

Introduction to algorithm&

C

Part-1



FUNDAMENTALS OF ALGORITHMS

Unit Structure

- 1.0 Objectives
- 1.1 Introduction
- 1.2 An Overview
 - 1.2.1 What is an Algorithm?
 - 1.2.2 Various problems solved by algorithms.
 - 1.2.3 Algorithms as a Technology
- 1.3 Notation of algorithms
 - 1.3.1 Asymptotic notation
 - 1.3.2 Standard notations and common functions
- 1.4 Pseudo-Code Conventions
 - 1.4.1 Assignment statements
 - 1.4.2 Control structures
- 1.5 Let us sum up.
- 1.6 List of References
- 1.7 Theory Questions

1. OBJECTIVES

After going through this chapter, you will be able to:

- Define algorithm, use of algorithms
- Describe different notations of algorithms
- State standard notations and common functions
- Classify different Pseudo-code Conventions

1.1 INTRODUCTION

In today's world, different types of software are developed like system software and application software. Software is nothing but the collection of programs. And program is nothing but the set of instructions. To develop any type of software, one requires the strategy to go for it. Design must be done before writing any program.

To write any program in any language, you require the concepts that is business logic. These concepts are nothing but the step by step procedure. This procedure consists of input, process and output. This is generally a thinking process. One program can have multiple logics or concepts.

This concept or thinking process, one has to write on paper in a step wise manner before actual implementation of program on computer. This is nothing but an algorithm. It is therefore necessary to study about an overview of algorithms like what are the pseudo-code conventions, etc.

1.2 AN OVERVIEW

An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output.

1.2.1 What is an algorithm?

Algorithm can be defined in many ways like given below:

Definition 1.1

An algorithm is a computable set of steps to achieve a desired result.

Definition 1.2

An algorithm is a set of rules that specify the order and kind of arithmetic operations that are used on specified set of data.

Definition 1.3

An algorithm is a sequence of finite number of steps arranged in a specific logical order which, when executed, will produce a correct solution for a specific problem.

Definition 1.4

An algorithm is a set of instructions for solving a problem. When the instructions are followed, it must eventually stop with an answer.

Definition 1.5

An algorithm is a finite, definite, effective procedure, with some output.

The essential properties an algorithm should have:

* an algorithm is finite

(w.r.t.: set of instructions, use of resources, time of computation)

* instructions are precise and computable.

* instructions have a specified logical order, however, we can discriminate between

- deterministic algorithms (every step has a well-defined successor), and
- non-deterministic algorithms (randomized algorithms, but also parallel algorithms!)

* produce a result.

For each algorithm, especially a new one, we should ask the following basic questions:

- * does it terminate?
- * is it correct?
- * is the result of the algorithm determined?
- * how much memory will it use?

1.2.2 Various problems solved by algorithm.

As algorithm is a step by step procedure of any program, many problems are solved by algorithm. The following are some examples:

The internet enables people all around the world to quickly access and retrieve large amounts of information. In order to do so, clever algorithms are employed to manage and manipulate this large volume of data. Examples of problems which must be solved include finding good routes on which the data will travel.

Electronic commerce enables goods and services to be negotiated and exchanged electronically. The ability to keep information such as credit card numbers, passwords, and bank statements private is essential if electronic commerce is to be used widely.

In manufacturing and other commercial settings, it is often important to allocate the resources in the most beneficial way. An oil company may wish to know where to place its wells in order to maximize its expected profit. A airline may wish to assign crews to flights in the least expensive way possible, making sure that each flight is covered. All of these are examples of problems that can be solved using linear programming.

An example of an NP-hard problem is the decision subset sum problem, which is this: given a set of integers, does any non-empty subset of them add up to zero? That is a yes/no question, and happens to be NP-complete. Another example of an NP-hard problem is the optimization problem of finding the least-cost route through all nodes of a weighted graph. This is commonly known as the Traveling Salesman Problem.

There are also decision problems that are NP-hard but not NP-complete, for example the halting problem. This is the problem which asks "given a program and its input, will it run forever?" That's a *yes/no* question, so this is a decision problem. It is easy to prove that the halting problem is *NP-hard* but not *NP-complete*. For example, the Boolean satisfiability problem can be reduced to the halting problem by transforming it to the description of a Turing machine that tries all truth value assignments and when it finds one that satisfies the formula it halts and otherwise it goes into an infinite loop. It is also easy to see that the halting problem is not in *NP* since all problems in *NP* are decidable in a finite number of operations, while the halting problem, in general, is not. There are also NP-hard problems that are neither NP-complete nor undecidable. For instance, the language of True quantified Boolean formulas is decidable in polynomial space, but not non-deterministic polynomial time.

1.2.3 Algorithms as a Technology

Computers may be fast, but they are not infinitely fast. And memory may be cheap, but it is not free. Computing time is therefore a bounded resource, and so is space in memory. These resources should be used wisely and algorithms that are efficient in terms of time or space will help the programmer. The various points are measured for algorithms such as efficiency.

Check your progress

1) Give the real-world example in which one of the following computational problems appears: sorting, determining the best order for multiplying matrices, or finding the convex hull.

2) Give one example of an application that requires algorithmic content at the application level, and discuss the function of the algorithms involved.

1.3 NOTATION OF ALGORITHMS

1.3.1 Asymptotic notation

1] A notation – Little Oh

The relation $f(x) \in o(g(x))$ is read as " $f(x)$ is little-o of $g(x)$ ". Intuitively, it means that $g(x)$ grows much faster than $f(x)$. It assumes that f and g are both functions of one variable. Formally, it states

$$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = 0.$$

For example,

- $2x \in o(x^2)$
- $2x^2 \notin o(x^2)$
- $1/x \in o(1)$

Little-o notation is common in mathematics but rarer in computer science. In computer science the variable (and function value) is most often a natural number. In mathematics, the variable and function values are often real numbers. The following properties can be useful:

$$\begin{aligned} o(f) + o(f) &\subseteq o(f) \\ o(f) + o(g) &\subseteq o(fg) \\ o(o(f)) &\subseteq o(f) \\ o(f) &\subset o(f) \end{aligned}$$

(and thus the above properties apply with most combinations of o and O).

As with big O notation, the statement " $f(x)$ is $o(g(x))$ " is usually written as $f(x) = o(g(x))$, which is a slight abuse of notation.

II] A notation – Big Oh

Let $f(x)$ and $g(x)$ be two functions defined on some subset of the real numbers.
One writes ??????

if and only if, for sufficiently large values of x , $f(x)$ is at most a constant times $g(x)$ in absolute value. That is, $f(x) = O(g(x))$ if and only if there exists a positive real number M and a real number x_0 such that

$$|f(x)| \leq M |g(x)| \text{ for all } x > x_0.$$

In many contexts, the assumption that we are interested in the growth rate as the variable x goes to infinity is left unstated, and one writes more simply that $f(x) = O(g(x))$.

The notation can also be used to describe the behavior of f near some real number a (often, $a = 0$): we say

$$f(x) = O(g(x)) \text{ as } x \rightarrow a$$

if and only if there exist positive numbers δ and M such that

$$|f(x)| \leq M |g(x)| \text{ for } |x - a| < \delta$$

If $g(x)$ is non-zero for values of x sufficiently close to a , both of these definitions can be unified using the limit superior:

$$|f(x) = O(g(x)) \text{ as } x \rightarrow a \text{ if and only if } \limsup_{x \rightarrow a} \left| \frac{f(x)}{g(x)} \right| < \infty$$

Example of Big oh

In typical usage, the formal definition of O notation is not used directly; rather, the O notation for a function $f(x)$ is derived by the following simplification rules:

- If $f(x)$ is a sum of several terms, the one with the largest growth rate is kept, and all others omitted.
- If $f(x)$ is a product of several factors, any constants (terms in the product that do not depend on x) are omitted.

For example, let $f(x) = 6x^4 - 2x^3 + 5$, and suppose we wish to simplify this function, using O notation, to describe its growth rate as x approaches infinity. This function is the sum of three terms: $6x^4$, $-2x^3$, and 5 . Of these three terms, the one with the highest growth rate is the one with the largest exponent as a function of x , namely $6x^4$. Now one may apply the second rule: $6x^4$ is a product of 6 and x^4 in which the first factor does not depend on x . Omitting this factor results in the simplified form x^4 . Thus, we say that $f(x)$ is a big-oh of (x^4) or mathematically we can write $f(x) = O(x^4)$.

One may confirm this calculation using the formal definition: let $f(x) = 6x^4 - 2x^3 + 5$ and $g(x) = x^4$. Applying the formal definition from above, the statement that $f(x) = O(x^4)$ is equivalent to its expansion,

$$|f(x)| \leq M |g(x)|$$

for some suitable choice of x_0 and M and for all $x > x_0$. To prove this, let $x_0 = 1$ and $M = 13$. Then, for all $x > x_0$:

$$\begin{aligned} |6x^4 - 2x^3 + 5| &\leq 6x^4 + |2x^3| + 5 \\ &\leq 6x^4 + 2x^4 + 5x^4 \\ &= 13x^4, \\ &= 13|x^4| \end{aligned}$$

so

$$|6x^4 - 2x^3 + 5| \leq 13|x^4|.$$

Usage of Big oh

Big O notation has two main areas of application. In mathematics, it is commonly used to describe how closely a finite series approximates a given function, especially in the case of a truncated Taylor series or asymptotic expansion. In computer science, it is useful in the analysis of algorithms. In both applications, the function $g(x)$ appearing within the $O(\dots)$ is typically chosen to be as simple as possible, omitting constant factors and lower order terms.

There are two formally close, but noticeably different, usages of this notation: infinite asymptotics and infinitesimal asymptotics. This distinction is only in application and not in principle, however—the formal definition for the "big O" is the same for both cases, only with different limits for the function argument.

Infinite asymptotics of Big oh

Big O notation is useful when analyzing algorithms for efficiency. For example, the time (or the number of steps) it takes to complete a problem of size n might be found to be $T(n) = 4n^2 - 2n + 2$.

As n grows large, the n^2 term will come to dominate, so that all other terms can be neglected — for instance when $n = 500$, the term $4n^2$ is 1000 times as large as the $2n$ term. Ignoring the latter would have negligible effect on the expression's value for most purposes.

Further, the coefficients become irrelevant if we compare to any other order of expression, such as an expression containing a term n^3 or n^4 . Even if $T(n) = 1,000,000n^2$, if $U(n) = n^3$, the latter will always exceed the former once n grows larger than 1,000,000 ($T(1,000,000) = 1,000,000^3 = U(1,000,000)$). Additionally, the number of steps depends on the details of the machine model on which the algorithm runs, but different types of machines typically vary by only a constant factor in the number of steps needed to execute an algorithm.

So the big O notation captures what remains: we write either

$$T(n) = O(n^2)$$

or

$$T(n) \in O(n^2)$$

and say that the algorithm has *order of n^2* time complexity.

Note that "=" is not meant to express "is equal to" in its normal mathematical sense, but rather a more colloquial "is", so the second expression is technically accurate (see the "Equals sign" discussion below) while the first is a common abuse of notation.^[1]

Infinitesimal asymptotics of Big oh

Big O can also be used to describe the error term in an approximation to a mathematical function. The most significant terms are written explicitly, and then the least-significant terms are summarized in a single big O term. For example,

$e^x = 1 + x + \frac{x^2}{2} + O(x^3)$ as $x \rightarrow 0$ expresses the fact that the error, the difference $e^x - (1 + x + x^2/2)$ is smaller in absolute value than some constant times $|x^3|$ when x is close enough to 0.

Properties of Big oh

If a function $f(n)$ can be written as a finite sum of other functions, then the fastest growing one determines the order of $f(n)$. For example $f(n) = 9 \log n + 5(\log n)^3 + 3n^2 + 2n^3 \in O(n^3)$.

In particular, if a function may be bounded by a polynomial in n , then as n tends to *infinity*, one may disregard *lower-order* terms of the polynomial.

$O(n^c)$ and $O(c^n)$ are very different. The latter grows much, much faster, no matter how big the constant c is (as long as it is greater than one). A function that grows faster than any power of n is called *superpolynomial*. One that grows more slowly than any exponential function of the form c^n is called *subexponential*. An algorithm can require time that is both superpolynomial and subexponential; examples of this include the fastest known algorithms for integer factorization.

$O(\log n)$ is exactly the same as $O(\log(n^c))$. The logarithms differ only by a constant factor (since $\log(n^c) = c \log n$) and thus the big O notation ignores that. Similarly, logs with different constant bases are equivalent. Exponentials with different bases, on the other hand, are not of the same order. For example, 2^n and 3^n are **not** of the same order.

Changing units may or may not affect the order of the resulting algorithm. Changing units is equivalent to multiplying the appropriate variable by a constant wherever it appears. For example, if an algorithm runs in the order of n^2 , replacing n by cn means the algorithm runs in the order of c^2n^2 , and the big O notation ignores the constant c^2 . This can be written as $c^2n^2 \in O(n^2)$. If, however, an algorithm runs in the order of 2^n , replacing n with cn gives $2^{cn} = (2^c)^n$. This is not equivalent to 2^n in general.

Changing of variable may affect the order of the resulting algorithm. For example, if an algorithm's running time is $O(n)$ when measured in terms of the number n of *digits* of an input number x , then its running time is $O(\log x)$ when measured as a function of the input number x itself, because $n = \Theta(\log x)$.

III] A notation – Omega

Big Omega. $f(n)$ is said to be $\Omega(g(n))$ if \exists a positive real constant C and a positive integer n_0 such that

$$f(n) \geq Cg(n) \quad \forall n \geq n_0$$

An Alternative Definition: $f(n)$ is said to be $\Omega(g(n))$ if \exists a positive real constant C such that $f(n) \geq Cg(n)$ for infinitely many values of n .

The Θ notation describes asymptotic tight bounds.

1.3.2 Standard notations and orders of common functions

Orders of common functions

Here is a list of classes of functions that are commonly encountered when analyzing the running time of an algorithm. In each case, c is a constant and n increase without bound. The slower-growing functions are generally listed first.

See [table of common time complexities](#) for a more comprehensive list.

Notation	Name	Example
$O(1)$	<u>constant</u>	Determining if a number is even or odd; using a constant-size <u>lookup table</u> or <u>hash table</u>
$O(\log n)$	logarithmic	Finding an item in a sorted array with a <u>binary search</u> or a balanced search <u>tree</u>
$O(n^c), 0 < c < 1$	fractional power	Searching in a <u>kd-tree</u>
$O(n)$	<u>linear</u>	Finding an item in an unsorted list or a malformed tree (worst case); adding two n -digit numbers
$O(n \log n) = O(\log n!)$	<u>linearithmic</u> , loglinear, or	Performing a <u>Fast Fourier transform</u> ; <u>heapsort</u> , <u>quicksort</u> (best and average

	quasilinear	case), or <u>merge sort</u>
$O(n^2)$	<u>quadratic</u>	Multiplying two n -digit numbers by a simple algorithm; <u>bubble sort</u> (worst case or naive implementation), <u>shell sort</u> , quicksort (worst case), <u>selection sort</u> or <u>insertion sort</u>
$O(n^c), c > 1$	<u>polynomial</u> or algebraic	Tree- <u>adjoining grammar</u> parsing; maximum <u>matching</u> for <u>bipartite graphs</u>
$L_n[\alpha c], 0 < \alpha < 1 = e^{(c+\alpha)(\ln n)^\alpha (\ln \ln n)^{1-\alpha}}$	<u>L-notation</u> or <u>sub-exponential</u>	Factoring a number using the <u>quadratic sieve</u> or <u>number field sieve</u>
$O(c^n), c > 1$	<u>exponential</u>	Finding the (exact) solution to the <u>traveling salesman problem</u> using <u>dynamic programming</u> ; determining if two logical statements are equivalent using <u>brute-force search</u>
$O(n!)$	<u>factorial</u>	Solving the traveling salesman problem via brute-force search; finding the <u>determinant</u> with <u>expansion by minors</u> .

