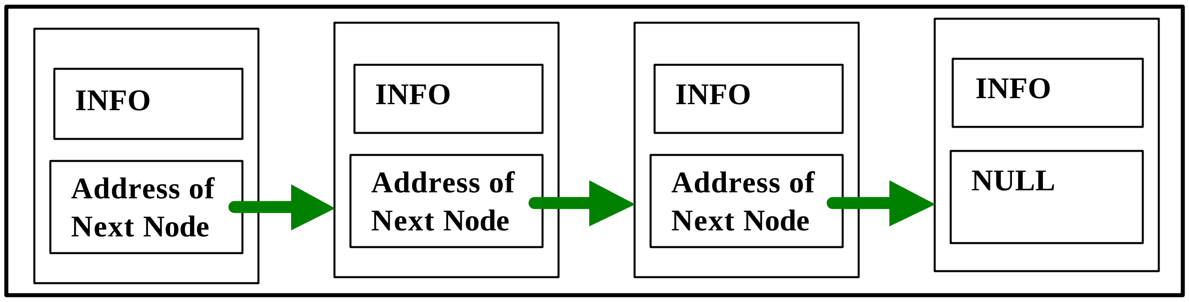


Fig 8.2: Pictorial representation of Singly linked list in Memory.

The additional member pointer next, keeps the address of the next node in the linked list. The pointer next of the last node always points to NULL. One more pointer to structure must be declared to keep track of the first node of the linked list, which is not a member pointer.

8.1.1 Creating a linked list

Creation of a linked list requires the following 3 steps to be performed.

1. Define the structure of the node that will hold the data for each element in the list. Let us assume that the data we intend to store is the empid of the employee (which is unique), his name and salary.

The structure is defined as follows:

```
struct node
{
    int empid;
    char name[20];
    float salary;
    struct node *next;
};
```

Fig 8.3: Structure for Employee node in the Linked list

Note that the last element in the structure is a pointer to the next node in the list.

2. In a linked list nodes are created dynamically as and when required. So let us create a general-purpose function that will return a node for which data has been entered by the user.

```
struct node *getnode()  /* creates a node and accepts data */
{
    struct node *temp;
    temp=(struct node *)malloc(sizeof(struct node));
    printf("enter the empid:");
    scanf("%d",&temp->empid);
    fflush(stdin);
    printf("enter the name:");
    scanf("%s",temp->name);
    fflush(stdin);
    printf("enter salary:");
    scanf("%f",&temp->salary);
    fflush(stdin);
    temp->next=NULL;
    return temp;
}
```

Fig 8.4: Code for creation of a node in the Linked list

The list needs a pointer that will point to the beginning of the list. For this, we shall create as a global pointer that can be directly accessed by all the functions.

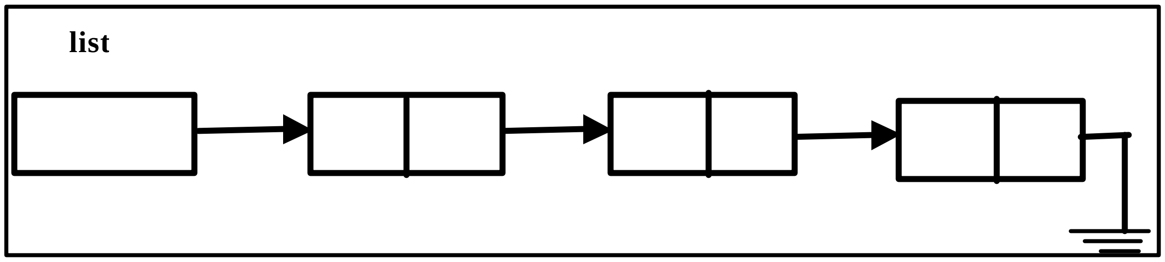


Fig 8.5: Header(list) for the Linked list

This pointer will be initialized to NULL in the main function.

3. Link the new node to the list.

Next, a function is required that will link a new node to the list. Let us call this function insert. This function will take the address of the node to be inserted as a parameter. If the list is empty, then the new node becomes the first node:

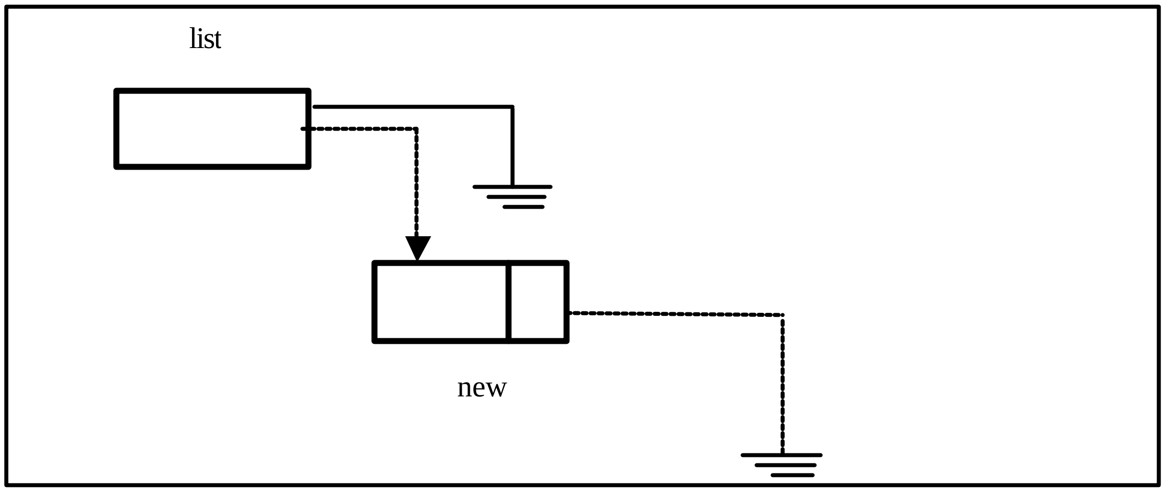


Fig 8.6: Making Header (list) to point the first node of the Linked list

Note that the dotted lines in the figure indicate the operation being performed in the list.

Assume that other nodes are required to be inserted in sorted order of empid. If the list is not empty, the new node has to be added either to the beginning of the list or to the middle or end of list:

8.1.2 To add to the beginning of the list

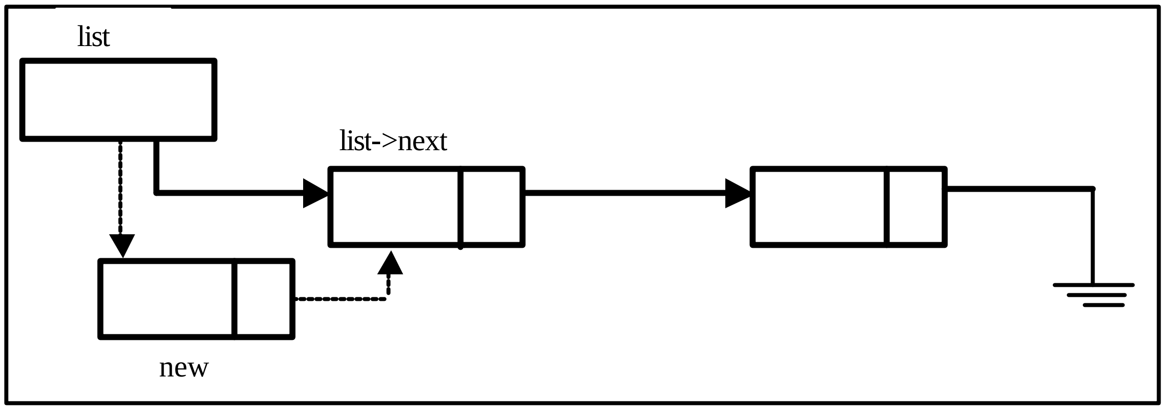


Fig 8.7: Insertion of a node at the beginning of the Linked list

The new node has to point to the existing first element of the list. This is done by the statement

```
      new->next=list;
```