The statement $f(n) = O(n!)$ is sometimes weakened to $f(n) = O(n^n)$ to derive simpler formulas for asymptotic complexity.

For any $k > 0$ and $c > 0$, $O(n^c(\log n)^k)$ is a subset of $O(n^{c+a})$ for any $a > 0$, so may be considered as a polynomial with some bigger order.

Check your progress

- 1) Give the difference between the various notations Little Oh, Big Oh and Omega.

1.4 PSEUDO-CODE CONVENTIONS

An algorithm is a well-ordered collection of unambiguous, effectively computable instructions that produce a result and halt in a finite amount of time.

The instructions that are unambiguous and effectively computable depend on the computing agent executing the algorithm. Establishing a well-ordered collection of those instructions depends on the language used to describe the algorithm. While we can write algorithms for ourselves without deep reflection on this definition (since we understand what is effectively computable and unambiguous for ourselves), if we want to communicate algorithmic solutions to others, we must establish a reasonable common computing agent. Below we describe a set of instructions that will define a language and effectively computable instructions for a *pseudomachine* that will serve as our target computing agent.

Pseudocode is a compact and informal high-level description of a computer programming algorithm that uses the structural conventions of a programming language, but is intended for human reading rather than machine reading. Pseudocode typically omits details that are not essential for human understanding of the algorithm, such as variable declarations, system-specific code and subroutines. The programming language is augmented with natural language descriptions of the details, where convenient, or with compact mathematical notation. The purpose of using pseudocode is that it is easier for humans to understand than conventional programming language code, and that it is a compact and environment-independent description of the key principles of an algorithm. It is commonly used in textbooks and scientific publications that are documenting various algorithms, and also in planning of computer program development, for sketching out the structure of the program before the actual coding takes place.

No standard for pseudocode syntax exists, as a program in pseudocode is not an executable program. Pseudocode resembles, but should not be confused with, skeleton programs including dummy code, which can be compiled without errors. Flowcharts can be thought of as a graphical alternative to pseudocode.

1.4.1 Assignment statements

Our computing agent must be able to keep track of a variety of values needed in executing the algorithm. The values involved in the execution of an algorithm make up the *state* of the computation. The state is dynamic over the execution time of the algorithm, as values associated with particular attributes of the algorithm may change as the algorithm executes.

As algorithm writers, we need a way to describe the various values in the state of the computation. Each value will be describe with a name, written in lowercase letters, and called a *variable*. For example, we may want to write an algorithm that outputs the integers between one and ten. In order to know which value to output next, we could establish a variable count that will represent the count to output next.

As an algorithm designer you should think of a variable as a box holding a value. The name of the variable (e.g. count) is the unique name of the box.

Lists of values and indices

Sometimes we need to keep track of lists of associated values, for example a list of ten names. Rather than think up ten different variable names, we can use list notation and refer to name[1], name[2], ..., name[10] to refer to the ten names. We say that name is the list of values and that the integer in [] is the *index* indicating which value in the list we are referring to.

When using a list of values, we may use a variable to keep track of which item in the list we are interested in. For example we may have a variable *i* that keeps track of which name we want to look at. name[*i*] would then refer to the value in list name found at the index indicated by the value of *i*.

Interacting with the User

After an algorithm designer writes an algorithm, a user decides to run the algorithm on a particular computing agent. The computing agent will need to interact with the user at least once, when outputting the result to the user. The agent may also need to receive information from the user.

Output from computing agent to user

Any of the following may be used to indicate the agent is outputting information to the user: Output Print Display

The command can be followed by text in quotes, which is output verbatim, and variables, whose value is output. Some examples:

Output "Please enter an integer"

Output "The current counter value is " count

Input from user to agent

Input from the user is indicated by Input followed by a list of variables that will store the values input from the user.

For example, Input in inputs a single value, storing it into the variable named n.

Changing the value of a variable

Algorithms need to be able to change the value of variables. To set the value of a variable to a new value we will use the following command:

Set *variable* To *value* Where *variable* is the name of the variable whose value the algorithm is changing and *value* is an expression whose value should be stored in the variable.

What are expressions?

Many times, the expressions set into the variable are mathematical expressions. The usual operators +, -, *, / are available. We also have two other operators, div and mod. The value $x \text{ div } y$ is the integer portion of x / y . The value $x \text{ mod } y$ is the integer remainder of x / y . For example, $7 \text{ div } 2$ is 3 and $7 \text{ mod } 2$ is 1 (because 2 goes into 7 three times with a remainder of 1).

Expressions can also be *boolean*. Boolean expressions have only two possible values, true or false. The operators used in boolean expressions include the usual comparison operators: <, =, >, <=, >= and logical operators, AND, OR, NOT.

1.4.2 Control structures

To describe a point in the algorithm in which steps are executed only if some condition is true, we use the if statement. if (*condition*) then BEGIN *statements...* END *statements...* indicates any statements may be placed between the BEGIN/END. The statements are executed only if the *condition* expression evaluates to true.

One may also want to indicate an alternative set of statements to be executed if the condition evaluates to false. In this case the statement is written as:

```
If (condition) Then
BEGIN
    statements...
END
Else
BEGIN
    statements...
END
```

Iteration

To indicate repetition of statements in an unambiguous way, the algorithm will use one of the following statements:

```
Repeat
    statements...
Until (condition)
```

executes the statements inside the Repeat/Until block, then tests the condition. If the condition is false, the statements are executed again. Note that the statements are always executed at least once.

```
RepeatWhile (condition)
BEGIN
    statements...
END
```

tests the condition. If it is true, the statements inside the BEGIN/END block are executed, and then the condition is tested again. When the condition is false, control falls out of the loop.

Example

The following algorithm inputs an integer and outputs the values between 1 and that value.

Output "Please enter a positive integer value"

```
Input n
Set count To 1
RepeatWhile (count < n+1)
BEGIN
    Output count
    Set count To (count + 1)
END
```

Check your progress

- 1) What are the different assignment statements?
 - 2) What are the different control structure statements?
 - 3) Write any one algorithm using Pseudo-Code Conventions.
-

1.5 LET US SUM UP

Thus, we have studied the basics of algorithm. How the real time problems can be solved by algorithms as well as the different notations are used to write algorithm that are studied. Thus, algorithms are very much helpful to develop the logical concepts on paper.

1.6 LIST OF REFERENCES

- 1) "Introduction of Algorithms", 2nd Edition, Thomas H. Cormen
- 2) <http://lcm.csa.iisc.ernet.in/>.

1.7 Theory Questions

1) What is meant by the complexity of an algorithm?

Complexity of an algorithm is the study of how long a program will take to run, depending on the size of its input & long of loops made inside the code.

2) Compare Complexity of sorting algorithms?

Bubble sort, selection sort and insertion sort have average complexity of $O(n^2)$... merge sort and heap have complexity $O(n \log n)$and quick sort has $O(n \log n)$ for average case ...its worst case...

3) What is Time complexity of binary search?

The best case complexity is $O(1)$ i.e if the element to search is the middle element. The average and worst case time complexity are $O(\log n)$.

4) What do you mean by Best, worst and average case complexity?

The best, worst and average case complexity refer to three different ways of measuring the time complexity (or any other complexity measure) of different inputs of the same size. Since some inputs of size n may be faster to solve than others, we define the following complexities:

- Best-case complexity: This is the complexity of solving the problem for the best input of size n .
- Worst-case complexity: This is the complexity of solving the problem for the worst input of size n .
- Average-case complexity: This is the complexity of solving the problem on an average. This complexity is only defined with respect to a probability distribution over the inputs. For instance, if all inputs of the same size are assumed to be equally likely to appear, the average case complexity can be defined with respect to the uniform distribution over all inputs of size n .

For example, consider the deterministic sorting algorithm quicksort. This solves the problem of sorting a list of integers which is given as the input. The best-case scenario is when the input is already sorted, and the algorithm takes time $O(n \log n)$ for such inputs. The worst-case is when the input is sorted in reverse order, and the algorithm takes time $O(n^2)$ for this case. If we assume that all possible permutations of the input list are equally likely, the average time taken for sorting is $O(n \log n)$.

5) What are the different types of algorithm?

The following are the various types of algorithm:

- Dynamic Programming Algorithms: This class remembers older results and attempts to use this to speed the process of finding new results.
- Greedy Algorithms: Greedy algorithms attempt not only to find a solution, but to find the ideal solution to any given problem.
- Brute Force Algorithms: The brute force approach starts at some random point and iterates through every possibility until it finds the solution.
- Randomized Algorithms: This class includes any algorithm that uses a random number at any point during its process.
- Branch and Bound Algorithms: Branch and bound algorithms form a tree of subproblems to the primary problem, following each branch until it is either solved or lumped in with another branch.
- Simple Recursive Algorithms: This type of algorithm goes for a direct solution immediately, then backtracks to find a simpler solution.
- Backtracking Algorithms: Backtracking algorithms test for a solution, if one is found the algorithm has solved, if not it recurs once and tests again, continuing until a solution is found.
- Divide and Conquer Algorithms: A divide and conquer algorithm is similar to a branch and bound algorithm, except it uses the backtracking method of recurring in tandem with dividing a problem into subproblems.



ALGORITHMS PROBLEMS AND ANALYSIS OF ALGORITHMS

Unit Structure

- 2.0 Objectives
- 2.1 Introduction
- 2.2 Develop Fundamentals algorithms
 - 2.2.1 Exchange the values of two variables with and without temporary variable
 - 2.2.2 Counting positive number from a set of integers
 - 2.2.3 Summation of set of numbers
 - 2.2.4 Reversing the digits of an integer
 - 2.2.5 Find smallest positive divisor of an integer other than 1
 - 2.2.6 Find G.C.D and L.C.M. of two as well as three positive integers
 - 2.2.7 Generating prime numbers
- 2.3 Analysis of algorithms
 - 2.3.1 Running time of an algorithm
- 2.4 Let us sum up.
- 2.5 List of References
- 2.6 Unit End Exercises

1. OBJECTIVES

After going through this chapter, you will be able to:

- Develop fundamental algorithms
- Analyze algorithms in terms of time and space complexities
- Write different algorithms for different problems

2.1 INTRODUCTION

In the last chapter, we have seen the concepts of algorithm. How to use various notations for algorithms. This is going to be helpful to us to develop the algorithms.

In this chapter, we are going to develop the algorithm for simple problems. We will also write the code using C / C++ language. This helps us to develop the algorithms for more complex problems.

Whenever the algorithms are written, the two things are considered. First one is the time complexity which will check the time utilization of CPU as well as other resources. And second thing is space complexity which will check the space utilization of primary memory. There has to be immediate output and minimum space utilization. These are the different factors such as turn around time, throughput, etc

2.2 DEVELOP FUNDAMENTALS ALGORITHMS

This section discusses more simple problems which are fundamental problems or basic problems based on which anybody can develop the algorithms on more complex problems. The following four steps are important while writing the algorithm.

Declaration section: In this step, the variables or constants which are used in the program are declared.

Input section: In this step, the variables need to be initialized to perform the operation.

Process section: In this step, the actual business logic is resided. The expression statements, conditional statements are written in this section.

Output section: After performing the operation, the output should be displayed to the user.

The following are some of the problems.

2.2.1 Exchange the values of two variables with and without temporary variable

The algorithm (with temporary variable) is as follows:

- 1) Include library header files such as `stdio.h`, etc.
- 2) Declare the prototype of the function as `exchange`
- 3) Define main function
 - a. Declare two integer variables `a,b`
 - b. Initialize the two variables
 - c. Display the variables which are initialized to user
 - d. Call `exchange` function by passing the addresses of `a,b`
- 4) Function `exchange`
 - a. Declare the temporary pointer variable `temp`
 - b. Process
 - i. `temp = *a;`
 - ii. `*a = *b;`
 - iii. `*b = temp;`
 - c. Display the pointer variables `*a, *b` after exchange

All it needs to know is what `exchange` arguments look like. This way we can put the `exchange` function after our main function. We could have easily put `exchange` before main and gotten rid of the declaration. The following code segment will fix `exchange` to use pointers, and move `exchange` above main to eliminate the need for the forward declaration.

```
#include <stdio.h>

void exchange ( int *a, int *b )
{
    int temp;

    temp = *a;
    *a = *b;
    *b = temp;
    printf(" From function exchange: ");
    printf("a = %d, b = %d\n", *a, *b);
}

void main()
{
```

```

int a, b;

a = 5;

b = 7;

printf("From main: a = %d, b = %d\n", a, b);

exchange(&a, &b);

printf("Back in main: ");

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

}

```

The rule of thumb here is that

- You use regular variables if the function does not change the values of those arguments
- You **MUST** use pointers if the function changes the values of those arguments

The following concept is to exchange the two variables without temporary variable.

Let's consider $A=0x10100000$ and $B=0x00001010$ then the contents of these A and B variables should be exchanged without third variable.

Step 1

(A) Exclusive OR (B), the output is as follows

(A) 10100000

(B) 00001010

(A) 10101010 XOR Result – contents of A

(B) 00001010 B is unchanged

Step 2

Now (B) Exclusive OR (A), the output is as follows

(B) 00001010

(A) 10101010

(B) 10100000 XOR Result – contents of B
 (A) 10101010 A is unchanged - same as after Step 1

Step 3

Now (A Exclusive OR (B), the output is as follows

(A) 10101010
 (B) 10100000

(A) 00001010 XOR Result – contents of A
 (B) 10100000 B is unchanged - same as after Step 2

Finally A has 00001010 and B has 10100000 (the two variables have been swapped) and a temporary variable is not used.

2. Counting positive number from a set of integers

Here the idea is to take the array of set of positive as well as integer values. An array is nothing but the set of similar type of elements. Now, in this array, the integer positive values are checked.

The algorithm is as follows:

- Include the library (header file) such as stdio.h
- Define main function
 - Declare an integer array of size 50 (means 0 to 49 range of elements can be kept in it) along with 3 other integer variables
 - Input the size the array and store it in n variable
 - Input the values in array using for loop
 - Check each value in array for positive integer value.
 - If $a[i] > 0$
 - Increment the counter
 - Display the number of positive numbers.

/* Program to count the no of positive numbers*/

```
#include< stdio.h >
void main()
{
int a[50],n, count_pos=0,i;
printf("Enter the size of the array\n");
scanf("%d",&n);
printf("Enter the elements of the array\n");
for i=0;i < n;i++)
scanf("%d",&a[i]);
```

```

for(l=0;l < n;l++)
{
if(a[l] > 0)
count_pos++;
}
printf("There are %d positive numbers in the array\n",count_pos);
}

```

3. Summation of set of numbers

To solve this problem, three arrays are declared. First two arrays will store the input values whereas the third array will store the result of the first two arrays addition.

The following algorithm is for summation of set of numbers:

- 1) Include the library (header file) as stdio.h
- 2) Define main function
 - a. Declare three integer arrays a[], b[] and c[]
 - b. Input the size of an array.
 - c. Input the two integer array as per size
 - d. Add two integer arrays and store the result in third integer array.
 - i. $c[I] = a[I] + b[I]$
 - e. Display the result

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

```
void main()
```

```

{
int a[50],n, b[50],I, c[50];
printf("Enter the size of the array\n");
scanf("%d",&n);
printf("Enter the elements of the array\n");
for I=0;I < n;I++)
scanf("%d",&a[I]);
for I=0;I < n;I++)
scanf("%d",&b[I]);
for I=0;I < n;I++)
c[I] = a[I] + b[I];
for I=0;I < n;I++)
printf("%d",c[I]);
}

```

2.2.4 Reversing the digits of an integer

This program expects the digits of an integer to be reversed. For example 123 as input should get reversed and output will be 321.

The following is the algorithm:

- 1) Include library (header file) such as stdio.h
- 2) Define the reversDigit() function
 - a. Declare two static variables initialized as 0 and 1 respectively.
 - b. If number > 0


```
reversDigits(num/10);

rev_num += (num%10)*base_pos;

base_pos *= 10;
```
- 3) Return rev_num
- 4) Define main function
 - a. Call the above function by passing the values
 - b. Display the result (reversed number)

```
#include <stdio.h>

/* Recursive function to reverse digits of num*/
int reversDigits(int num)
{
    static int rev_num = 0;
    static int base_pos = 1;
    if(num > 0)
    {
        reversDigits(num/10);
        rev_num += num%10*base_pos;
        base_pos *= 10;
    }
    return rev_num;
}

/*Driver program to test reversDigits*/
int main()
```

```

{
    int num = 4562;
    printf("Reverse of no.is d", reversDigits(num);

    getchar();

    → return 0;
}

```

Time Complexity: $O(\log(n))$ where n is the input number
Space Complexity: $O(1)$

1.2.5 Find smallest positive divisor of an integer other than 1

To get an idea about the given problem, let us take the example and get a result. Following is an example for the same:

Dividing each of 1, 2, 3, . . . , 12 into 152 reveals the divisors 1, 2, 4, and 8. The remaining positive divisors of 152 are then given by

$$152/2 = 76$$

$$152/4 = 38$$

$$152/8 = 19$$

Therefore, the divisors for 152 are 2, 4, 8, 19, 38, 76

The following code is to display the number which is positive divisor of an integer.

```

#include <stdio.h>
int main()
{
    int n, sum = 0, i;
    scanf("%d",&n); // read the input
    for(i = 1; i<n; i++)
    {
        if(n % i == 0)
        {

```

```

        sum += i;
    }
}
if(sum == n)
{
    printf("%d",n);
}
Else
{
    printf("NO\n");
}
→ return 0;
}

```

6. Find G.C.D and L.C.M. of two as well as three positive integers

The algorithm is as follows:

- 1) Include header file such as `stdio.h`
- 2) Define main function
 - a. Call lcm function by passing two integer values
 - i. Declare one integer variable 'n'
 - ii. Check for `if(n%a == 0 && n%b == 0)` from 1 to n times
 - iii. If true then return n
- 3) Display n

`/* a & b are the numbers whose LCM (Least common multiple) is to be found */`

```

int lcm(int a, int b)
{
    int n;
    for(n=1;;n++)
    {
        if(n%a == 0 && n%b == 0)
            return n;
    }
}

```

The algorithm is as follows:

- 1) Include header file such as `stdio.h`
- 2) Define main function
 - a. Call gcd function by passing two integer values

- i. Declare one integer variable 'n'
- ii. $c = a \% b$; do this till it is true
- iii. check for $\text{if}(c == 0)$
- iv. If true then return b
- v. $a = b$; and $b = c$;

3) Display b

/* a & b are the numbers whose GCD (Greatest common Divisor) is to be found given $a > b$ */

```
int gcd(int a,int b)
{
    int c;
    while(1)
    {
        c = a%b;
        if(c==0)
            return b;
        a = b;
        b = c;
    }
}
```

7. Generating prime numbers

In this problem, the prime numbers have to be generated. Prime number is nothing but the number is only divisible by itself. The algorithm is as follows:

- 1) Include header file such as `stdio.h`
- 2) Define main function
 - a. Declare four integer variables n, i, j, c
 - b. Input the end number up to which numbers are to be generated as prime numbers
 - c. While($i \leq n$)
 - i. Initialize the counter variable $c = 0$;
 - ii. Loop from $j = 1$ to i
 1. Check for $(i \% j == 0)$
 - a. Increase the counter by one as $c++$
 2. Check if($c == 2$)
 - a. display i
 3. end of while loop

```
#include <stdio.h>
main()
{
    int n,i=1,j,c;
    clrscr();
    printf("Enter Number Of terms");
    printf("Prime Numbers Are Following");
    scanf("%d",&n);
```

```

while(i<=n)
{
    c=0;
    for(j=1;j<=i;j++)
    {
        if(i%j==0)
            c++;
    }
    if(c==2)
        printf("%d",i)
    i++;
}
getch();
}

```

Check your progress

- 1) Write an algorithm for finding factorial number and also write a code in C/C++.
- 2) Write an algorithm for finding Fibonacci series and also write a code in C/C++.
- 3) Write an algorithm for finding the average of 10 integer values along with a code in C/C++.

2.3 ANALYSIS OF ALGORITHMS

Analysis of Algorithms (AofA) is a field in computer science whose overall goal is an understanding of the complexity of algorithms. While an extremely large amount of research is devoted to worst-case evaluations, the focus in these pages is methods for average-case and probabilistic analysis. Properties of random strings, permutations, trees, and graphs are thus essential ingredients in the analysis of algorithms.

To analyze an algorithm is to determine the amount of resources (such as time and storage) necessary to execute it. Most algorithms are designed to work with inputs of arbitrary length. Usually the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps (time complexity) or storage locations (space complexity).

Algorithm analysis is an important part of a broader computational complexity theory, which provides theoretical estimates for the resources needed by any algorithm which solves a given computational problem. These estimates provide an insight into reasonable directions of search for efficient algorithms.

In theoretical analysis of algorithms it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input. Big O notation, omega notation and theta notation are used to this end. For instance, binary search is said to run in a number of steps proportional

to the logarithm of the length of the list being searched, or in $O(\log(n))$, colloquially "in logarithmic time". Usually asymptotic estimates are used because different implementations of the same algorithm may differ in efficiency. However the efficiencies of any two "reasonable" implementations of a given algorithm are related by a constant multiplicative factor called a hidden constant.

2.3.1 Running time of an algorithm

Run-time analysis is a theoretical classification that estimates and anticipates the increase in running time (or run-time) of an algorithm as its input size (usually denoted as n) increases. Run-time efficiency is a topic of great interest in Computer Science: A program can take seconds, hours or even years to finish executing, depending on which algorithm it implements (see also performance analysis, which is the analysis of an algorithm's run-time in practice).

Since algorithms are platform-independent (i.e. a given algorithm can be implemented in an arbitrary programming language on an arbitrary computer running an arbitrary operating system), there are significant drawbacks to using an empirical approach to gauge the comparative performance of a given set of algorithms.

Take as an example a program that looks up a specific entry in a sorted list of size n . Suppose this program were implemented on Computer A, a state-of-the-art machine, using a linear search algorithm, and on Computer B, a much slower machine, using a binary search algorithm. Benchmark testing on the two computers running their respective programs might look something like the following:

n (list size)	Computer A run-time (in nanoseconds)	Computer B run-time (in nanoseconds)
15	7 ns	100,000 ns
65	32 ns	150,000 ns
250	125 ns	200,000 ns
		250,000 ns
1,000	500 ns	

Based on these metrics, it would be easy to jump to the conclusion that Computer A is running an algorithm that is far superior in efficiency to what Computer B is running. However, if the size of the input-list is increased to a sufficient number, that conclusion is dramatically demonstrated to be in error.

I] Best case

The term best-case performance is used in computer science to describe the way an algorithm behaves under optimal conditions. For example, the best case for a simple linear search on a list occurs when the desired element is the first element of the list.

Development and choice of algorithms is rarely based on best-case performance: most academic and commercial enterprises are more interested in improving average performance and worst-case performance

II] Average case

Determining what average input means is difficult, and often that average input has properties which make it difficult to characterise mathematically (consider, for instance, algorithms that are designed to operate on strings of text). Similarly, even when a sensible description of a particular "average case" (which will probably only be applicable for some uses of the algorithm) is possible, they tend to result in more difficult to analyse equations.

III] Worst case

Worst-case analysis has similar problems: it is typically impossible to determine the exact worst-case scenario. Instead, a scenario is considered such that it is at least as bad as the worst case. For example, when analysing an algorithm, it may be possible to find the longest possible path through the algorithm (by considering the maximum number of loops, for instance) even if it is not possible to determine the exact input that would generate this path (indeed, such an input may not exist). This gives a safe analysis (the worst case is never underestimated), but one which is pessimistic, since there may be no input that would require this path.

Check your progress

- 1) Write an algorithm for sequential search and find its best, average and worst case.
- 2) Write an algorithm for binary search and find its best, average and worst case.
- 3) Write an algorithm for bubble sort and find its best, average and worst case.
- 4) How does one calculate the running time of an algorithm?
- 5) How can we compare two different algorithms?
- 6) How do we know if an algorithm is 'optimal'?

2.4 Let us sum up

Thus, in this chapter we have studied how to develop the algorithms along with the source code. After doing these small problems, more complex problems can be solved by developing the algorithm. Running time of algorithms are analyzed afterwards in terms of best case / average case / worst case.

2.5 LIST OF REFERENCES

1) "Introduction of Algorithms", 2nd Edition, Thomas H. Cormen

2.6 UNIT END EXERCISES

1) Write an algorithm to reverse digits of a number in iterative way.

Ans:

Algorithm:

Input: num

(1) Initialize rev_num = 0

(2) Loop while num > 0

(a) Multiply rev_num by 10 and add remainder of num

divide by 10 to rev_num

rev_num = rev_num*10 + num%10;

(b) Divide num by 10

(1) Return rev_num

2) What are the different algorithm paradigms?

Ans: The different following paradigms are:

- Divide and Conquer.
- Dynamic Programming
- Greedy method
- Backtracking

3) Give the example of some algorithm along their run time complexity.

Ans: There are following three algorithms with their run time complexity:

- There is an algorithm (mergesort) to sort n items which has run-time $O(n \log n)$.
- There is an algorithm to multiply 2 n -digit numbers which has run-time $O(n^2)$.
- There is an algorithm to compute the n th Fibonacci number which has run-time $O(\log n)$.

4) What do you mean by Asymptotic Analysis?

Ans: Noting problems in providing a precise analysis of the running time of programs, computer scientists developed a technique which is often called asymptotic analysis. In asymptotic analysis of algorithms, one describes the general behavior of algorithms in terms of the size of input, but without delving into precise details.

There are many issues to consider in analyzing the asymptotic behavior of a program. One particularly useful metric is an upper bound on the running time of an algorithm. The "big O" is to be called of an algorithm.

Big O is defined somewhat mathematically, as a relationship between functions.

$f(n)$ is $O(g(n))$ iff

- there exists a number n_0
- there exists a number $d > 0$
- for all $n > n_0$, $\text{abs}(f(n)) \leq \text{abs}(d * g(n))$

It says that after a certain point (n_0), $f(n)$ is bounded above by a constant (d) times $g(n)$. The constant (d) helps accommodate the variation in the algorithm.

For algorithms,

- n is the "size" of the input (e.g., the number of items in a list or vector to be manipulated).
- $f(n)$ is the running time of the algorithm.

Some common big-O bounds

- An algorithm that is $O(1)$ takes constant time. That is, the running time is independent of the input. Getting the size of a vector is often an $O(1)$ algorithm.
- An algorithm that is $O(n)$ takes time linear in the size of the input. That is, we basically do constant work for each "element" of the input. Finding the smallest element in a list is often an $O(n)$ algorithm.

An algorithm that is $O(\log_2(n))$ takes logarithmic time. While the running time is dependent on the size of the input, it is clear that not every element of the input is processed.



OPERATORS IN C AND TYPE CONVERSIONS

3.0 OBJECTIVES

At the end of this chapter, you will be able to perform various types of operations on the data values which are to be processed.

3.1 INTRODUCTION

Operators in C are used to perform various operations on the variables and constants. The values on which the operation is to be performed are called as operands e.g., $a - b$. In this operation a and b are called as operands and “-” is called as the operator.

Operators which operate upon one operand are called as unary operators. Those which operate on two operands are called binary operators and those which operate on three operands are called ternary operators.

Operators can be classified as follows

- (i) Arithmetic Operators
- (ii) Relational Operators
- (iii) Logical Operators
- (iv) Assignment Operators
- (v) Compound Assignment Operators
- (vi) Increment and decrement Operators
- (vii) Conditional Operators
- (viii) Bitwise Operators
- (ix) Comma Operators/Separator

3.1.1 Arithmetic Operators

As the name suggests, Arithmetic Operators are used to perform arithmetic operations i.e., to perform calculations. These operators operate upon numeric data.

The Arithmetic operators include Add, Subtract, Multiply, Divide and modulus.

Add operator is denoted by + sign. This operator adds the value of the first operand to the second and is written as $a + b$

Ex. $a = 5, b = 6$ then $a + b = 11$

Subtract operator is denoted by – sign. This operator subtracts the value of the second operand from the value of the first operand

Ex. $a = 12, b = 4$ then $a - b = 8$

Multiply operator is denoted by * sign. This operator multiplies the value of the first operand with the value of the second operand.

Ex. $a = 12, b = 4$ then $a * b = 48$

Divide operator is denoted by / sign. This operator divides the value of the first operand by the value of the second.

Ex. $a = 12, b = 4$ then $a / b = 3$

Modulus operator is denoted by % sign. This operator divides the value of the first operand by the value of the second and the result of the operation is the remainder.

Ex. $a = 12, b = 5$ $a \% b = 2$

3.2.1 Relational Operators

Relational operators are used to compare two operands. These operators are used in the test conditions of the control statements (Chapter V).

The relational operators are :

- (i) < Less than
- (ii) < = Less than or equal
- (iii) > greater than
- (iv) > = greater than or equal

Less than operator is denoted by < sign. This operator used to test whether first operand is less than the second.

Ex. $x < y$

Less than or equal to operator is denoted by < = sign. This operator is used to test whether the first operand is less than or equal to the second operand.

Ex. $x <= y$

Greater than operator is denoted by > sign. This operator is used to test whether the first operand is greater than the second operand.

Ex. $x > y$

Greater than or equal to is denoted by > = sign. This operator is used to test whether the first operand is greater than or equal to the second operand.

Ex. $x > = y$

3.1.3 Equality Operators

The equality operator is denoted by = = sign and it is used to test whether the first operand is equal to the second operand.

Ex. $x = = y$

Not Equal Operator

The not equal operator is denoted by `!=` sign and it is used to test whether the first operand is not equal to the second operand.

Ex. `x != y`

3.1.4 Logical Operators:

Logical operators AND , OR are used to join two or more test conditions and the Logical operator NOT is used with a single condition.

The AND operator is denoted by `&&` sign.

Ex. `if (x == 2 && y > 3)`

The AND operator evaluates the first condition and if it is true, it evaluates the second condition and if this condition is also true, the result of the AND operator is true. However, if the first condition is false, the second condition is not evaluated and the result of the AND operator is false. In other words, the outcome of the AND operator is true only if both the conditions are true.

OR operator is denoted by `||` sign

Ex. `if (x == 2 || y > 3)`

The OR operator evaluates both the conditions and even if any one condition is true, the result of the OR operator is true. In other works, the outcome of the OR operator is FALSE only if both the conditions are false.

NOT operator is used with a single condition along with the equal `=` sign. NOT operator is denoted by `!` and it is used to test whether an operand is not equal to another operand.

Ex. `x != y`

3.1.5 Assignment operator

The Assignment operator is denoted by the = sign. It is used to assign a value from the right hand side of the equal sign to the operand on the left hand side of the equal sign.

Ex. (i) $x = 20$

(ii) $x = a + b$

3.1.6 Compound Assignment Operators

Compound Assignment operators are a combination of arithmetic operators and the equality operator.