and the list pointer must point to the new node:

```
      list=new;
```

8.1.3 To add to the middle of the list

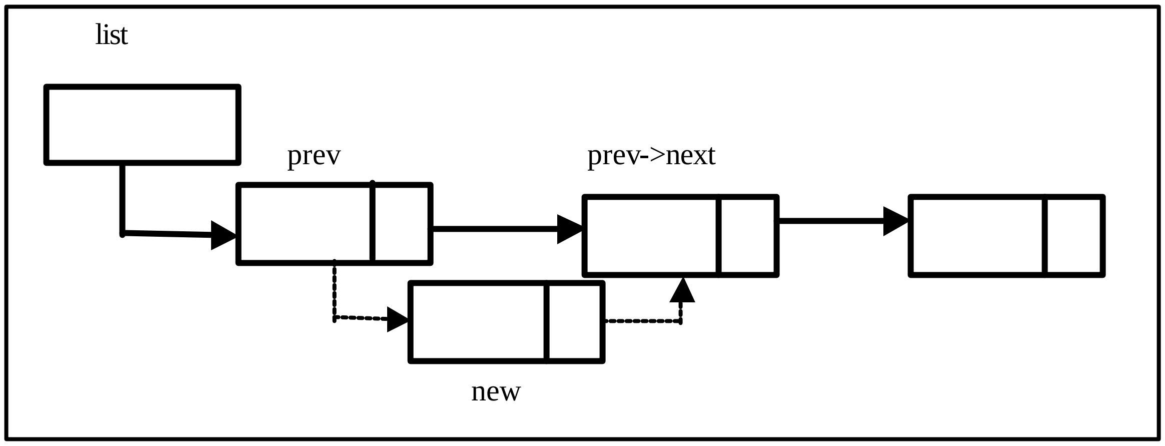


Fig 8.8: Insertion of a node at the middle of the Linked list

The new node has to be inserted after the node pointed by *prev* node then previous node should point to *new* and *new* will point to next of *prev*.

```
new->next = prev->next;
prev->next = new;
```

8.1.4 To add to the end of the list

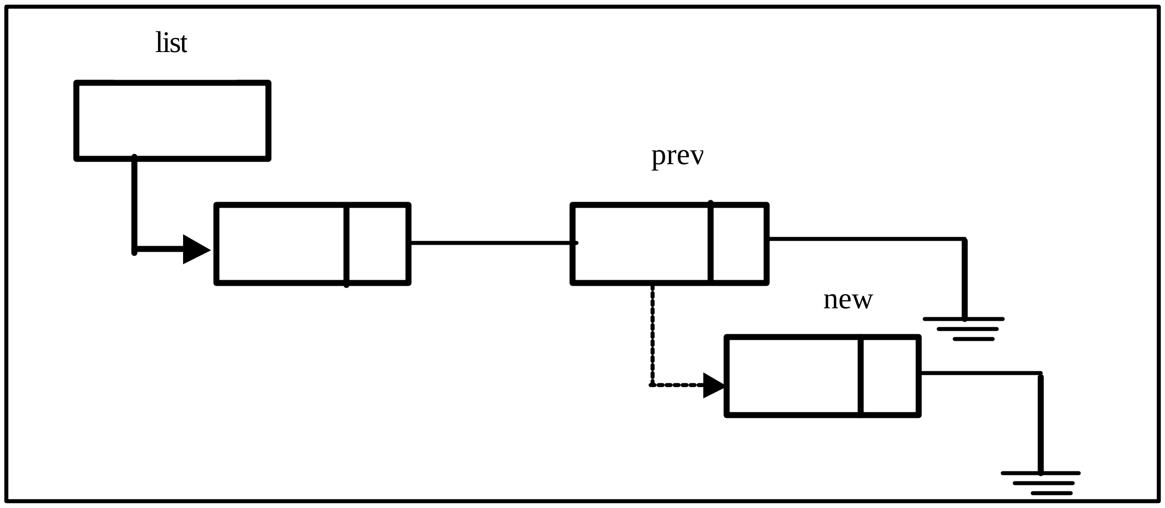


Fig 8.9: Insertion of a node at the end of the Linked list

The last node should point to new node and new node should point null to become last node.

```
new->next = null;
```

which is equivalent to

```
    new->next = prev->next;
```

then

```
    prev->next = new;
```

8.1.5 Insertion of new node in the list

Note that in a singly linked list, nodes can only be inserted or deleted after a given node. So a general-purpose search function will be needed which will return the address of the node after which the new node is to be inserted/deleted. For the moment, let us assume a search function that takes the empid as a parameter and returns the address of the last node whose empid value is less than the given node. We can now code the insert function as

```
int insert(struct node *new)
{
     struct node *prev;
     int flag;
     if (list==NULL)  /* list empty */
     {
          list=new;
          return 0;
     }
     prev=search(new->empid,&flag);
     if(flag==1)  /* duplicate empid */
          return -1;
     if(prev==NULL)  /* insert at beginning */
     {
          new->next=list;
          list=new;
     }
     else  /* insert at middle or end */
     {
          new->next=prev->next;
          prev->next=new;
     }
     return 0;
}
```

Fig 8.10: Code for insertion of a node in the Linked list

8.1.6 Searching a node in the list

The search function traverses the list to find the first node whose empid value is greater than or equal to the empid value received as a parameter. Flag is set to 1 if a node is found with the same empid value else it is set to 0. Thus the search function can also be used to find a node with a given empid value. The search function can thus be used to display the details of a given node (identified by empid). We display the details of the next node to that returned by search if flag is set to 1. During insert however, if a node with the same empid is found the insert operation fails.

To code the search function:

```
struct node * search(int id,int *flag)
{
     struct node *prev,*cur;
     *flag=0;
     if (list==NULL)  /* list empty */
          return NULL;
     for(prev=NULL,cur=list;((cur) && ((cur->empid) < id));
                                   prev=cur,cur=cur->next);
     if( (cur) && ( cur->empid==id))
    /* node with given empid exists */
          *flag=1;
     else
          *flag=0;
     return prev;
}
```

Fig 8.11: Code for creation of a node in the Linked list

8.1.7 Displaying the linked list

To display all the nodes in a linked list, we need to traverse the list sequentially and print the details

```
void displayall()
{
     struct node *cur;
     system("clear");
     if(list==NULL)
     {
          printf("list is empty\n");
           return;
     }
     printf("empid, name, salary\n");
     for(cur=list;cur;cur=cur->next)      {
         printf("%4d%-22s%8.2f\n",cur->empid,cur->name,
                                       cur->salary);
}
```

Fig 8.12: Code for displaying all nodes in the Linked list

8.1.8 Deletion of existing node from the linked list

To modify a node search for the node and accept the details again. To delete a node the links have to be reformulated to exclude the deleted node. The memory allocated for the deleted node must also be freed.

The node to be freed

- may not be existing in the list
- may be the first node (in which case the list pointer must be reinitialised)
- may be any other node or the list may be empty.