They are $+=$, $-=$, $*=$, $/=$, $\%=$

$+=$ Add assignment operator

$-=$ Minus assignment operator

$*=$ Multiply assignment operator

$/=$ Divide assignment operator

$\%=$ Modulus assignment operator

Each of the operators first performs the arithmetic operation and then the assignment operation.

Ex. `int a = 21`

Operation	Equivalent to	Output
$a += 5$	$\Rightarrow a = a + 5$	26
$a -= 5$	$\Rightarrow a = a - 5$	16
$a *= 5$	$\Rightarrow a = a * 5$	105
$a /= 5$	$\Rightarrow a = a / 5$	4
$a \% = 5$	$\Rightarrow a = a \% 5$	1

3.1.7 Increment and Decrement operators

Increment and Decrement operators are special type of operators in C.

The Increment operator is denoted by `++` and is used to increase the value of the operand by 1.

The Decrement operator is denoted by `--` and is used to decrease the value of the operand by 1.

Both these operators have a special property. They can be placed before or after the operand.

If the operator is placed before the operand, it is called as prefix operator and if it placed after the operand, it is called as postfix operator.

If the operator is a prefix operator the value of the operand changes before it is used and if it is a postfix operator, the value of the operand changes after it has been used. Both these operators are unary operators.

Ex. (i) `x = 10;`
`y = ++x;`

increases the value of `x` by 1 and then assigns the value to `y`.

Thus, after the execution of the 2 statements `x = 11, y = 11`

(ii) `x = 10;`
`y = x ++;`

value of `x` is assigned to `y` and then the value of `x` increases by 1

Thus, after the execution of the two statements `x = 11, y = 10`

(iii) `x = 10;`

decreases the value of `x` by 1 and then assigns the value of `y`

$$y = --x;$$

Thus, after the execution of the 2 statements $x = 9, y = 9$

(iv) $x = 10;$
 $y = x--;$

value of x is assigned to y and then
 the value of x decreases by 1

Thus, after the execution of the 2 statements $x = 9, y = 10$

3.1.8 Conditional Operators

Conditional operators are denoted by '?' and ':'. These operators are used in the place of if / else statements.

The syntax of the conditional operators is as follows :

(test conditions) ? statement 1 : statement 2;

The test condition is checked and if it is true statement 1 is executed otherwise statement 2 is executed. The conditional operator is called as ternary operator because it operates on 3 operands.

Ex. `int x, y;`

`(x > y) ? printf ("%d", x); printf ("%d", y);`

If the value of x is greater than y , x gets printed otherwise y gets printed.

3.1.9 Bitwise Operators

The Bit operators include AND OR, XOR, left shift and right shift. These operators change the value of the operand at the bit level. The value is converted into its corresponding binary form and is then operated upon.

AND operator is denoted by '&' and it is used to join two operands by AND. If the corresponding bits of both the operands is 1 (ON) then the result of the operator is 1 otherwise it is 0.

Ex.

```
int x = 3, y = 5,
z = x & y;

x = 3 = 00000011
y = 5 = 00000101
-----
x & y = 00000001
      = 1
```

OR operator is denoted by '|' and it is used to join two operands by OR. If either of the two operands is 1 (True) the result is 1, otherwise it is 0.

Ex. int x = 3, y = 5, z;

```
z = x | y;

x = 3 = 00000011
y = 5 = 00000101
y = 5 = 00000101
z = 7
```

XOR is denoted by '^' and it is used to join the two operands by XOR. The result is 1 when only one of the two operands is 1 otherwise it is zero.

Ex. int x = 3, y = 5, z;

```
z = x ^ y;

x = 3 = 00000011
y = 5 = 00000101
x ^ y = 00000110
∴ z = 6
```

Left shift operator

This operator is denoted by `<<` and is used to shift off the bits on the left hand side and all vacated bits on the right hand side are replaced by zeros.

Ex. `int x = 3, z;`

```
z = x << 2;
x = 3 = 00000011
x << 2 = 00001100
z = 12
```

Right shift operator

This operator is denoted by `>>` and is used to shift the bits on the right hand side and all vacated bits on the left hand side are replaced by zeros.

Ex. `int x = 3, z;`

```
z = x >> 1;
x = 3 = 00000011
x >> 1 = 00000001
z = 1
```

Comma operator is treated as a separator between 2 operations.

Ex. `++m, k - = 2`

3.2 EXPRESSIONS IN C

Expressions in C consists of operands which are joined with the help of operators.

Ex. $3 * a + b$

In the above expression 3, a, b are operands and *, + are operators.

Any expression produces a value. Whenever two or more operators are present in an expression, the expression is evaluated in a particular order. This order is called as the precedence. The operator with the highest precedence is evaluated first and then to the next level and so on.

Associativity

When 2 operators having the same precedence are present in an expression, then the associativity of the operators is used to determine the sequence of the evaluation. The associativity may be from left to right or from right to left.

The following Table shows various operators, their precedence and their associativity.

Precedence	Operator	Associativity
1	!, ++, --, -	Right to Left
2	*, /, %	Left to Right
3	+, -	Left to Right
4	<<, >>	Left to Right
5	<, <=, >, >=	Left to Right
6	=, !=	Left to Right
7	&	Left to Right
8	^, !, &&	Left to Right
9	::	Left to Right

10	? : , = , + = , - = ,	Right to Left
	* = , / = , % =	
11	,	Left to Right

Type Conversion

Whenever an expression involves operands which are not of the same type, it is called as mixed data type expression and before such an expression is evaluated; all operands have to be converted into the same type.

C converts lower data types to higher data types.

The hierarchy of the data types is as follows.

Char → Short int → long int → float → double → long double

The conversion from lower data type to higher data type is done automatically and the values of the operands are not affected.

Rules for automatic type conversions

- (1) If one operand is declared as long double, the other operand gets converted to long double.
- (2) If one operand is declared as double, the other gets converted to double.
- (3) If one operand is declared as float, the other gets converted to float.

In case of Arithmetic operators

- (i) If both the operands are integers the result is integer
- (ii) If both the operands are real (float) the result will be real (float) correct to 6 places of decimals.
- (iii) If one operand is real and the other is integer, the integer operand gets converted to float and then the operation is executed and the result will be a real number correct to six places of decimals

Thank You



Introduction to algorithm & C Part-2



Ex.

	Float a	Outcome	Output
(i)	a = 5/3.0	1.666667	1.666667
(ii)	a = 5.0/3	1.666667	1.666667
(iii)	a = 5.0/3.0	1.6666667	1.666667

In the above example the integers 5 in (i) and 3 in (iii) get converted to float before the division takes place and since a is declared as float the output will be same as the outcome of the operation.

Converting data types from lower level to higher level is automatic. However converting data type from higher level to lower level is not automatic. This can be done explicitly and is called type cast.

The syntax of type cast is as follows (data type) expression.

Ex. float x = 12.345600

int y = int(x)

This will give the value of y as 12 since the decimal part will get truncated.

The same rule applies to arithmetic operators.

Ex.	int a	outcome	Output
(i)	a = 5/3	1	1
(ii)	a = 5/3.0	1.666667	1
(iii)	a = 5.0/3	1.666667	1
(iv)	a = 5.0/3.0	1.666667	1

In the above example (ii), (iii) and (iv) integers get converted to float and the outcome of the operations are real but since a has been declared as integer, the decimal part gets truncated in the output.

Illustrations

Indicate the output for the following programs

```
(i)    # include <stdio.h>
      main ()
      {
          int x = 10, y = 5;           Line 1
          x-- = 2;                     Line 2
          y+ = -- x;                   Line 3
          printf ("%d %d", x, y);
          return 0;
      }
```

Working	x	y
Line 1	10	5
Line 2	8	5
Line 3	7	12

Output 712

```
(ii)   # include <stdio.h>
      main ()
      {
          int a = 4, b = 3, c, d ;     Line 1
          c = 3 * a ++;                Line 2
          d = a + b * 2;               Line 3
          printf ("%d ", c);
          printf ("%d ", d);
          return 0;
      }
```

}

Working	a	b	c	d
Line 1	4	3		
Line 2	5	3	12	
Line 3	5	3	12	11

Output 12

11

```
(iii) #include <stdio.h>
      main ()
      {
          int a, b, h, k;
          a=12, b=5;
          h=(3*a+b)/2;
          a-- --b, b*=3;
          k=(4*a-b)/3
          printf ("%d ", h, k);
          return 0;
      }
```

Line 1

Line 2

Line 3

Line 4

Working	a	b	h	k
Line 1	12	5		
Line 2	12	5	20	
Line 3	8	12	20	
Line 4	8	12	20	6

Output 206

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

      main ()

      {

int a=6;           Line 1
float b=5.5, c;   Line 2
c=--a+b*3;       Line 3
print f ("%f", c);
return 0;

      }
```

Working	a	b	c
Line 1	6		
Line 2	6	5.5	
Line 3	5	5.5	20.500000

Output 20.500000

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

      main ()

      {

int a=4, b=7, c, d;   Line 1
c=a+b;               Line 2
(c>10)?20+c:d=40-c;  Line 3
print f ("%d", d);
return 0;

      }
```

Working	a	b	c	d	
Line 1	4	7			
Line 2	4	7	11		
Line 3	31	4	7	11	31

Output

Check your Progress

I) Answer in one or two sentences.

(i) Write the following statements in two different ways.

(a) Increase the value of x by 1

(b) Decrease the value of y by 2

(ii) What are unary operators? Give 2 examples.

(iii) What is the difference between = and == operators?

(iv) What is the difference between a/b and a%b?

(v) Write a statement to print ON if code = 1 otherwise print OFF using conditional operators.

II) Indicate the output for the following programs.

```
(i) #include <stdio.h>
main ()
{
    int a=10, b=4, c
    c = a+b;
    a++, b+=2;
    printf ("%d%d%d", a, b, c);
    return 0;
}
```

(Answer : 11614)

```
(ii) #include <stdio.h>
main ()
{
    int a = 1, b = 5, c = 4, d;
    c += a++;
    d = ++b * 3;
    printf ("%d%d", a, c);
    printf ("%d%d", b, d);
    return 0;
}
```

(Answer: 25
618)

```
(iii) #include <stdio.h>
main ()
{
    int x = 4, y = 5, z = 2;
    y -= z++;
    x *= 2;
    printf ("%d%d%d", x, y, z);
    return 0;
}
```

(Answer: 833)

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

```
int p = 5, q = 2, r = 7;
r += 4;
p *= 2, q--;
r += p * q;
printf ("%d%d%d", p, q, r);
return 0;
}
```

(Answer : 10121)

```
(v) #include <stdio.h>
main ()
{
    int l = 4, m = 3, n = 7, k;
    k = ++n;
    m += 2, l /= 2;
    k = m + n - l;
    printf ("%d", k);
    return 0;
}
```

(Answer : 10)

```
(vi) #include <stdio.h>
main ()
{
    int k = 8, l = 5;
    k -= l++;
    (k > l) ? printf ("%d", k); printf ("%d", l);
    printf ("%d", l);
    return 0;
}
```

(Answer : 6)

```
(vii) # include <stdio.h>
main ()
{
    int a = 100, b = 20, c, d;
    c = 5 * b --;
    d = a / 10 - 2;
    printf ("%d", b);
    printf ("%d", c);
    printf ("%d", d);
    return 0;
}
```

(Answer : 19

100

8)

```
(viii) # include <stdio.h>
main ()
{
    int pass = 0, fail = 0;
    int marks = 72;
    (marks > 60)? pass ++: fail ++;
    printf ("Pass = %d", pass);
    printf ("Fail = %d", fail);
    return 0;
}
```

(Answer : Pass = 1
Fail = 0)

```
(ix) # include <stdio.h>
main ()
{
    int a, b, h;
    a = 5, b = 8;
    h = (a+b) / 2;
    printf ("%d", h);
    return 0;
}
```

(Answer : 6)

```
(x) # include <stdio.h>
main ()
{
    int x = 5, y = 7;
    x += y--;
    printf ("%d%d", x, y);
    return 0;
}
```

(Answer : 126)



DATA INPUT AND OUTPUT FUNCTIONS

4.0 OBJECTIVES

At the end of this chapter, you will be able to input various types of data and obtain the output in a desired form.

4.1 INTRODUCTION

Any program execution consists of 3 steps :

- (i) input data for processing
- (ii) processing the data
- (iii) obtain the output in the desired format

This chapter covers steps (i) and (iii) i.e. you will learn how to input data and how to obtain the output.

For this purpose, there are certain functions called as library functions. These functions form a part of the C language.

There are two types of input / output functions.

- (i) Formatted Input / Output functions.
- (ii) Unformatted Input / Output functions.

4.2 FORMATTED INPUT / OUTPUT FUNCTIONS

The formatted input statement is the `scanf ()` function and the formatted output function is the `printf ()` function.

4.2.1 `scanf ()` function.

This function is used to input data to the computer with the help of the keyboard.

The syntax of the `scanf ()` function is as follows.

```
scanf ( "Control string", List of arguments separated by commas );
```

The control string consists of format specification which specifies the type of data to be input. It consists of the % sign and the conversion character.

The conversion characters are :

- c for single character
- d for single decimate integer
- ld for long integer
- o for octal notation
- x for hexadecimal notation
- f for float
- s for string of characters
- e for floating point value with exponential format
- g for %e or %f whichever is shorter

The control string is read from left to right and each format specification is assigned to the arguments from left to right. The number of format specifications must match with the number of arguments.

e.g. `scanf ("%d %f ", & x, & y);`

this indicates that the user has to input the values of the two variables x and y where x will take integer value and y will take floating point value.

In case of numeric and character type data, each argument is preceded by the ampersand & sign which is called as a pointer, as shown in the above example. However, in case of string data, the pointer is not used.

e.g. `scanf ("%s", name);`

4.2.2 Printf () function

The printf () function is used to obtain the output in a desired format. The syntax of the printf () function is as follows:

`printf ("Control String", list of arguments);`

The control string gives the format specification.

The conversion characters are the same as for the scanf () function.

However, in the format specifications, modifiers are used to obtain the output in a desired format. Such modifiers are also called as flags.

The modifiers used are as follows:

- + to insert the sign in the numeric data
- to print the output left justified. (By default output appears right justified 0)
- 0 to pad additional width by 0 in case of numeric data.
- w to specify the minimum width of the output.
- w.n to specify the width along with the number of decimal places.

These modifiers are placed between the % sign and the conversion character.

We shall understand the use of the control string with the help of the following examples:

1) int x = 1234;

.

Statement	Output
(i) printf ("%d", x);	1234
(ii) printf ("%6d", x);	bb 1234
(iii) printf ("%+5d", x);	+1234
(iv) printf ("%-6d", x);	1234 bb
(v) printf ("% 06d", x);	001234

Note : Blank spaces may be shown as ' ^ ' or

~~b~~

Explanation for the above 5 examples:

(i) No width specification hence output appears as it is.

(ii) width = 6. x consists of 4 digits, 2 blank spaces on the left hand side since by default data appears right justified.

(iii) Since x does not have a sign, it is treated as positive and format specification includes + sign. Hence output shows + sign on the left side of the value.

(iv) - sign indicates that the data is left justified. Hence blank spaces appear on the right hand side.

v) width is 6. x contains 4 digits, hence the 2 additional places are padded with zeros.

Whenever the output requires printing float type of data, number of decimal places can be specified using the format % w.nf. where w denotes the minimum width, n denotes the number of decimal places.

Example

Float x = 12.3456

Statement	Output
(i) printf ("%f" , x);	12.345600
(ii) printf ("%4.3f" , x);	12.346
(iii) printf ("%+3.2f" , x);	+12.35
(iv) printf ("%-6.1f" , x);	12.3 bb
(v) printf ("%9.4f" , x);	bb 12.3456

Explanation

- (i) %f indicates 6 decimal places hence 2 zeros on the right hand side
- (ii) no. of decimal places is less than the given value hence the 3rd decimal gets rounded off.
- (iii) Same as (ii)
- (iv) - sign given the output left justified hence 2 blank spaces appear on the right hand side
- (v) total width is 9 and 7 places have been consumed hence two blank spaces or the left hand side

Character type data uses the format % wc or % - wc

Example

char x = "A";

Statement	output
(i) Printf ("% 4C", x);	<i>bbb</i> A
(ii) Printf ("% - 4C", x);	A <i>bbb</i>

String type of output uses the format specification % ws, % w.ns and % - w.ns where w denotes the minimum width of the string and n denotes the first P characters to be printed.

Example

Char x (20) = " Mumbai University "

Statement	output
(i) printf ("%s" x);	Mumbai University
(ii) printf ("%12s" x);	Mumbai University
(iii) printf ("%20s " x);	bbb Mumbai
(iv) printf ("%15.6s " x);	bbbbbbbbb Mumbai
(v) printf ("%15.6s " x);	Mumbai bbbbbbbbb

4.3 UNFORMATTED INPUT OUTPUT FUNCTIONS

Unformatted Input output functions are used to input and output data without any specifications.

The unformatted input functions include getchar(), getch(), getche(), gets(), getc(), and the unformatted output functions include putchar(), putch(), putc(), puts()

We shall study the input functions and then the output functions.

4.3.1 getchar () function is used to input a single character with the help of the standard input device, the keyboard.

The syntax of the getchar() function is getchar();

When the getchar() function is used in a program, the user has to enter a character and press the enter key so that the character is stored and displayed on the screen. The getchar() function is linked to the header file stdio.h

4.3.2 getch() and getche() are used to input a single character with the help of the key board. When a character is input using getch() and getche() functions, there is no need to press the enter key after the character is input.

The difference between getch() and getche() function is that getche() echoes the character input, onto the screen. These two functions are linked to the header file conio.h

4.3.3 gets() function is used to input a string of characters. After the string has been input, the user has to press the enter key at the end of the string. This function is linked to the header file stdio.h

4.3.4 getc() function.

The getc() function is used to input a single character from a streamfile (istream). Stream indicates the flow of data. In case of input stream the data flows from the keyboard and in case of output stream the data flows to the screen.

The syntax of getc() function is: `getc (File * stream);`

This function uses the header file `stdio.h`

In case of an error or in case of end of the file, EOF is returned instead of the character.

4.4 PUTHCHAR () FUNCTION

`putchar()` is used to display a single character on the screen. This function uses the header file `stdio.h`

The syntax of this function is: `putchar();`

4.4.1 puts () function

puts() function is used to display a string of characters.

The syntax of this function is puts();

4.4.2 putchar() function

putchar() function is used to send a character to a stream file (ostream).

The syntax is: putchar(c, file * stream);

This function uses the headerfile stdio.h

In case an error is detected, EOF is displayed.

Examples

(i) # include <stdio.h>

main ()

{char k;

k = getchar();

putchar(k);

return 0;

}

Input k = 'A'; press Enter

Output A

(ii) # include <conio.h>

{ char k;

```
k = getch();  
putch(k);  
return 0;
```

Input 'A' (No need to press enter key)

Output A

```
(iii) # include <conio.h>  
{char k;}  
k= getch();  
putch(k);  
return 0;  
}
```

Input 'K' (No need of enter key. But value is echoed on screen)

Output K

```
(iv) # include <stdio.h>  
{char city [6];  
gets(city);  
puts (city);  
return 0;  
}
```

Input "DELHI" <Enter> (Will appear on screen)

Output DELHI

Check your Progress

1. Answer in one or two sentences.

(i) What is the use of the following functions?

(a) getchar()

(b) getch()

(c) gets()

(d) puts()

(ii) Give the difference between :

(a) getchar() and getch()

(b) getch() and getche()

(c) putchar() and puts()

2 Indicate the output for the following programs

(i) # include <stdio.h>

main()

{ int x = 12, y = -27;

float a = 24.92, b = 675.378;

printf ("%4d % - 6d \n", x, y);

printf ("% 6.2 f % 12.1 f", a, b);

return 0;

}

Output:

bb 12 - 27 *bbb*

b 24.92

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

main()

{char x = "P", int a = +27, float b = 16.27;

printf("%4c%-4d", x, a);
printf("\n%-5c%12.4f", x, b);

return 0;

}
```

Output: *bbbP+2.7b*

Pbbbbbbb16.2700

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

main()

{ int x = - 37, float y = 200.92;

printf ( "%d %f", x, y);

printf ( "\n % -7d % -7d % .3f", x, y );

return 0;

}
```

Output - 37200.920000

-37 ~~bbb~~ 200.920

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

main ()

{ char x = 'a', float y = 12.5;
```

```
int z = 16;  
printf ( "%-3c% 4.4f %05d", x,y,z );  
printf ( "\n% 4c% 6.0f% -+4d", x,y,z );  
return 0;  
}
```

Output:

```
ab12.500000016  
bbabbbb13+16b
```

Q3Q3Q3

Control Statements for Decision Making

Unit Structure

1. Objectives
2. Introduction
3. Branching Statements
4. Switch Statements
5. Looping Statements
 1. While loop
 2. do loop
 3. For Statement
6. Jump Statements
7. goto.....Label Statements
8. Unit End Exercises

5.0 OBJECTIVE

After going through this chapter you will be able to:

- (i) alter the sequence of the execution of the program
- (ii) set up loops to repeat a set of statements, desired number of times.
- (iii) transfer control to different statements in the program.

5.1 INTRODUCTION

By default a program in C is executed from the main function from top to bottom. However, the program execution can be altered with the help of control statements i.e. the control statements control the flow of execution of the program.

Control statements can be classified into 3 types.

- (i) Branching statements
- (ii) Looping statements
- (iii) Jump statements

5.2 BRANCHING STATEMENTS

Branching statements are used to check a test condition and depending upon the outcome of the test condition, branch to the corresponding statement. The Branching statements include:

- (i) If Statement
- (ii) Switch Statement

5.2.1 If Statement

As mentioned earlier, if statement allows execution of a set of statements depending upon the outcome of the test condition.

If Statement has 2 formats as follows:

Format I

```
if (test condition) statement 1,  
  
next statement;
```

In this format, the computer checks the test condition and if the condition is true, statement 1 gets executed and then the next statement is executed. If the test condition is false, statement 1 is ignored and directly the next statement is executed.

Example

```
# include <stdio.h>  
  
main()  
{ int x;  
scanf("%d", &x),  
if (x > 10) x += 5;  
printf ("%d", x );  
return 0;  
}
```

In the above example, the user has to input the value of x. If this value is greater than 10, x increases by 5 and then its value gets printed. However if $x \leq 10$, the value of x does not change before printing.

Format II

```
if (Test condition) statement 1;  
  
else statement 2;  
  
next statement
```

In this format, the test condition is checked and if it is true, statement 1 is executed and then the next statement is executed and if the test condition is false, statement 2 is executed and then the next statement is executed i.e. in the if / else format, 2 alternative statements are provided, one, if the condition is true and another if the condition is false.

Example.

```
# include <stdio.h>

main()

{ int marks;
scanf ( "%d", &marks );

if (marks >= 35)

printf ("PASS" );

else

printf( "FAIL" );

return 0;

}
```

In the above example, if the marks input are more than or equal to 35 PASS is printed otherwise FAIL is printed.

5.2.2 Nested if statement

Sometimes one if statement may be embedded in another if statement. Such a statement is called as the Nested if statement.

The format of the nested if statement is as follows:

```
if (test condition 1)

statement 1;

else

if (test condition 2)

statement 2;

else statement 3;
```

In the above format, test condition 1 is checked and if it is true statement 1 is executed otherwise test condition 2 is checked. If true, statement 2 is executed otherwise statement 3 is executed.

Example

Calculate the commission earned by a salesman according to the following.

Sales Rate

First Rs. 5000	nil
Next Rs. 3000	6%
Next Rs. 2000	8%
Exceeding Rs. 10000	10%

Solution:

```
# include <stdio.h>

main()
{ float sales, com;

scanf ( "%f", &sales );

if (sales <= 5000 ) com
= 0;

else

if (sales >= 8000)
com = (sales - 5000) *0.06;

else

if (sales <= 10000)
com = 180 + (Sales - 8000) *0.08;

else

com = 180 + 160 + (Sales - 10000) *0.1;
```

```
print f ( "Com = Rs.% .2 f ", com );  
return 0;
```

5.3 SWITCH STATEMENT

If and if / else statements can be used if a test condition gives only 2 outcomes true or false. However, there arise situations where there may be more than 2 outcomes for a test condition. In such cases, switch statement is used.

The syntax of the switch statement is as follows:

```
switch (expression)  
{ case constant 1 : Statement 1 ; break;  
  case constant 2 : statement 2; break;  
  case constant 3 : statement 3; break;  
  
  default : statement K; }
```

The switch statement is executed as follows:

The expression following switch is evaluated. This value is matched with the various case constants and when a match is found, the corresponding statement is executed.

Each case statement is followed by the break statement.

The break statement breaks the loop, i.e. the remaining statements are ignored by the computer and the control is transferred out of the loop.

Default statement -

Inside the switch() loop is the default statement. This statement is executed when none of the case constants match with the expression.

Note that the default statement does not require a break statement since it is the last statement in the loop.

Example

```
# include <stdio.h>

main ()

{ int marks;

char grade;

scanf ( "% d", &marks )

switch (Marks / 10)

{ case 9 : grade = 'A'; break;

case 8 : grade = 'B'; break;

case 7 : grade = 'C'; break;

case 6 : grade = 'D'; break;

default : grade = 'F';

}

printf ( "Grade = %c" grade );

return 0;

}
```

In the above example if the marks entered are 84, marks / 10 = 8 and hence grade will be B and if no match is found grade will be F.

5.4 LOOPING STATEMENTS

Sometimes a set of statements has to be executed repeatedly. In such cases, looping statements are used to instruct the computer to repeat the statements within the loop, desired number of times.

The loop statements include

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

5.4.1 While loop

While loop is used to execute a set of Statements called the body, repeatedly as long as the test condition is true.

The syntax of the while loop is as follows

```
while (test condition)
{
set of statements or body.
```

The while loop is executed as follows.

The test condition is checked and if the condition is true the body of the loop is executed and the control is transferred back to the test condition. This process is repeated as long as the test condition is true and when the test condition is false, the control is transferred to the next statement after the loop.

Example Calculate and print the sum

1+2+3+.....+50 using while loop

```
# include < stdio.h >

main ()

{

int i=1, S=0;

while ( i <= 50)

{s += i ;

i ++ ;

}

printf ( "\n sum = %d", s );

return 0;

}
```

5.4.2 do - while loop

The do-while loop is used to execute a set of statements repeatedly as long as the test condition is true.

The Syntax of the do-while loop is as follows.

```
do

{

} while (test condition);
```

The do-while loop is executed as follows. The body of the loop will be executed once and then the test condition will be checked. If it is true, the control is transferred back to the loop. This process continues as long as the test condition is true and when the test condition is false, the control is transferred to the next statement after the loop.

Example

Calculate and print the sum $25^2 + 23^2 + \dots + 1^2$ using the do-while

loop

Solution:

```
# include <stdio.h>

main()
{ int i = 25, s=0;
  do
  {s + = i*i;
   i -=2;
  } while (i >= 1);
  printf ( "sum=%d", s );
  return 0;
}
```

Difference between while and do-while loop statements.

(i) In the while loop, the test condition is placed before the body of the loop and if the test condition is not true, the body of the loop will not be executed even once.

(ii) In the case of the do-while loop statements the test condition is placed after the body of the loop. Hence, the body of the loop is executed at least once.

5.4.3 For Statement

The for statement is a loop statement which allows the user to execute a set of statements repeatedly, desired number of times.

The format of the for statement is as follows.

```
for (exp1; exp2; exp3)
{
    body
}
```

exp 1 is used to initialise the variable

exp 2 is the test condition for terminating the loop

exp 3 Prepares for the next iteration.

The for loop is executed as follows exp 1 is executed once at the beginning of the loop where the initial value is assigned to the variable.

The body of the loop is executed and at the end of the loop the control is transferred to exp 3 where the value of the variable gets altered.

The control is then transferred to exp. 2 where the test condition is checked if the condition is true, the body of the loop gets executed.

This process continues as long as the test condition is true and when the test condition is false, the loop gets terminated and control is transferred to the next statement after the loop.

Example

Write a program to calculate and print $S = 2+4+6+8+\dots+40$

Solution:

```
# include < stdio.h >

main ()

{int i, s=0;

for (i=2; i<=40; i + = 2)

{ s + = i;

}

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

return 0;

}
```

5.5 JUMP STATEMENT

Jump statements are used to exit a loop and transfer control to different statements in the program.

The jump statements include

- (i) break statement
- (ii) continue statement
- (iii) goto label statement

5.5.1 break statement

The break statement is used to exit from the while, do-while, for and switch statements and transfer control to the statement following the loop statement.

If the break statement is used with a nested loop, it only exits the innermost loop and control is transferred to the next statement outside the loop.

Example Indicate the output for the following program

```
# include <stdio.h>

main()

{ int i ; s = 0;

for ( i = 4; i < 7; i + +)

{ if (i % 2! = 0) break;

printf ( "% - 4d", i );

}

return 0;

}
```

Output: 4 ~~bbb~~

5.5.2 continue statement

The continue statement is used with the while, do-while and for statements.

Whenever the continue statement is given in the loop, it breaks the loop and transfers control to the next iteration of the loop.

In case of the while and do-while loop, the continue statement causes the control to transfer to the test condition and then continues the execution as a normal loop statement.

In case of the for loop, the continue statement causes the control to be transferred to the next iteration and then to the test condition, after which the execution continues like a normal for loop.

Example Indicate the output for the following program.

```
# include <stdio.h>

main ()

{int i;
for ( i = 1; i < 8; i + +)
if (i % 2! = 0)
continue;

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

return 0;

}
```

Output will be

```
2
4
6
```

5.6 GOTO..... LABEL STATEMENTS

The goto label statement is a jump statement. It causes the computer to locate the identifier label and execute the statement following the label.

The syntax of the goto.....label is as follows.

```
goto LABEL;  
  
.....  
  
.....  
  