- To delete the first node

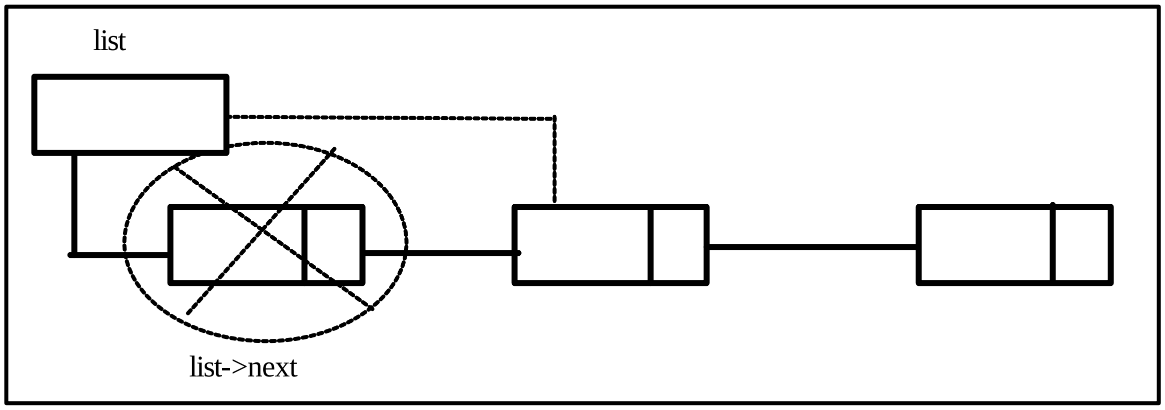


Fig 8.13: Deletion of the first node from the Linked list

```
    temp=list;  /* where temp is defined as struct node */
    list = list->next;
    free(temp);
```

If the first node is also the last node in the list , list automatically becomes NULL.

- To delete other nodes

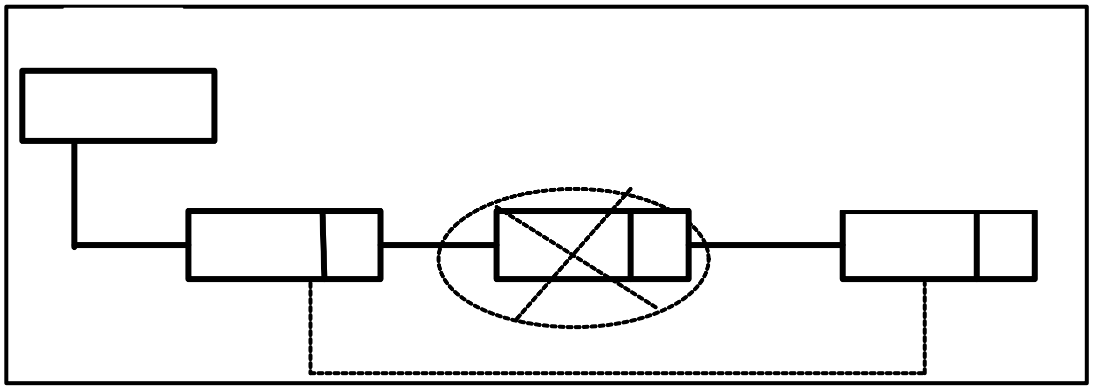


Fig 8.14: Deletion of the middle node from the Linked list

```
    temp=prev->next;  /* prev is the node returned by search */
    prev->next=temp->next;
    free(temp);
```

- So the delete function can be coded as

```
int delete(int id)
{
    struct node *prev,*temp;
    int flag;
    if (list==NULL)  /* list empty */
          return -1;
    prev=search(id,&flag);
    if(flag==0)  /* empid not found */
          return -1;
    if(prev==NULL)
    /*node to delete is first node(as flag is 1) */
    {
          temp=list;
           list=list->next;
           free(temp);
     }
     else
     {
          temp=prev->next;
          prev->next=temp->next;
          free(temp);
     }
     return 0;
 }
```

Fig 8.15: Deletion of the first node from the Linked list

8.2 Complete Program for the operations of Linked list

Let us put the above modules into a complete program that will insert, delete or display information from a singly linked list

```
#include<stdio.h>
#include<alloc.h>
struct node
{
     int empid;
     char name[20];
     float salary;
     struct node *next;
};
struct node *list; /* global pointer to beginning of list */
struct node * getnode() /*creates a node and accepts data */
{
     struct node *temp;
     temp=(struct node *)malloc(sizeof(struct node));
     printf("enter the empid:");
     scanf("%d",&temp->empid);
     fflush(stdin);
     printf("enter the name:");
     scanf("%s",temp->name);
     fflush(stdin);
     printf("enter salary:");
     scanf("%f",&temp->salary);
     fflush(stdin);
     temp->next=NULL;
     return temp;
}
/* search returns address of previous node; current node is */
struct node * search(int id,int *flag)
{
    struct node *prev,*cur;
    *flag=0;
    if (list==NULL)  /* list empty */
          return NULL;
    for(prev=NULL,cur=list;((cur) && ((cur->empid) < id));
                               prev=cur,cur=cur->next);
    if((cur)&&(cur->empid==id))
   /* node with given empid exists */
          *flag=1;
    else
          *flag=0;
    return prev;
}
int insert(struct node *new)
{
    struct node *prev;
    int flag;
    if (list==NULL)  /* list empty  */
    {
          list=new;
          return 0;
    }
    prev = search(new->empid,&flag);
    if(flag == 1)  /* duplicate empid */
          return -1;
    if(prev==NULL)  /*insert at beginning */
    {
          new->next=list;
          list=new;
    }


    else  /* insert at middle or end */
    {
          new->next=prev->next;
          prev->next=new;
    }
    return 0;
}
void displayall()
{
     struct node *cur;
     system("clear");
     if(list==NULL)
     {
          printf("list is empty\n");
          return;
     }
     printf("empid name salary\n");
     for(cur=list;cur;cur=cur->next)
           printf("%4d%-22s%8.2f\n",cur->empid,cur->name,
                                    cur->salary);
}
int delete(int id)
{
     struct node *prev,*temp;
     int flag;
     if (list==NULL)   /* list empty */
          return -1;
     prev=search(id,&flag);
     if(flag==0)  /* empid not found  */
          return -1;
     if(prev==NULL)
     /* node to delete is first node (as flag is 1) */
     {
          temp=list;
          list=list->next;
          free(temp);
     }
     else
     {
          temp = prev->next;
          prev->next = temp->next;
          free(temp);
     }
     return 0;
}




void main(void)
{
        struct node *new;
        int choice=1,id;
        list=NULL;

        do
        {
                printf("\n\n\n\n\n\t\t\t\tMenu\n\n");
                printf("\t\t\t\t1.Insert\n");
                printf("\t\t\t\t2.Delete\n");
                printf("\t\t\t\t3.Display list\n");
                printf("\t\t\t\t0.Exit\n");
                printf("\n\n\t\t\t\t......enter choice:");
                scanf("%d",&choice);
                fflush(stdin);
                system("clear");
                switch(choice)
                {
                        case 1 : new=getnode();
                                 if(insert(new)== -1)
                                         printf("error:cannot insert\n");
                                 else
                                         printf("node inserted\n");
                                 getchar();
                                 break;
                        case 2 :
                                 printf("enter the empid to delete:") ;
                                 scanf("%d",&id);
                                 fflush(stdin);
                                 if(delete(id)==-1)
                                        printf("deletion failed\n");
                                 else
                                         printf("node deleted\n");
                                 getchar();
                                 break;
                        case 3 :
                                 displayall();
                                 getchar();
                                 break;
                        case 0 :
                                 exit();
                }
         }while(choice !=0);
}
```

Fig 8.16: Complete code for Linked list operations

8.3 Doubly Linked List

A doubly linked list is very similar to the normal linked list, except that it has two links: One to the next node and the other to the previous node. So, the structure now has two additional member pointers for each link. The advantage of a doubly linked list is that you can traverse in both directions using the doubly linked list. The structure definition now looks as follows:

E..g :

```
struct node
{
                data_type info;
                  struct node * next ;
                  struct node * prev ;
};
```

The linked program generally remains the same, except that now you need to handle the additional link. So when a new node is to be added to an existing linked list, you need to assign to the pointer prev, the address of the previous node.