LABEL : statement;
```

Example

```
# include <stdio.h>  
  
{ int age;  
scanf ( "%d", &age );  
if (age>60) goto remark; printf  
( "Not a senior citizen");  
return 0;  
  
}
```

The label name follows the rules of identifiers and is followed by a colon.

Rules about goto label statements:

(i) Goto... label should not be used in the middle of the body of the loop statement

(ii) Goto... label should not be used in the middle of the switch statement.

Some standard programs:

1) Write a program to calculate and print

$$S = 2^2 + 5^2 + 8^2 + \dots + 50^2$$

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

```
main()
```

```
{int i, s;
```

```
s=0;
```

```
for ( i = 2; i <= 50; i + = 3)
```

```
{s+ = i* j;}
```

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

```
return 0;
```

```
}
```

2) Write a program to calculate and print

$$S = 2 \times 3 + 4 \times 5 + 6 \times 7 + \dots + 20 \times 21$$

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

```
main()
```

```
{ int i, j, s;
```

```
s=0;
```

```
for ( i = 2, j=3; i <= 20; i + = 2, j + =2)
```

```

{s+= i* j};

printf ( "sum=%d", s );

return 0;

}

```

3) Write a program to input Sales and calculate and print the sales tax as follows

Sales	Tax Rate
First Rs. 5000	4%
Next Rs. 3000	6%
Next Rs. 2000	10%
Exceeding Rs. 10000	12%

```

# include <stdio.h>

main()

{ float sales; tax;

scanf ( "%f", & sales );

if (sales <= 5000 )

tax = sales * 0.04;

else

if (sales <= 8000)

tax = 200 +(sales - 5000) *0.06;

else

if (sales < 10000)

tax = 200 + 180 + (sales - 8000) *0.1;

else

tax = 200 + 180 + 200 (sales - 10000) *0.01;

```

```
printf ( "Sales = Rs.% .2 f \n ", sales );
printf ( "tax = Rs.% .2 f \n ", tax );
return 0;
}
```

4) Write a program to calculate and print the greatest common divisor between 2 positive integers

```
# include <stdio.h>
main()
{ int. x, y;
scanf ( " %d %d ", & x, & y);
While (x != y )
{ if (x > y)
x = x - y;
(else y = y - x;
}
printf ( "GCD = %d ", x );
return 0;
}
```

5) Write a program to input 10 values of x and calculate and print the arithmetic mean and the standard deviation. Where mean

$$= \frac{\sum x}{N}, sd = \sqrt{\frac{\sum x^2}{N} - (mean)^2}$$

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

```
main ( )
{int i, x, s, s1;

float mean, sd;

s=0, s1 = 0;

for (i = 1; i <= 10; i ++ )
{ scanf ( " %d ", & x );

s + = x; s1+ = x * x;

}

mean = s/10;
sd = sqrt (S1 / 10 - mean * mean); printf
( "Arithmetic mean = %f" mean ); printf
( "\n std deviation = %f ", sd ); return 0;

}
```

6) Convert the following if statements into switch statements

```
if (code == 1) printf ( "Doctor" );

else

if (code == 2) printf ( "Engineer" );

else

if (code == 3) printf ( "Banker" );

else

printf ( "Teacher" );
```

The corresponding switch statements will be

```
switch (Code)

{ case 1 : printf ( "Doctor" ); break;
```

```
case 2 : printf ( "Engineer" ); break;
case 3 : printf ( "Banker" ); break;
default : printf( "Teacher" );
}
```

Indicate the output for the following programs

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

main()
{int i=1;
while (i < 5)
{printf ( "\n 4d", i);
i + = 2 ; }

return 0;}

}
```

Output will be

~~bbb~~ 1

~~bbb~~ 3

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

main()

int i = 10, s = 0;

do
{ s + = 2 * i;
i - = 3;
```

```
} while (i > 3);  
  
printf ( "\ns=%d", s);  
  
return 0;  
  
}
```

Output will be as follows.

S=42

```
(iii) # include <stdio.h>  
  
main ()  
  
{ int i;  
  
for (i = 2; i < 7; i ++)  
{ if (i%2 == 0) continue;  
printf ( "% 5d", i);  
  
}  
  
return 0;  
  
}
```

Output will be as follows

Bbbb3bbbb5

```
iv) # include <stdio.h>  
  
main ()  
  
{ int i; s=0;  
  
for (i = 5; i < 12; i +=2)  
{ if (i%3 == 0) break;  
printf ( "%d", i );  
  
s + = i ;
```

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

return 0;

}
```

Output will be as follows

57

12

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

{ int i = 8;

do

{printf ( "%d", i);

i /= 2 ;

} while (i > 3);

return 0;

}
```

Output will be as follows

84

5.7 Unit End Exercises

1. Write a program to calculate and print

$S = 25 \times 24 + 24 \times 23 + \dots + 2 \times 1$ using do - while loop

2. Write a program to print all integers between 75 and 125 which are divisible by 8, using the while loop.

3. Write a program to calculate and print

$S = 50^2 + 49^2 + \dots + 20^2$ using for loop

4. Write a program to input the annual income and calculate the income tax as follows.

Income	Tax
Upto Rs. 1,00,000	Nil
Next Rs. 1,50,000	10%
Next Rs. 1,00,000	20%
Excess	30%

5. Write a program to input the number of calls made by the subscriber and calculate and print the telephone bill as follows:

First 60 calls	Free
Next 100 calls	@80p / call
Next 100 calls	@Re 1 / call
Excess	Rs 1.20 / call

Print the name of subscriber, telephone no., no of calls and bill amount.

6. Answer in one or two sentences.

(i) What is the use of the if statement?

(ii) What is the use of:

- a) switch statement?
- b) for statement
- c) while statement

d) do-while statement.

(iii) What is the use of:

a) break statement

b) default statement

c) continue statement

7. Give one point of difference between:

(i) while and do-while statements.

(ii) break and continue statements.

8. State whether the following statements are true or false:

(i) Every switch statement must have a default statement.

(ii) The do-while loop is executed at least once in a program

(iii) The for loop is executed at least once in a program.

9. Convert the following if statements into switch statement

```
if (code == 'M') printf ( "Manager" );
```

```
else
```

```
if (code == 'O') printf ( "Officer" );
```

```
else
```

```
if (code == 'C') printf ( "Clerk" );
```

```
else
```

```
if printf ( "peon" );
```

10. Indicate the output for the following programs:

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

    main ()

    {a=5, b=3;

    do

    { a + = b;

    b + + ;}

    while (a<15);

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

    return 0;

    }
```

Ans. Output will be 176

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

    main ()

    { int a = 100, b=10;

    while (a > 60)

    { a = a - b;

    b + = 5;

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

    return 0;

    }
```

Ans. Output will be 9015
7520

5525

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

      main()

      { int x = 10, i ;

        for (i=5; i<=10; i++)

          {if (x %2 == 0)

            x ++;

           else

            X -- ;

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

           }

        return 0;
```

Ans. Output will be 11

10

11

10

11

10

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

      main()

      {int i;

        for (i = 5; i < 10; i++)

          { (i%3)!= 0)

            continue;
```

```
printf ( "%d\n", i );  
}  
return 0;  
}
```

Ans. Output will be 6

9

V) # include <Stdio>

```
main ()  
{int i = 10, k = 7;  
while (i < 20)  
K + i ;  
If (k > 35) break ;  
i + 3; }  
printf ( "%-4d", k );  
→ return 0;  
}
```

Ans. Output will be 46 ~~bb~~

❧❧❧❧

Thank You



Introduction to algorithm&

C

Part-3



ARRAYS AND STRINGS

Unit Structure

1. Objectives
 2. Introduction
 3. What Are Arrays?
 4. One-Dimensional Arrays
 5. Declaring One-Dimensional Arrays
 6. Initialization Of One-Dimensional Arrays
 7. Two-Dimensional Arrays
 8. Initialization of Two-Dimensional Arrays
 9. Unit End Exercises
-

6.0 OBJECTIVES

After going through this unit you will be able to:

1. Understand what are arrays.
2. What is the need for arrays
3. How arrays can be used in C Language
4. Declare and use one dimensional and two dimensional arrays
5. Understand the need for character and string variables
6. Declare and use character and string variables
7. Use functions to handle character and string data

6.1 INTRODUCTION

In the previous chapters we have used some basic data types like int, float, double and char type. All these data types have a basic limitation and that is they can store only one value **at a time**. Due to this inherent limitation they can store only small amount of data and can be used in applications which do not require large volumes of data. In real life applications it is more likely that we will encounter larger amounts of data. It is important and interesting to note that one variable of any type can actually handle a large amount of data **but not simultaneously**. For e.g. if we wish to find the average of hundred numbers we do not need hundred variables to store them as we can process them one at a time. The problem would arise if we wanted all of them at the same time. What are the situations in which this would be necessary? We will discuss some of the applications in this chapter which need simultaneous storage of large sets of numbers or names.

To process large amounts of data we need a special data type that would efficiently store data and allow the user to access and process the data efficiently. C Language meets the need of the user to support large sets of data by introducing a data structure called **array**.

6.2 WHAT ARE ARRAYS?

An array is a fixed size collection of elements all of the same primary data type. It can be thought of as group of similar data items which are referred by a common name. In it's most basic form an array can be used to represent a list of entities such as names or roll numbers of students or item numbers. Some typical examples could include

1. List of names of students
2. List of marks scored by students
3. Inventory list in a store

4. List of subscribers of a magazine
5. Monthly sales of a particular year
6. Telephone directory of a city

Since an array represents a structured related set of items such as the examples given above indicate it is classified as a **Data Structure** in C language. There are other data structures in C such as lists, trees and queues some of which will be discussed in later chapters. Presuming that there are 50 students whose marks are to be stored an array can be defined to represent the data as below.

marks[50]

Each item in this list is called an element of the array. The individual elements stored in the arrays are stored sequentially with the first array element stored in the first reserved location and the second element in the second reserved position and so on. Arrays are particularly useful when a large number of values are required to be stored in separate variables such as in sorting. Arrays are also known as **subscripted variables**.

An individual element in an array can be accessed by giving the name of the array and the position of the item in the array. The elements position is called its **index** and **subscript** value. For e.g. marks[0] refers to the first element in the array marks, marks[1] refers to the second value here the name of the array is marks and the index values are 0,1, etc. It must be noted that the index can be an integer constant, variable or expression that is greater than or equal to 0.

The facility to use a single name to represent a collection of items and the ability to access any individual member by specifying the index allows the user to develop compact and efficient programs. Arrays can be used to represent simple lists of the type mentioned above and they can also be used to represent tables with rows and columns. Arrays which represent such lists which are two dimensional (rows and columns) or more are called multi-dimensional arrays. We will first understand one dimensional arrays and then proceed to two or more dimensions.

6.3 ONE-DIMENSIONAL ARRAYS

A list of items which is given one common variable name and comprising of only one subscript is called a single subscripted variable or a one-dimensional array. The concept of a subscript is not new and we have used it in mathematics whenever we represent a sequence of numbers such as x_1, x_2, x_3, \dots and so on to represent a set of related numbers. In C the subscripted variable is expressed as $x[1], x[2], x[3], \dots$. It is important to note that the subscript can begin with zero.

For example if we wish to store the marks scored by five students 56,78,90,86,43 in an array we could do it as follows

```
int marks [5] ;
```

The above statements declares to the computer that there will be an array by the name marks and size 5 and its individual elements will be marks[0], marks[1], marks[2], marks[3], and marks[4]. It is important for a beginner not to confuse the index 0,1,2,3,4 with the actual values. The values to the array elements (which are individual variables in their own right) can be assigned in a number of ways. One way to do that is

```
marks [0] = 56;
```

```
marks [1] = 78;
```

```
marks [2] = 90;
```

```
marks [3] = 86;
```

```
marks [4] = 43;
```

It is also possible to give values together at one time as follows.

```
int marks[5] = {45, 56, 78, 98, 65}
```

or

```
int marks[ ] = {45, 56, 78, 98, 65}
```

The subscripts of an array can be integer constants, integer variables or expressions that give integers. C on its own doesn't check the upper or lower limit of an array, (they are also called bounds of an array), so if we have declared an array by writing `int marks[5]` , then the index cannot be a number except 0,1,2,3 and 4. It is the responsibility of the programmer to ensure that these bounds are not violated.

6.4 DECLARING ONE-DIMENSIONAL ARRAYS

Much like the other variables in C language an array also needs to be declared before it is used so that the C compiler or interpreter can allocate space for them in memory. The format for declaring an array is

```
type variable-name [size];
```

Here the type specifies the type of element that will be contained in the array, and this can be any of the basic types of C such as int, float, double or char. The size indicates the maximum number of elements that can be stored in the array and is generally an integer number or a symbolic constant with an integer value.

For e.g. `float temp[20];` indicates that temp is an array of type float and the maximum number of elements it can store is 20. The index or the subscript can be any number from 0 to 19.

In C language character strings are treated as if they are arrays of characters with each element of array storing one character from the string.

For e.g. `char cityname [10];` will declare `cityname` as an array comprising of maximum 10 characters. If we assign the name "MUMBAI" to this array, then it is stored in the memory as follows

'M'
'U'
'M'
'B'
'A'
'I'
'\0'

Observe that the string is terminated with the null character '\0'. So it is always necessary to provide for one extra space whenever size is mentioned while declaring arrays of strings.

Example of a program to read an array of ten integer numbers and find the average.

```
main()
```

```
{ int i;
```

```
  int x[10], y, sum =0;
```

```
/* Entering numbers and storing them in an array */

for ( i = 0 ; i < 10 ; i++ )
{
    printf("Enter ten numbers one at a time");
    scanf( "%d", &y);

    x [i] = y;
    sum = sum + x[i];
}

/* Print the values of the element sof the array and their sums */

printf(" The numbers are \n");
for ( i = 0 ; i < 10 ; i++ )
{
    printf("%d\n", x[i]);
}

printf("\nThe sum of all the elements of the arrays is = %d", sum);

}
```

If we type the ten numbers as 10,15,20,25,30,35,40,45,50,55 the output of the above program will be

The numbers are

10

15

20

25

30

35

40

45

50

55

The sum of all the elements of the arrays is = 325

6.5 INITIALIZATION OF ONE-DIMENSIONAL ARRAYS

Once an array is declared it must be supplied with initial values. Since arrays like other variables in C language are memory locations there is a possibility that they may contain some initial garbage values. The process of supplying the values to an array variable is called initialization.

Initializing an array can be done in two ways. Either at Compile time or Run time.

Initialization at Compilation stage

Like other variables in C language an array can be initialized when they are being declared. It can be done as follows

type variable-name [size] = { list of Values };

The values in the list are to be separated by commas.

For e.g. int roll[5] = {1,2,3,4,5};

Will declare an array of size 5 and will assign number 1 to the first element that is roll[0], number 2 to roll[1] and so on respectively.

It is not necessary that all elements have to be assigned values.

For e.g. if we write

int x[10] = {1,2,3};

then the first three elements will be given values 1,2 and 3 respectively and all the remaining elements will get the value zero. If the array is of type character then the remaining values will be NULL.

Character arrays can also be initialized in the same way by specifying the array type and assigning it to a string.

Initialization at run-time stage

Arrays can also be initialized at the run time stage by specifying their values in a for or while loop particularly when the array is large.

For e.g.

```
int x[50];  
  
for (i =1; i < 50; i++)  
{  
x[i] = 0;  
}
```

Will initialize all the values of the array x to 0.

It is also possible to input values directly into an array by using a keyboard input function such as scanf().

```
For e.g. int x[2];  
  
scanf("%d%d", &x[1],&x[2]);
```

will give the values supplied by the user to the two elements of the array x.

Based on the matter presented the student should be able to solve some simple problems using one dimensional arrays.

6.6 TWO DIMENSIONAL ARRAYS

In the section above we discussed array variables which were in the form of a single dimension list. However there can be situations where we may want to store data which is the form of a table. A table has rows and columns much like a matrix.

Consider the following table which shows region wise sales of four products. There are four regions and four products whose sales figures are represented in the table below

Regions	Products			
	Product1	Product1	Product1	Product1
East	300	400	250	560
West	500	250	375	800
North	175	245	400	200
South	378	200	120	560

Each row in the above table represents the sales of that region and each column represents the sales of a particular product.

The table can also be seen as a 4 x 4 matrix with four rows and four columns.

In matrix terminology the value 300 in the first column and first row would be represented by using two subscripts X_{11} where the first subscript represents the row and the second subscript represents the column. In general the values would be represented by X_{ij} where i represents the row i and j represents the column.

In the C language it is possible to define the table given above by using two dimensional arrays. Two dimensional array to represent the above table would be denoted as $X[4][4]$.

The general format is

```
type array_name [row size][col size] ;
```

It is to be noted that the declaration cannot be done as X[4,4] which is the convention in many other computer languages.

Like single dimensional arrays the index of both the rows and columns would begin with the number 0. So for a two dimensional array X[4][4] the arrays variables would begin from X[0][0] and would go on till X[3][3]. In the context of the above table the value 300 would be stored in X[0][0] and the value 500 in the variable X[0][1].

We now illustrate the use of two dimensional arrays as follows. We will write a C language program which reads in the value of the above two dimensional table and calculate region wise and product wise sales and also find the Total Sales of the organization.