8.4 Stacks

A stack is an ordered collection of items into which new items may be inserted and from which elements may be deleted at one end, called **top of the stack**.

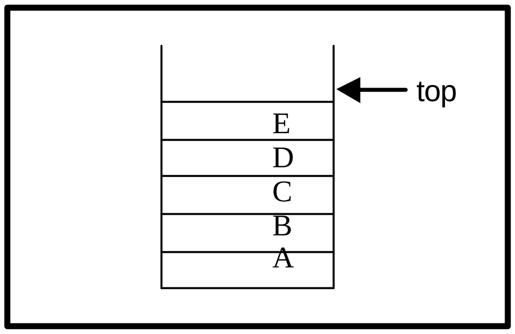


Fig 8.17: A stack containing stack items.

The stack provides for insertion and deletion of items, so the stack is a dynamic, constantly changing object. The definition specifies that a single end of the stack is designated as the stack top. New items may be put on top of the stack, or items, which are at the top of the stack, may be removed. The stack implements the concept of **LIFO** (Last In First Out).

There are two operations that can be performed on a stack. When an item is added to the stacked, it is pushed onto the stack, and when an item is removed, it is popped from the stack. Given a stack s, and an item I, performing the operation push(s, i) adds the item i to the top of the stack s. Similarly, the operation pop(s) removes the top element and returns it as a function value.

Thus the assignment operation

```
    i = pop(s);
```

removes the element at the top of s and assigns its value to i.

8.5 Queues

A queue is an ordered collection of items from which items may be deleted at one end (called the front of the queue) and into which items may be inserted at the other end (called the rear of the queue). The queue implements the concept of FIFO (First In First Out).

Three operations can be applied to a queue. The operation insert(q,x) inserts item x at the rear of the queue q. The operation x = remove(q) deletes the front element from the queue q and sets x to its contents. The third operation, empty(q), returns false or true depending on whether or not the queue contains any elements.

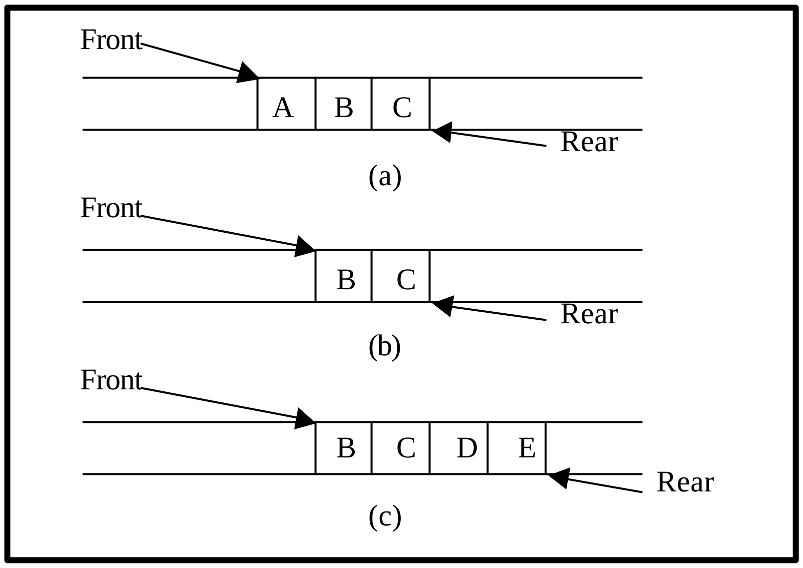


Fig 8.18: A queue

The queue in Fig 8.3 can be obtained by the following sequence of operations. We assume that the queue is initially empty.

insert(q, A);	
insert(q, B);	Fig 8.2 (a)
insert(q, C);	Fig 8.2 (b)
x = remove(q);	
insert(q, D);	
insert(q, E);	Fig 8.2 (c)

9 File Handling

In all the C programs considered so far, we have assumed that the input data was read from standard input and the output was displayed on the standard output. These programs are adequate if the volume of data involved is not large. However many business-related applications require that a large amount of data be read, processed and saved for later use. In such a case, the data is stored on storage device, usually a disk.

9.1 Introduction

So far we have dealt with various input/output functions like printf(), scanf(), getchar() etc.

Now let us pay attention to the functions related to **disk I/O**.

These functions can be broadly divided into two categories.

- High-level file I/O functions also called as standard I/O or stream I/O functions.
- Low-level file I/O functions also called as system I/O functions.

The low-level disk I/O functions are more closely related to the computer's operating system than the high-level disk I/O functions. This chapter is concerned only with high-level disk I/O functions.

As you can see the high-level file I/O functions are further categorised into text and binary but this chapter will deal only with text mode. We will directly jump to functions, which perform file I/O in high-level, unformatted text mode.

9.2 Unformatted high-level disk I/O functions

9.2.1 Opening a file with fopen() function

Before we can write information to a file on a disk or read it, we must open the file. Opening a file establishes a link between the program and the operating system. The link between our program and the operating system is a structure called **FILE,** which has been defined in the header file "**stdio.h**". The FILE structure contains information about the file being used, such as current size, location in memory etc. So a file pointer is a pointer variable of the type FILE.

It is declared as

```
     FILE *fp;
```

where fp is a pointer of FILE type.

The general format of **fopen()** is

```
     FILE *fp;
     fp=fopen(“file_name”, “type”);
```

where,

File_name	character string that contains the name of the file to be opened.
Type	a character string having one of the following modes in which we can open a file.

File Type/ File mode	Meaning
r	Opens an existing file for reading only. If the file does not exist, it returns NULL.
w	Opens a new file for writing only. If the file exists, then it’s contents are overwritten. Returns NULL, if unable to open file.
a	Opens an existing file for appending. If the file does not exist then a new file is created. Returns NULL, if unable to open file.
r+	Opens an existing file for reading, writing and modifying the existing contents of the file. Returns NULL, if unable to open file.
w+	Opens a new file for both reading and writing. If the file already exists then it’s contents are destroyed. Returns NULL, if unable to open file.
a+	Opens an existing file for reading and appending. If the file does not exist, then a new file is created.

9.2.2 Closing a file with fclose() function

When we have finished working with the file, we need to close the file. This is done using the function **fclose()** through the statement

```
     fclose(fp);
```

fclose closes the file to which the file pointer **fp** points to. It also writes the buffered data in the file before close is performed.

9.3 Character Input/Output in files

The **getc()** and **putc()** functions can be used for character I/O. They are used to read and write a single character from/to a file.

9.3.1 The function getc()

The function **getc()** is used to read characters from a file opened in read mode by **fopen()**. The general format is:

```
getc(fp);
```

getc() gets the next character from the input file to which the file pointer **fp** points to. The function **getc()** will return an end-of-file EOF marker when the end of the file has been reached or if it encounters an error.

9.3.2 The function putc()

The general format of **putc()** is:

```
putc(c,fp);
```

where **putc()** function is used to write characters to a disk file that can be opened using **fopen()** in **"w"** mode. **fp** is the file pointer and c is the character to be written to the file.

On success the function putc() will return the value that it has written to the file, otherwise it returns EOF.

Now we have seen functions fopen(), fclose(), getc(), putc() etc. As a practical use of the above functions we can copy the contents of one file into another.

```
/* This program takes the contents of a text file and
   copies into another text file, character by character */
# include <stdio.h>
void main(void)
{
       FILE *fs,*ft;
       char ch;
       fs=fopen("pr1.c","r");     /* open file in read mode */
       if(fs==NULL)
       {
              puts("Cannot open source file");
              exit(0);
       }
       ft=fopen("pr2.c","w");     /* open file in write mode */
       if(ft==NULL)
       {
              puts("Cannot open target file");
              fclose(fs);
              exit(0);
       }



       while(1)
```

```
     {
          ch=getc(fs);
           if(ch==EOF)
                break;
                putc(ch,ft);
     }
      fclose(fs);
     fclose(ft);
}
```

Fig 9.1: Program to copy one file to another file

9.4 Command Line Arguments (Using argc and argv parameters)

The **main()** function takes two arguments called **argv** and **argc**.

The general format is

```
     main(argc,argv)
     int argc;
     char *argv[ ];
```

The integer **argc (argument count)** contains the number of arguments in the command line, including the command name.

argv (argument vector) is an array which contains addresses of each arguments.

When there is a need to pass information into a program while you are running it, then the information can be passed into the main() function through the built in arguments argc and argv.

- Consider an example that will print your name on the screen if you type it directly after the program name.

```
/* Program that explains argc and argv */
# include <stdio.h>
main(argc,argv)
int argc;
char *argv[ ];
{
if (argc==1)
     {
          printf(“ You forgot to type your name \n”);
          exit();
     }
     printf(“Hello %s”, argv[1]);
}
Output:        % Hello Message
```

```
                    You forgot to type your name
                     % Hello Message Boston’s
                    Hello Boston’s
```

Fig 9.2: Sample Code using command line arguments

```
/* This program copies one file to another using
   command line arguments */
#include <stdio.h>
main(int argc, char *argv[ ])
{
     char ch;
     FILE *fp1, *fp2;
     if ((fp1=fopen(argv[1],”r”))==NULL)
     {
           printf(“Cannot open file %s \n”,argv[1]);
           exit();
     }
     if ((fp2=fopen(argv[2],”w”))==NULL)
     {
           printf(“Cannot open file %s \n”,argv[2]);
           exit();
     }
     while((ch=getc(fp1))!=EOF)
   /* read a character from one file */
           putc(ch,fp2);
     fclose(fp1);
     fclose(fp2);
}
Output:
          mcopy pr1.c pr2.c
 (pr1.c will get copied to pr2.c)
```

Fig 9.3: Program to copy one file to another using command line arguments

9.5 String (line) Input/Output in Files

We have seen putc() and getc() functions as character I/O in files. But reading or writing strings of characters from and to files is as easy as reading and writing individual characters.

The functions fgets() and fputs() can be used for string I/O..

9.5.1 Library Call fgets()

The routine fgets() is used to read a line of text from a file.

The general format is:

```
     char *fgets( char *s, int n, FILE *fp);
```

The function fgets() reads character from the stream fp into the character array 's' until a newline character is read, or end-of-file is reached, or n-1 characters have been read. It then appends the terminating null character after the last character read and returns 's'. If end-of-file occurs before reading any character or an error occurs during input fgets() returns NULL.

9.5.2 Library Call fputs()

The routine **fputs()** is used to write a line of text from a file.

The general format is:

```
     int fputs(const char *s, FILE *fp);
```

The function **fputs()** writes to the stream **fp** except the terminating null character of string **s**. It returns EOF if an error occurs during output otherwise it returns a nonnegative value.

The program given below writes strings to a file using the function **fputs()**.

```
/* Receives strings from keyboard and writes them to file. */
#include<stdio.h>
void main(void)
{
     FILE *fp;
     char s[80];
     fp=fopen("test.txt","w");
     if(fp==NULL)
     {
          puts("Cannot open file");
          exit(0);
     }
     printf("Enter few lines of text \n ");
     while(strlen(gets(s)) >0)
     {
          fputs(s,fp);
          fputs("\n",fp);
     }
     fclose(fp);
}
```

Fig 9.4: Program to accept the text and write it in the file

In this program we have set up a character array to receive the string, the fputs() function then writes the contents of the array to the disk. Since the fputs() function does not automatically add a newline character we have done this explicitly.

```
/* Program to read strings from the file and displays
   them on the screen */
#include<stdio.h>
void main(void)
{
     FILE *fp;
     char s[80];
     fp=fopen("test.txt","r");
     if(fp==NULL)
     {
          puts("Cannot open file");
          exit(0);
     }
     while(fgets(s,79,fp) !=NULL)
     printf("%s",s);
     fclose(fp);
}
```

Fig 9.5: Program to read strings from the file and display them on the screen

The function fgets() takes three arguments. The first is the address where the string is stored and second is the maximum length of the string. This argument prevents fgets() from reading it too long a string and overflowing the array. The third argument is the pointer to the structure FILE.

9.6 Formatted high-level disk I/O functions

C language provides two functions **fprintf()** and **fscanf()** which provides **formatted Input/Output** to the files. The functions **fprintf()** and **fscanf()** are used in the same manner as **scanf()** and **printf()** and require a file pointer as their first argument.

9.6.1 The Library Function fprintf()

The general format is:

```
     int fprintf(fp,format,s)
     FILE *fp;
     char *format;
```

The call **fprintf()** places output on the named output to which the file pointer **fp** points,

s represents the arguments whose values are printed.

format is the format specifier string. The format conventions of printf() work exactly same with **fprintf()**.

9.6.2 The function fscanf()

The function **fscanf()** reads from the file to which the file pointer points.

The general format is

```
     int fscanf(fp,format,s)
     FILE *fp;
     char *format;
```

The function **fscanf()** reads from the file to which the file pointer **fp** is pointing. **fscanf()** returns the number of values read.

format is the format specifier string.

s represents the arguments (or buffer area) where data is stored after the read operation.

The following program shows the use of **fprintf()** and **fscanf()**.

```
/* This program is taking input from keyboard and writing
   it to the file and then printing on the screen */
# include<stdio.h>
void main(void)
{
     FILE *fp;
     char s[80];
     if ((fp=fopen(“test.txt”,”w”))==NULL)
     {
          printf(“Cannot open the file \n”);
          exit(0);
     }
     fscanf(stdin,”%[^\n]”,s);/* reading from the keyboard  */
     fprintf(fp,”%s”,s);  /* writing to the file */
     fclose(fp);
     if((fp=fopen(“test.txt”,”r”))==NULL)
      {
          printf(“Cannot open the file \n”);
          exit();
     }
     fscanf(fp,”%[^\n]”,s);     /* reading from the file */
     fprintf(stdout,”%s”,s);    /* printing on the screen */
}
```

Fig 9.6: Program to explain fscanf() and fprintf()

9.7 Direct Input/Output

Direct input/output functions provide facilities to read and write a certain number of data items of specified size. The functions are **fread()** and **fwrite()**.