```
/* C language Program to Calculate Region Wise Product wise Sales using two dimensional arrays */
```

```
#define REGIONS      4

#define PRODUCTS 4

main()
{
    int x[REGIONS][PRODUCTS] ;
    int regtot[REGIONS], prodtot[PRODUCTS];
    int i,j, grandtot;

    /* Reading the table into the two dimensional array */
    printf( "Enter the Sales data region wise( row wise) one at a time \n");
```

```
for ( i = 0; i < REGIONS; i++)

{

    regtot[i] =0;

    for ( j = 0 ; j < PRODUCTS; j++)

        {

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

            regtot[i]= regtot[i] + x[i][j];

        }

    grandtot = grandtot + regtot[i];

}

/* Find Product Wise Sales */

for ( j = 0; j < PRODUCTS ; i++)

{

    prodtot[j] =0;

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

        {

            prodtot[i]= prodtot[i] + x[i][j];

        }

}
```

```
/* Printing all totals */

printf(" Region Totals \n\n");

for ( i = 0; i < REGIONS; i++)
    {
        printf("Region [%d] = %d\n", i+1, regtot[i]);
    }

printf(" Product Totals \n\n");

for ( j = 0; j < PRODUCTS ; j++)
    {
        printf("Product [%d] = %d\n", j+1, prodtot[j]);
    }

printf("\n Grand Total = %d\n", grandtot);

}
```

6.7 INITIALIZATION OF TWO DIMENSIONAL ARRAYS

Two dimensional arrays can also be initialized by following their declaration by a list of values enclosed in braces.

For e.g `int x[2][3] = {1,1,1,2,2,2};`

will initialize the first elements of the first row by the value 1 and the second row will get the values 2. The initialization is done row wise. It can also be done by explicitly mentioning each rows within the braces as follows

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

the above statement will have the same effect as the first one.

The array can also be initialized by mentioning the values in the matrix format as follows

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

6.8 UNIT END ECERCISES

1. Declare an array of size 15 which will hold integer values
2. Declare an array of size 5 and give values 10,12,14,16,18 to the elements
3. Declare an array of size 15 which will hold the name of a person
4. Declare an array of size 10 and initialize all its elements to 0
5. Declare an array of size 20 that can hold fractional numbers

6. Allow the user to enter 10 numbers, store them in an array and print them.
7. Allow the user to enter 10 numbers and find their average.
8. Allow the user to enter 15 numbers and print them in the reverse order.
9. Allow the user to enter 10 numbers, find the average and then print those numbers that are above average.
10. Input ten distinct numbers and store them in an array, find the Highest and Lowest.
11. Input the list of marks scored by a class of 50 students and generate a frequency table with intervals 0-10,11-20,21-30 and so on up to 91-100.
12. Input the Roll number and marks of 10 students and store them in two separate arrays. Ask the user to enter a particular roll number and display the marks of that student.
13. Input the Roll number and marks of 10 students and store them in two separate arrays. Display the Roll number and marks of the student getting highest marks.



SORTING AND MERGING

Unit Structure

1. Objectives
2. Introduction
3. Bubble Sort
4. Selection Sort
5. Insertion Sort
6. Heap Sort
7. Merge Sort
8. Algorithmic Efficiency
9. Analyzing Algorithms
- 7.9 Algorithmic Efficiency -- Various Orders And Examples

7.10 Unit End Exercises

7.0 OBJECTIVES

After going through this chapter you will be able to:

- Understand the Purpose of Sorting
 - Understand the different methods of Sorting.
 - Identify the advantages of different algorithms of Sorting
 - Be able to write programs in C to implement the algorithms for Sorting
 - Explain what is meant by Efficiency of an algorithm
 - Compare algorithms for Efficiency
-

7.1 INTRODUCTION

Sorting in general refers to the process of arranging or ordering things based on criteria (numerical, chronological, alphabetical, hierarchical etc.). In computer applications sorting is of immense importance and is one of the most extensively researched subjects. One of the main reasons for data to be sorted is to allow fast access of the data and given the huge volume of data it becomes inevitable to have fast machines and fast access to the data. A simple requirement such as finding somebody's name in a telephone directory requires that the telephone directory be sorted in alphabetical order for easy access. It is one of the most fundamental algorithmic problems as the data involved can increase and the programmer has to look at various alternatives to improve speed and better memory utilization. Sorting is also fundamental to many other fundamental algorithmic problems such as search algorithms, merge algorithms etc. It is estimated that around 25% of all CPU cycles are used to sort data. There are many approaches to sorting data and each has its

own merits and demerits. In this section we discuss some of the common sorting algorithms.

7.2 BUBBLE SORT

Bubble Sort is probably one of the oldest, easiest, straight-forward, sorting algorithms. However as we shall see Bubble sort is not very efficient. Bubble Sort works by comparing each element of the list with the element next to it and swapping them if required. With each pass, the largest or the smallest of the list is "bubbled" to the end of the list whereas the smaller/larger values sink to the bottom. It is similar to selection sort although not as straight forward. Instead of "selecting" maximum values, they are bubbled to a part of the list. We will now look at the C language program which implements the above algorithm.

```
void BubbleSort(int a[], int array_size)
{
    int i, j, temp;
    for (i = 0; i < (array_size - 1); ++i)
    {
        for (j = 0; j < array_size - 1 - i; ++j )
        {
            if (a[j] > a[j+1])
            {
                temp = a[j+1];
                a[j+1] = a[j];
                a[j] = temp;
            }
        }
    }
}
```

A single, complete "bubble step" is the step in which a maximum element is bubbled to its correct position. This is handled by the inner for loop.

```
for (j = 0; j < array_size - 1 - i; ++j )
{
    if (a[j] > a[j+1])
    {
        temp = a[j+1];
        a[j+1] = a[j];
        a[j] = temp;
    }
}
```

}

Examine the following table. (Note that each pass represents the status of the array after the completion of the inner for loop, except for pass 0, which represents the array as it was passed to the function for sorting)

```

8 6 10 3 1 2 5 4 } pass 0 ( This is the Unsorted array )
6 8 3 1 2 5 4 10 } pass 1 ( largest element goes down)
6   3   1   2   5   4   8   10   }   pass   2
3   1   2   5   4   6   8   10   }   pass   3
1   2   3   4   5   6   8   10   }   pass   4
1   2   3   4   5   6   8   10   }   pass   5
1   2   3   4   5   6   8   10   }   pass   6
1 2 3 4 5 6 8 10 } pass 7

```

The above tabulation clearly depicts how each bubble sort works. Note that each pass results in one number being bubbled to the end of the list. Bubble sorting is a simple sorting technique in sorting algorithm. In bubble sorting algorithm, we arrange the elements of the list by forming pairs of adjacent elements. It means we repeatedly step through the list which we want to sort, compare two items at a time and swap them if they are not in the right order. Another way to visualize the bubble sort algorithm is as its name, the smaller element bubble to the top.

7.3 SELECTION SORT

Now we consider the Selection Sort. It basically determines the minimum (or maximum) of the list and swaps it with the element at the index where it is supposed to be. The process is repeated such that the nth minimum (or maximum) element is swapped with the element at the n-1th index of the list. The below is an implementation of the algorithm in C.

```

void SelectionSort(int a[], int array_size)
{
    int i;
    for (i = 0; i < array_size - 1; ++i)
    {
        int j, min, temp;
        min = i;
        for (j = i+1; j < array_size; ++j)
        {
            if (a[j] < a[min])
                min = j;
        }
        temp = a[i];

```

```

        a[i] = a[min];
        a[min] = temp;
    }
}

```

Consider the following table. (Note that each pass represents the status of the array after the completion of the inner for loop, except for pass 0, which represents the array as it was passed to the function for sorting)

8	6	1	0	3	1	2	5	4	} pass 0
1	6	1	0	3	8	2	5	4	} pass 1
1	2	1	0	3	8	6	5	4	} pass 2
1	2	3	1	0	8	6	5	4	} pass 3
1	2	3	4	8	6	5	1	0	} pass 4
1	2	3	4	5	6	8	1	0	} pass 5
1	2	3	4	5	6	8	1	0	} pass 6
1	2	3	4	5	6	8	1	0	} pass 7

At pass 0, the list is unordered. Following that is pass 1, in which the minimum element 1 is selected and swapped with the element 8, at the lowest index 0. In pass 2, however, only the sublist is considered, excluding the element 1. So element 2, is swapped with element 6, in the 2nd lowest index position. This process continues till the sub list is narrowed down to just one element at the highest index (which is its right position).

7.4 INSERTION SORT

The Insertion Sort algorithm is a commonly used algorithm. Even if you haven't been a programmer or a student of computer science, you may have used this algorithm. If we understand how you sort a deck of cards. You start from the beginning, go through the cards and as you find cards misplaced by precedence you remove them and insert them back into the right position. Eventually what you have is a sorted deck of cards. The same idea is applied in the Insertion Sort algorithm. The following is an implementation of the insertion sort in C.

```

void insertionSort(int a[], int array_size)
{
    int i, j, index;
    for (i = 1; i < array_size; ++i)
    {
        index = a[i];
        for (j = i; j > 0 && a[j-1] > index; j--)
            a[j] = a[j-1];
        a[j] = index;
    }
}

```

```

    }
}

```

Examine the following table. (Note that each pass represents the status of the array after the completion of the inner for loop, except for pass 0, which represents the array as it was passed to the function for sorting)

8	6	10	3	1	2	5	4	}	pass	0
6	8	10	3	1	2	5	4	}	pass	1
6	8	10	3	1	2	5	4	}	pass	2
3	6	8	10	1	2	5	4	}	pass	3
1	3	6	8	10	2	5	4	}	pass	4
1	2	3	6	8	10	5	4	}	pass	5
1 2 3	5 6 8	10				4			}pass6	
1 2 3	4 5 6 8 10								} pass 7	

The pass 0 is only to show the state of the unsorted array before it is given to the loop for sorting. Now try out the deck-of-cards-sorting algorithm with this list and see if it matches with the tabulated data. For example, you start from 8 and the next card you see is 6. Hence you remove 6 from its current position and "insert" it back to the top. That constituted pass 1. Repeat the same process and you'll do the same thing for 3 which is inserted at the top. Observe in pass 5 that 2 is moved from position 5 to position 1 since its $< (6,8,10)$ but > 1 . As you carry on till you reach the end of the list you'll find that the list has been sorted.

7.5 HEAP SORT

Note : This section needs the prior knowledge of trees and nodes and should be taken up after the pre-requisites are learnt.

Heap sort algorithm, as the name suggests, is based on the concept of heaps. It begins by constructing a special type of binary tree, called heap, out of the set of data which is to be sorted. Note:

A Heap by definition is a special type of binary tree in which each node is greater than any of its descendants. It is a complete binary tree.

A semi-heap is a binary tree in which all the nodes except the root possess the heap property.

If N be the number of a node, then its left child is $2*N$ and the right child $2*N+1$.

The root node of a Heap, by definition, is the maximum of all the elements in the set of data, constituting the binary tree. Hence the sorting process basically consists of extracting the root node and reheapifying the remaining

set of elements to obtain the next largest element till there are no more elements left to heap. Elementary implementations usually employ two arrays, one for the heap and the other to store the sorted data. But it is possible to use the same array to heap the unordered list and compile the sorted list. This is usually done by swapping the root of the heap with the end of the array and then excluding that element from any subsequent reheapings.

Significance of a semi-heap - A Semi-Heap as mentioned above is a Heap except that the root does not possess the property of a heap node. This type of a heap is significant in the discussion of Heap Sorting, since after each "Heaping" of the set of data, the root is extracted and replaced by an element from the list. This leaves us with a Semi-Heap. Reheaping a Semi-Heap is particularly easy since all other nodes have already been heaped and only the root node has to be shifted downwards to its right position. The following C function takes care of reheaping a set of data or a part of it.

```
void downHeap(int a[], int root, int bottom)
void downHeap(int a[], int root, int bottom)
{
    int maxchild, temp, child;
    while (root*2 < bottom)
    {
        child = root * 2 + 1;
        if (child == bottom)
        {
            maxchild = child;
        }
        else
        {
            if (a[child] > a[child + 1])
                maxchild = child;
            else
                maxchild = child + 1;
        }

        if (a[root] < a[maxchild])
        {
            temp = a[root]; a[root]
            = a[maxchild];
            a[maxchild] = temp;
        }
        else return;

        root = maxchild;
    }
}
```

In the above function, both root and bottom are indices into the array. Note that, theoretically speaking, we generally express the indices of the nodes starting from 1 through size of the array. But in C, we know that array indexing begins at 0; and so the left child is

```
child = root * 2 + 1
/* so, for eg., if root = 0, child = 1 (not 0) */
```

In the function, what basically happens is that, starting from root each loop performs a check for the heap property of root and does whatever necessary to make it conform to it. If it does already conform to it, the loop breaks and the function returns to caller. Note that the function assumes that the tree constituted by the root and all its descendants is a Semi-Heap.

Now that we have a downheaper, what we need is the actual sorting routine.

```
void heapsort(int a[], int array_size)
{
    int i;
    for (i = (array_size/2 - 1); i >= 0; --i)
    {
        downHeap(a, i, array_size-1);
    }

    for (i = array_size-1; i >= 0; --i)
    {
        int temp;
        temp = a[i];
        a[i] = a[0];
        a[0] = temp;
        downHeap(a, 0, i-1);
    }
}
```

Note that, before the actual sorting of data takes place, the list is heaped in the for loop starting from the mid element (which is the parent of the right most leaf of the tree) of the list.

```
    for (i = (array_size/2 - 1); i >= 0; --i)
    {
        downHeap(a, i, array_size-1);
    }
```

Following this is the loop which actually performs the extraction of the root and creating the sorted list. Notice the swapping of the i th element with the root followed by a reheaping of the list.