9.7.1 Library Call fread()

The general format is:

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

The function **fread()** reads into array **ptr** upto **nitems** data items of size **size** from the stream **fp** and returns the number of items read.

If an error is encountered **fread()** returns EOF otherwise returns the number of items read.

The file position indicator is advanced by the number of characters successfully read. For example, assuming 4-byte integers, the statement

```
      rchar=fread(buf,sizeof(int),20,input);
```

reads 80 characters from **input** into the array **buf** and assigns 80 to **rchar**, unless an error or end-of-file occurs.

9.7.2 Library Call fwrite()

The general format is

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

The function **fwrite()** appends at the most **nitems** item of data of size **size** in the file to which the file pointer **fp** points to, from the array to which the pointer **ptr** points to.

The function returns the number of items written on success, otherwise EOF if an error is encountered.

The file position indicator is advanced by the number of characters successfully written. For example,

```
     wchar=fwrite(buf,sizeof(char),80,output);
```

writes 80 characters from the array **buf** to **output**, advances the file position indicator for **output** by 80 bytes. and assigns 80 to **wchar** unless an error or end-of-file occurs.

One of the most useful applications of **fread()** and **fwrite()** involves the reading and writing of user defined data types, especially structures.

A simple mailing_list program using fread() and fwrite() is given below. The functions load() and save() perform the loading and saving operations of the database.

```
# include <stdio.h>
# include <string.h>
# define SIZE 100
void int_list(void);
void enter();
void display(void);
void save(void);
void load(void);
void menu();
int i,t;
struct list_type
{
     char name[20];
     char street[2];
     char city[10];
     char state[3];
     char pin[10];
}list[SIZE];
void main(void)
{
     char choice;
     printf(“Enter choice (e/d/s/l/q)”);
     scanf(“%c”,&choice);
     for(;;)
     {
          switch(choice)
          {
               case 'e':
                    enter();
                    break;
               case 'd':
                    display();
                    break;
               case 's':
                    save();
                    break;
               case 'l':
                    load();
                    break;
               case 'q':
                    exit();
                    break;
          }
     }
}
void int_list(void) /* initialize the list */
{
     register int t;
     for(t=0;t<100;t++)
     strcpy(list[t].name,"\0");/*zero length signifies empty */
}
```

```
void enter(void)
{
     register int i;
     for(i=0;i<SIZE;i++)
          if(!*list[i].name)
               break;
     if(i==SIZE)
     {
          printf("list full\n");
          return;
     }
     printf("name");
     gets(list[i].name);
     printf("Street:");
     gets(list[i].street);
     printf("State:");
     gets(list[i].state);
     printf("Pin:");
     gets(list[i].pin);
}
/* display the list */
void display(void)
{
     register int t;
     for(t=0;t<SIZE;t++)
     printf("%s\n",list[t].name); /* printf all the
                                  information the same way */
}
/* save the list */
void save(void)
{
     FILE *fp;
     if((fp=fopen("maillist","w+"))==NULL)
     {
          printf("Cannot open file \n");
          return;
     }
}
/* load the file */
void load(void)
{
     FILE *fp;
     register int i;
     if((fp=fopen("maillist","r+"))==NULL)
     {
          printf("Cannot open file \n");
          return;
     }
}


void menu(void)
```

```
{
    /* print choices and return appropriate choice */
}
```

9.8 Error Handling Functions

The error handling functions provide facilities to test whether EOF returned by a function indicates an end-of-file or an error.

9.8.1 The function feof()

Because the buffered file system is designed to handle both text and binary files, it is necessary that there should be some way other than the return value of getc() to determine that the end-of-file mark is also a valid integer value that could occur in a binary file.

The general format is

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

Where **fp** is a valid file pointer.

The function **feof()**returns true (non-zero) if the end of the file pointed to by **fp** has been reached otherwise it returns zero.

9.8.2 The function ferror()

The general format is

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

The function **ferror()** returns a non-zero value if the error indicator is set for the stream **fp** and 0 otherwise.

9.8.3 The function perror()

The general format is

```
    void perror(const char *s);
```

The function **perror()** writes to the standard error output **stderr** the string **s** followed by a colon and a space and then an implementation- defined error message corresponding to the integer in errno, terminated by a newline character.

The program given below receives records from keyboard, writes them to a file and also display them on the screen.

```
#include<stdio.h>
void main(void)
{
     FILE *fp,*fpr;
     char another='Y';
     struct emp
     {
          char name[40];
          int age;
          float bs;
     };
     struct emp e;
     fp=fopen("emp.dat","w");
     if(fp==NULL)
     {
          puts("Cannot open file");
          exit(0);
     }
     while(another=='Y')
     {
          printf("\n enter name , age basic salary\n");
          scanf("%s%d%f",&e.name,&e.age,&e.bs);
          fwrite(&e,sizeof(e),1,fp);
          printf("Add another record (Y/N)");
          fflush(stdin);
          another=getchar();
     }
     fclose(fp);
     fpr=fopen("emp.dat","r");
     if(fpr==NULL)
     {
          puts("Cannot open file");
          exit(0);
     }
     while(fread(&e,sizeof(e),1,fpr)==1)
          printf("%s %d %f \n",e.name,e.age,e.bs);
     fclose(fpr);
}
```

Fig 9.7: Program to accept, write and display the record

9.9 File Positioning

A file may be accessed sequentially or randomly. In a sequential access, all the preceding data is accessed before accessing a specific portion of a file. Random access permits direct access to a specific portion of a file. **fseek()**, **ftell()** and **rewind()** are the functions used in random access of a file.

9.9.1 The function fseek()

The general format is

```
int fseek(FILE *fp,long offset, int ptrname);
```

fseek() sets the position of the next input or output operation in the file to which the file pointer **fp** points to. The new position is at the signed distance offset bytes from the beginning , from the current position or from the end of the file depending upon the value of the **ptrname**. The third argument can be either **SEEK_CUR**, **SEEK_END** or **SEEK_SET**.

The function returns 0 when successful otherwise a nonzero value.

- SEEK_END means move the pointer from the end of the file.
- SEEK_CUR means move the pointer from the current position.
- SEEK_SET means move the pointer from the beginning of the file.

Here are some examples of calls to **fseek()** and their effect on the file position indicator.

fseek(fp,n,SEEK_CUR)	sets cursor ahead from current position by n bytes
fseek(fp,-n,SEEK_CUR)	sets cursor back from current position by n bytes
fseek(fp,0,SEEK_END)	sets cursor to the end of the file
fseek(fp,o,SEEK_SET)	sets cursor to the beginning of the file

9.9.2 The Function ftell()

The general format is

```
long ftell(FILE *fp);
```

The function **ftell()** returns the current value of the file position indicator associated with **fp**.

9.9.3 The function rewind()

The general format is

```
void rewind(FILE *fp);
```

The function **rewind()** resets the current value of the file position indicator associated with **fp** to the beginning of the file.

The call

```
rewind(fp);
```

has the same effect as

```
void fseek( fp,0,SEEK_SET);
```

The use of **rewind()** allows a program to read through a file more than once without having to close and open the file again.

10 Miscellaneous

10.1 The C Preprocessor

The **C preprocessor** is exactly what its name implies. It is a collection of special statements, called **directives**. It can be an independent program or its functionality may be embedded in the compiler.

10.2 Introduction to Preprocessor

It is a program that processes the source text of a C program before the program is passed to the compiler.

It has four major functions

- Macro replacement
- Conditional compilation
- File inclusion
- Error generation

The C preprocessor offers several features called **preprocessor directives**. Each of these preprocessor directives begin with a # symbol. We will learn the following preprocessor directives here

- #define directive
- #include directive
- #undef directive
- #error directive
- Conditional compilation directives.

10.3 Macro substitution

This is a very useful feature. The preprocessor replaces every occurrence of a simple **macro** in the program text by a copy of the body of the macro. The body of the macro may itself contain other macros. It is achieved using the **#define** directive.

The general syntax is

```
     #define macro-name  sequence-of-tokens
```

The above declaration associates with the macro-name whatever sequence-of-tokens appears from the first blank after the macro-name to the end of the file.

It is a convention to write all macros in capitals to identify them as symbolic constants.

```
/* This program explains macro substitution using #define */
#include <stdio.h>
/* Associates macro name GREET with value “hello” */
#define GREET “hello”
/* Associates macro name NAME with values Ash wini */
#define NAME “Ash”  “  “  “wini”
     /* Associates macro name MAX with value 10 */
void main(void)
{
     printf(“\n%s\t”,NAME);
     printf(“%s”,GREET);
}
Output:
     Ash wini        hello
```

Fig 10.1: Program using macros

The program given below shows **#define** directive used to define operators.

```
#include <stdio.h>
#define && AND
#define || OR
void main(void)
{
     int f=1,x=4,y=90;

     if((f < 5) AND (x<=20 OR y <=45))
          printf(“Your pc will work fine.....”);
     else
          printf(“In front of the maintenance man......”);
}
```

Fig 10.2: Program to explain #define directive.

10.3.1 Macros with arguments

The macros that we have used so far are called **simple macros**. Macros can have arguments. This is also called as **parameterized macros**.

```
#include <stdio.h>
#define AREA( r )  (3.14*r*r)
void main(void)
{
     float radius;
     printf(“Enter the radius \t”);
     scanf(“%f”,&radius);
     printf(“\nArea of the circle is %f”,AREA(radius));
}
```

Fig 10.3: Program to explain macros with arguments

10.3.2 Nesting Of Macros

We can also use one macro in the definition of another macro. That is macro definitions may be **nested**. For instance, consider the following macro definitions

```
/* This program shows use of nesting of macros */
#include <stdio.h>
#define  SQUARE(x)   (x*x)
#define CUBE(x)(SQUARE(x) * x)
void main(void)
{
     int no;
     printf("Enter the number  ");
     scanf("%d",&no);
     printf("\nSquare of a number is %d",SQUARE(no));
     printf("\nCube of a number is %d",CUBE(no));
}
```

Fig 10.4: Program to explain the use of nesting of macros

10.4 Undefining a Macro

A defined macro can be undefined, using the statement

```
     #undef identifier
```

This is useful when we want to restrict the definition only to a particular part of the program.

In the above program macro-name *SQUARE* and *CUBE* can be undefined using the statement

```
     #undef SQUARE
     #undef CUBE
```

10.5 File Inclusion

This preprocessor directive causes one file to be included in another. This feature is used in two cases

- If we have a very large program, it is good programming practice to keep different sections in separate file. These files are included at the beginning of main program file.
- Many a times we need some functions or some macro definitions almost in all programs that we write. In such a case, commonly needed functions and macro definitions can be stored in a file and that file can be included wherever necessary.

There exist two ways to write **#include** statements. These are

```
     #include <filename>
     #include "filename"
```

The meaning of each form is given below.

#include <program1.h>	This command would look for the file program1.h in the default include directory.
#include “program1.h”	This command would look for the file program1.h in the default include directory as well as current directory.

For example if we have the following three files

function.c	contains some functions
proto.h	contains prototypes of functions
test.c	contains test functions

We can make use of a definition or function contained in any of these files by including them in the program as shown below.

```
#include <stdio.h>
#include “function.c”
#include “proto.h”
#include “test.c”
#define M   50

void main(void)
{
        .......     /* Here the code in the above three files */
        ........    /* is added to the main code */
        ........    /* and the file is compiled */
}
```

10.6 Conditional Compilation

Conditional compilation allows selective inclusion of lines of source text on the basis of a computed condition. Conditional compilation is performed using the preprocessor directives

- #ifdef
- #ifndef
- #elif
- #else
- #endif

We can have the compiler skip over, part of a source code by inserting the preprocessing commands **#ifdef** and **#endif**.

The general form is

```
    #ifdef macroname
        statement 1;
        statement 2 ;
    #else
        statement 3 ;
    #endif
```

If macro-name has been #defined, the block of code (statement 1 and statement 2) will be processed otherwise else statement 3 will be executed.

```
#include <stdio.h>
#ifndef PI
#define PI 3.14
#endif
void main(void)
{
     float area,rad;
     printf(“Enter the radius :- “);
     scanf(“%f”,&rad);
      area=PI*rad*rad;
      printf(“\n The area of the circle is %.2f “,area);
}
```

#ifndef directive is the opposite of the **#ifdef** directive. The **#ifdef** includes the code if the identifier is defined before but **#ifndef** includes it if the identifier has not been defined before.

#elif statemenrt is analogous to the else is construct. Using this, a switch case construct can be constructed for preprocessing purpose.

10.7 Error Generation

Using **#error** directive, we can display the error message on occurrence of error.

The directive of the form

```
#error token_sequence
```

causes the implementation to produce a diagnostic message containing the **token_sequence**.

For example

```
     #ifndef PI
          #error  “PI NOT DEFINED”
     #endif
```

If *PI* is not defined preprocessor will print the error message *“PI NOT DEFINED”* and compilation will not check further.

10.8 User Defined Data Types

For the purpose of effective documentation of the program, sometimes user requires to define a new data type of its own. It helps to increase clarity of the program. It thereby provides greater ease of maintenance of the program, which is an important part of software management.

10.8.1 typedef Statement

C provides a facility called **type definition**, which allows users to define new data types that are equivalent to existing data types. Once a user-defined data type has been established, then new variables, arrays, structures and so on can be declared in terms of this new data type.

In general terms, a new data type is defined as

```
    typedef type new-type;
```

where,

type	An existing data type(either a standard data type or a previous user-defined data type).
new-type	The new user-defined data type.

It should be understood, however, that the new data type will be new in the name only. In reality, this new data type will not be fundamentally different from one of the standard data types.

Here is a simple declaration involving the use of typedef.

```
    typedef int age;
```

In this declaration age is a user-defined data type equivalent to type int. Hence, variable declaration

```
    age male,female;
```

is equivalent to writing

```
    int male,female;
```

In other words, male and female are regarded as variables of type age, though they are actually integer type variables.

Similarly, the declaration

```
    typedef float height[100];
    height boy,girl;
```

define height as a 100-element, floating-point array type. Hence, boy and girl are 100-element, floating point arrays.

Another way to express the above declaration is

```
    typedef float height;
    height boy[100],girl[100];
```

though the former declaration is somewhat simpler.

The typedef feature is particularly convenient when defining structure, since it eliminates the need to repeatedly write struct tag whenever a structure is referenced. As a result, the structure can be referenced more concisely. In addition, the name given to a user-defined structure type often suggests the purpose of the structure within the program.

In general terms, a user-defined structure type can be written as

```
typedef struct
{
     member 1;
     member 2;
     ...
     member n;
}new-type;
```

where, new-type is the user-defined structure type. Structure variables can then be defined in terms of the new data type.

```
/* Example of typedef statement */
typedef struct
{
     int acct_no;
     char acct_type;
     char name[20];
     float balance;
}record;
record oldcustomer,newcustomer;
```

The first declaration defines record as a user-defined data type. The second declaration defines *oldcustomer* and *newcustomer* as structure variables of type *record*.

The typedef feature can be used repeatedly, to define one data type in terms of other user-defined data type.

Following are some examples of structure declarations.

```
typedef struct
{
     int month,day,year;
}date;
typedef struct
{
int acct_no;
      char acct_type;
     char name[20];
     float balance;
     date lastpayment;
}record;
record customer[50];
```

In above example, *date* and *record* are user-defined structure types, and *customer* is a 50-element array whose elements are structures of type record.(Recall that *date* was a tag rather than actual data type in example). The individual members within the ith element of customer can be written as *customer[i].acct_no*, *customer[i].name*, *customer[i].lastpayment.month*, and so on. as before.

There are, of course, variations on this theme. Thus, an alternate declaration can be written as,

```
typedef struct
{
     int month,day,year;
}date;
typedef struct
{
     int acct_no;
     char acct_type;
     char name[20];
     float balance;
     date lastpayment;
}record[50];
record customer;
or simply
typedef struct
{
     int month,day,year;
}date;
struct
{
     int acct_no;
     char acct_type;
     char name[20];
     float balance;
     date lastpayment;
}customer[50];
```

All three sets of declarations are equivalent.

10.8.2 Enumerations

Enumeration types provide the facility to specify the possible values of a variable by meaningful symbolic means. This can help in making the program more readable.

It is a data type similar to a structure. Its members are constants that are written as identifiers, though they have signed integer values. These constants represent values that can be assigned to corresponding enumeration variables.

The general format for defining an enumerated data type is

```
     enum tag {member 1,member 2,...,member n};
```

Where,

Enum	Required keyword.
tag	An identifier that names the enumeration type.
member1, member2	Identifiers called enumeration constants or enumerators.

For example, the declaration

```
      enum colour {black,white,pink,red,green,yellow,blue};
```

defines an enumeration type colour whose values are black, white, pink, red, green, yellow and blue.

An enumeration type is implemented by associating the integer value with the enumeration constant. Thus, the value 0 is associated with black, 1 with white, 2 with pink, 3 with red, 4 with green, 5 with yellow and 6 with blue.

These enumeration assignments can be overridden by initialising the enumerators to different integer values. Subsequent enumerators without explicit associations are assigned integer values one greater than the value associated with the previous enumerators.

For example, the declaration

```
      enum colour {black,white=10,pink=-1,red,
                   green=3,yellow,blue};
```

The enumeration constants will now represent the following integer values :

black	0
white	10
pink	-1
red	0
green	3
yellow	4
blue	5

An enumeration constant must be unique with respect to other enumeration constants and variables of within the same name scope.

Thus, the declaration

```
      enum dyes {purple,orange,magneta,green};
```

is invalid as the identifier green has already been defined to be an enumerator of colour.

We can assign the values of enumerators to variables as shown below

```
 enum mar_status
 {
        single,
        married,
        divorced
}person1,person2;
person1=married; /* Assign value married to person1 */
person2=single;  /* Assign value single to person2 */
```

```
/* Example : enumerated data type */
# include <stdio.h>
# include <string.h>

enum e_dept    {
    Accounts,
    Software,
    Marketing,
};
struct emp     {
    char name[10];
    int age;
    float salary;
    enum e_dept dept;
};
char *dept_names[3] = { "Accounts", "Software", "Marketing" } ;

void main(void)
{

    struct emp e;
    strcpy(e.name,"Martin");
    e.age=35;
    e.salary=8865.70;
    e.dept=Software;

    /* Printing the value of e variable */
    printf("\nName       : %s",e.name);
    printf("\nAge        : %d",e.age);
    printf("\nSalary     : %f",e.salary);
    printf("\nDepartment : %s",dept_names[e.dept]);
}
```

Fig 10.5: Program using enumerated data type

Using enumeration variables in the program, can often increase the logical clarity of that program. These variables are particularly useful a flags, to indicate various options for carrying out a calculation or to identify various conditions that may have arisen as a result of previous internal calculations.

C does not provide facilities for reading or writing values of enumeration types. They may only be read or written as integer values.

10.9 Unions

Unions like structures, contain members whose individual data types may differ from one another. However, the members that compose a union all share the **same storage area** within the computer's memory, whereas each member within a structure is assigned its own unique storage area. The compiler allocates sufficient space to hold the largest data item in the union. Thus, unions are used to conserve memory. They are useful for applications involving multiple members, where values need not be assigned to all of the members at any one time.

The general format for declaration of union is

```
storage-class union tag_name
{
     data_type member1;
     data_type member2;
     ....
};
```

Where,

storage-class	Optional storage class specifier
union	A keyword that introduces a union definition.
tag	The name of the union
member1, member2	Set of type of declarations for the member data items that make up the union.

For example, the statement

```
union book_bank
{
     char *author;
     int pages;
     float price;
}data;
```

defines a variable *data* that can hold an *integer*, a *float* and a *pointer* to char.

Elements of a union are accessed in the same manner with the help of **dot operator** like structures.

For example, an integer pages is accessed as

```
      data.pages
```

Pointer variables can be declared along with the union declaration, or declared separately using tag name as follows

```
union book_bank
{
     char author;
     int pages;
     float price;
}data,*ptr;
```

or

```
    union book_bank *ptr;
```

declares data to be a variable of type *union book_bank* and the variable ptr to be a pointer to a *union data* variable.

10.9.1 Operations on a Union

In addition to the features discussed above, union has all the features provided to a structure except for minor changes which are a consequence of the memory sharing properties of a union.

Following are some valid operations on unions.

- An union variable can be assigned to another union variable.
- The address of the union variable can be extracted by using the address of operator(&).
- A function can accept and return a union or a pointer to a union.

10.9.2 Differences between Structures and Unions

There are important differences between **structures** and **unions** though the syntax used for declaring them is very similar.

Memory Allocation

The amount of memory required to store a structure variable is the sum of sizes of all the members in addition to the padding bytes that may be provided by the compiler. While in case of a union, the amount of memory required to store is the same as that required by its largest member.

This is illustrated using following example

```
/* program to check size of a structure and a union */
# include<stdio.h>
void main(void)
{
   struct
   {
     char name[20];
     int empno;
     float salary;
   }emp;
   union
   {
     char name[20];
     int empno;
     float salary;
   }desc;
   printf(“ The size of the structure is %d\n",sizeof(emp));
   printf(“ The size of the union is %d\n",sizeof(desc));
}
```

```
Output :
         The size of the structure is 31
         The size of the union is 20
```

Fig 10.6: Program to find the size of the structure and the union

Operations of Members

While all the **structure** members can be accessed at any point of time, only one member of a **union** may be accessed at any given time. This is because although a union contains sufficient storage for the largest type, it may contain only one value at a time; it is incorrect to store something as one type and then extract as another.

Thus the following statements -

```
     data.pages=100;
     printf("%s",data.price);
   /* where price is member of union data */
```

produce anomalous results.

It is the programmer's responsibility to keep track of the active variable (i.e. variable which was last accessed).

Appendix A: Table of Figures

Appendix B: List of tables