```

        for (i = array_size-1; i >= 0; --i)
    {
        int temp;
        temp = a[i];
        a[i] = a[0];
        a[0] = temp;
        downHeap(a, 0, i-1);
    }

```

The following are some snapshots of the array during the sorting process.
The unordered list –

8 6 10 3 1 2 5 4

After the initial heaping done by the first for loop.

10 6 8 4 1 2 5 3

Second loop which extracts root and reheaps.

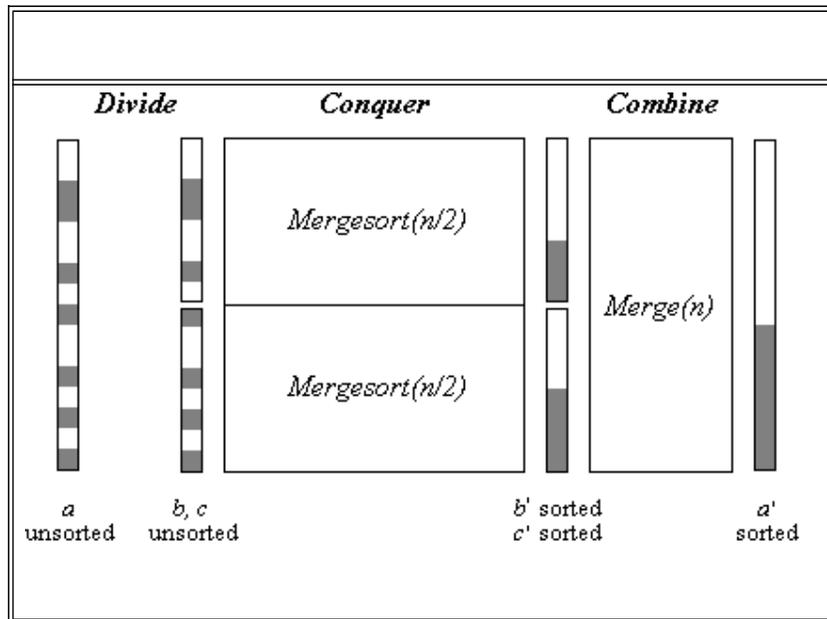
8	6	5	4	1	2	3	10	}	pass	1
6	4	5	3	1	2	8	10	}	pass	2
5	4	2	3	1	6	8	10	}	pass	3
4	3	2	1	5	6	8	10	}	pass	4
3	1	2	4	5	6	8	10	}	pass	5
2	1	3	4	5	6	8	10	}	pass	6
1	2	3	4	5	6	8	10	}	pass	7
1 2 3 4 5 6 8 10 } pass 8										

Heap sort is one of the preferred sorting algorithms when the number of data items is large. Its efficiency in general is considered to be poorer than quick sort and merge sort.

7.6 MERGE SORT

The sorting algorithm Mergesort produces a sorted sequence by sorting its two halves and merging them. With a time complexity of $O(n \log(n))$ Mergesort is optimal.

The Mergesort algorithm is based on a divide and conquer strategy. First, the sequence to be sorted is decomposed into two halves (*Divide*). Each half is sorted independently (*Conquer*). Then the two sorted halves are merged to a sorted sequence (*Combine*) as shown in the figure below.



The following procedure *mergesort* sorts a sequence a from index lo to index hi .

```
void mergesort(int lo, int hi)
{
    if (lo < hi)
    {
        int m = (lo + hi) / 2;
        mergesort(lo, m);
        mergesort(m + 1, hi);
        merge(lo, m, hi);
    }
}
```

First, index m in the middle between lo and hi is determined. Then the first part of the sequence (from lo to m) and the second part (from $m+1$ to hi) are sorted by recursive calls of *mergesort*. Then the two sorted halves are merged by procedure *merge*. Recursion ends when $lo = hi$, i.e. when a subsequence consists of only one element.

The main work of the Merge sort algorithm is performed by function *merge*. There are different possibilities to implement this function. Here is the complete C language implementation of the Merge Sort algorithm.

```
mergesort(int a[], int low, int high)
{
    int mid;
    if (low < high)
    {
        mid = (low + high) / 2;
        mergesort(a, low, mid);
```

```
mergesort(a,mid+1,high);
merge(a,low,high,mid);
}
→ return 0;

}
```

```
merge(int a[], int low, int high, int mid)
{
int i, j, k, c[50];
i=low;
j=mid+1;
k=low;
while((i<=mid)&&(j<=high))
{
    if(a[i]<a[j])
    {
        c[k]=a[i];
        k++;
        i++;
    }
    else
    {
        c[k]=a[j];
        k++;
        j++;
    }
}
while(i<=mid)
{
    c[k]=a[i];
    k++;
    i++;
}
while(j<=high)
{
    c[k]=a[j];
    k++;
    j++;
}
for(i=low;i<k;i++)
{
    a[i]=c[i];
}
```

}

7.7 ALGORITHMIC EFFICIENCY

In computer science, often the question is not how to solve a problem, but how to solve a problem well. For instance, take the problem of sorting. Many sorting algorithms which we have discussed above are well-known; the problem is of all the different ways to sort how to decide which is appropriate and how to decide which algorithm will be efficient in a given situation. In this section we understand how to compare the relative efficiency of algorithms and why it's important to do so.

If it is possible to solve a problem by using a brute force technique, such as trying out all possible combinations of solutions (for instance, sorting a group of words by trying all possible orderings until you find one that is in order), then why is it necessary to find a better approach? The simplest answer is, if you had a fast enough computer, maybe it wouldn't be. But as it stands, we do not have access to computers fast enough. For instance, if you were to try out all possible orderings of 100 words, that would require $100!$ (100 factorial) orders of words. (Explanation) That's a number with a 158 digits; were you to compute 1,000,000,000 possibilities per second, you would still be left with the need for over 1×10^{149} seconds, which is longer than the expected life of the universe. Clearly, having a more efficient algorithm to sort words would be inevitable.

Before going further, it's important to understand some of the terminology used for measuring algorithmic efficiency. Usually, the efficiency of an algorithm is expressed as how long it runs in relation to its input. For instance, in the above example, we showed how long it would take our naive sorting algorithm to sort a certain number of words. Usually we refer to the length of input as n ; so, for the above example, the efficiency is roughly $n!$. You might have noticed that it's possible to come up with the correct order early on in the attempt -- for instance, if the words are already partially ordered, it's unlikely that the algorithm would have to try all $n!$ combinations. Often we refer to the average efficiency, which would be in this case $n!/2$. But because the division by two is nearly insignificant as n grows larger (half of 2 billion is, for instance, still a very large number), we usually ignore constant terms (unless the constant term is zero).

Now that we can describe any algorithm's efficiency in terms of its input length (assuming we know how to compute the efficiency), we can compare algorithms based on their "order". Here, "order" refers to the mathematical method used to compare the efficiency -- for instance, n^2 is "order of n squared," and $n!$ is "order of n factorial." It should be obvious that an order of n^2 algorithm is much less efficient than an algorithm of

order n . But not all orders are polynomial -- we've already seen $n!$, and some are order of $\log n$, or order 2^n .

Order is abbreviated with a capital O : for instance, $O(n^2)$. This notation, known as big- O notation, is a typical way of describing algorithmic efficiency; note that big- O notation typically does not call for inclusion of constants. Also, if you are determining the order of an algorithm and the order turns out to be the sum of several terms, you will typically express the efficiency as only the term with the highest order. For instance, if you have an algorithm with efficiency $n^2 + n$, then it is an algorithm of order $O(n^2)$.

7.8 ANALYZING ALGORITHMS

Analysis of algorithms is the theoretical study of computer program performance and resource usage.

The ability to analyze a piece of code or an algorithm and understand its efficiency is vital for many applications as in recent times due to rapid changes in hardware it is now possible to capture and store large amounts of data for processing. It becomes inevitable that we understand how efficient the algorithms are to improve speed of response. A good understanding of the efficiency also helps the programmer in optimizing the code for better performance.

One approach to determining an algorithm's order is to start out assuming an order of $O(1)$, an algorithm that doesn't do anything and immediately terminates no matter what the input. Then, find the section of code that you expect to have the highest order. From there, work out the algorithmic efficiency from the outside in -- figure out the efficiency of the outer loop or recursive portion of the code, then find the efficiency of the inner code; the total efficiency is the efficiency of each layer of code multiplied together.

For instance, to compute the efficiency of a simple selection sort

```
for(int x=0; x<n; x++)  
  
    {  
  
        int min = x;  
  
        for(int y=x; y<n; y++)  
  
            {  
  
                if(array[y]<array[min])  
  
                    min=y;  
  
            }  
  
        int temp = array[x];  
  
        array[x] = array[min];  
  
        array[min] = temp;  
  
    }
```

We compute that the order of the outer loop (for(int x = 0; ..)) is $O(n)$; then, we compute that the order of the inner loop is roughly $O(n)$. Note that even though its efficiency varies based on the value of x , the average efficiency is $n/2$, and we ignore the constant, so it's $O(n)$. After multiplying together the order of the outer and the inner loop, we have $O(n^2)$.

In order to use this approach effectively, you have to be able to deduce the order of the various steps of the algorithm. And you won't always have a piece of code to look at; sometimes you may want to just discuss a concept and determine its order. Some efficiencies are more important than others in computer science, and on the next section, you'll see a list of the most important and useful orders of efficiency, along with examples of algorithms having that efficiency.

7.9 ALGORITHMIC EFFICIENCY -- VARIOUS ORDERS AND EXAMPLES

Below, we will list some of the common orders and give example algorithms.

O(1) An algorithm that runs the same no matter what the input. For instance, an algorithm that always returns the same value regardless of input could be considered an algorithm of efficiency O(1).

O(log n)

Algorithms based on binary trees are often O(log n). This is because a perfectly balanced binary search tree has log n layers, and to search for any element in a binary search tree requires traversing a single node on each layer.

The binary search algorithm is another example of a O(log n) algorithm. In a binary search, one is searching through an ordered array and, beginning in the middle of the remaining space to be searched, whether to search the top or the bottom half. You can divide the array in half only log n times before you reach a single element, which is the element being searched for, assuming it is in the array.

O(n)

Algorithms of efficiency **n** require only one pass over an entire input. For instance, a linear search algorithm, which searches an array by checking each element in turn, is O(n). Often, accessing an element in a linked list is O(n) because linked lists do not support random access.

O(n log n)

Often, good sorting algorithms are roughly O(n log n). An example of an algorithm with this efficiency is merge sort, which breaks up an array into two halves, sorts those two halves by recursively calling itself on them, and then merging the result back into a single array. Because it splits the array in half each time, the outer loop has an efficiency of log n, and for each "level" of the array that has been split up (when the array is in two halves, then in quarters, and so forth), it will have to merge together all of the elements, an operations that has order of n.

$O(n^2)$

A fairly reasonable efficiency, still in the polynomial time range, the typical examples for this order come from sorting algorithms, such as the selection sort example on the previous page.

 $O(2^n)$

The most important non-polynomial efficiency is this exponential time increase. Many important problems can only be solved by algorithms with this (or worse) efficiency. One example is factoring large numbers expressed in binary; the only known way is by trial and error, and a naive approach would involve dividing every number less than the number being factored into that number until one divided in evenly. For every increase of a single digit, it would require twice as many tests.

There are a number of other issues of efficiency of algorithm which are not directly related to performance but are nonetheless very important for designing good software.

Designing great software is not just about performance. Some of this issues are perhaps more important than performance.

Here is the list of things more important than performance.

- modularity,
- correctness,
- maintainability,
- security,
- functionality,
- robustness,
- user-friendliness,
- programmer's time,
- simplicity,
- extensibility,
- reliability, and
- scalability.

7. UNIT END EXERCISES

1. Given a list of ten numbers Write a C program to sort them in ascending order using the bubble sort algorithm. Print the original list and the sorted list.
2. Given the names of ten students write a C program to sort them in descending order using insertion sort. Print the original list and the sorted list.
3. Given the roll numbers and names of students in two parallel arrays sort them in ascending order of roll numbers using any of the algorithms.
4. Given 30 numbers sort them using Mergesort by partitioning the array.
5. Write a program to input a number n and find its factorial. In terms of efficiency comment on the efficiency order of the algorithm.
6. Given two integers x and n write a C program to find the value of x to the power of n. Can you think of solving it in any alternative ways so that the efficiency of the algorithm is improved.
7. Write a program to store the roll numbers and names of 20 students in two different arrays. Allow the user to enter a roll number and search for the name and display it. Review the logic of your program for efficiency.



FUNCTION AN MACROS

Unit Structure

1. Objectives
2. Introduction
3. Function Characteristics
4. *Function Basics*
5. Function Prototypes
6. Function Philosophy
7. Macro Definition And Substitution
8. Scope Of Function Variables
9. Recursive Functions
- 8.9 Unit End Exercises

1. OBJECTIVES

After going through this chapter you will be able to

1. Understand what Functions are and why are they needed.
2. Be able to define a Function in terms of its arguments and return values
3. Understand when and how to use Functions
4. Understand what are Macros and why they are needed
5. Explain how Macros are different from functions?
6. Understand what is Recursion?
7. Explain the Advantages of Recursion
8. Write programs for some standard situations for recursive functions such as Fibonacci Sequence and Towers of Hanoi
9. Be able to understand situations where recursion is needed

2. INTRODUCTION

By this Chapter it must have been obvious to the learner that every statement in C language is in reality a function. You must have also learnt that every program must have a special function called main. While it is possible to write the entire program with just one function called main one realizes that with tasks which are complex the program written with just one main function will start becoming larger and larger due to repetition of code and debugging and maintaining the code will become a tedious task. For this reason it becomes inevitable to break the program into independent functional parts which can then be coded independently and then combined into one single unit. This also allows teams of programmers to function independently and then consolidate the program. Such independently coded programs are called subroutines or subprograms and every computer language has a way of incorporating them to solve complex programming tasks. In C such subprograms are called **Functions**.

In C Language Functions are classified into two categories. **Standard** or **Library** functions such as **printf()**, **scanf()** etc which are available to the programmer as part of the standard package given by the company which supplies the compiler or interpreter. The definition of these functions is stored in the corresponding header files such as `stdio.h` and `conio.h`. The other category which we shall discuss in detail in this chapter is the user defined functions which are created by the programmer for some specific purpose and for which a standard function does not exist. These functions can also become part of the library functions and a programmer can create his/her own header files in which all functions of a certain type can be stored for later use. We will first look at the common characteristics of all functions and then learn how to create user defined functions.

8.2 FUNCTION CHARACTERISTICS

1. A function is a **black box** that we've locked part of our program into. The idea behind a function is that it *compartmentalizes* or modularizes part of the program, and in particular, that the code within the function has some useful properties:
2. It performs some **well-defined task**, which will be useful to other parts of the program.
3. It might be useful to other programs as well; that is, we might be able to **reuse it** (and without having to rewrite it).
4. The rest of the program doesn't have to know the details of how the function is implemented. This can make the rest of the program easier to think about.
5. The function performs its task *well*. It may be written to do a little more than is required by the first program that calls it, with the anticipation that the calling program (or some other program) may later need the extra functionality or improved performance. (It's important that a finished function do its job well, otherwise there might be a reluctance to call it, and it therefore might not achieve the goal of reusability.)
6. By placing the code to perform the useful task into a function, and simply calling the function in the other parts of the program where the task must be performed, the rest of the program becomes clearer: rather than having some large, complicated, difficult-to-understand piece of code repeated wherever the task is being performed, we have a single simple function call, and the name of the function reminds us which task is being performed.

Since the rest of the program doesn't have to know the details of how the function is implemented, the rest of the program doesn't care if the function is reimplemented later, in some different way (as long as it continues to perform its same task, of course!). This means that one part of the program can be rewritten, to improve performance or add a new feature (or simply to fix a bug), without having to rewrite the rest of the program.

Functions are the most important way to deal with the issue of software complexity. We will learn when it's appropriate to break the main program into functions, and *how* to set up function interfaces to best achieve the qualities mentioned above: reusability, information hiding, clarity, and maintainability.

8.3 FUNCTION BASICS

In this section we will learn how to define a function. It has a *name* that you call it by, and a list of zero or more *arguments* or *parameters* that you forms the input to it for it to act on; it has a *body* containing the actual instructions (statements) for carrying out the task the function is supposed to perform; and it may give you back a *return value*, of a particular type. It is not necessary that a function must return a value and it is also possible that a function may have more than one return statements.

Here is a very simple function, which accepts one argument, multiplies it by 2, and hands that value back:

```
int multbytwo(int x)
{
    int retval;
    retval = x * 2;
    return retval;
}
```

On the first line we see the return type of the function (`int`), the name of the function (`multbytwo`), and a list of the function's arguments, enclosed in parentheses. Each argument has both a name and a type; `multbytwo` accepts one argument, of type `int`, named `x`. The name `x` is arbitrary, and is used only within the definition of `multbytwo`. The caller of this function only needs to know that a single argument of type `int` is expected; the caller does not need to know what name the function will use internally to refer to that argument. (In particular, the caller does not have to pass the value of a variable named `x`.)

Next we see, surrounded by the familiar braces, the body of the function itself. This function consists of one declaration (of a local variable `retval`) and two statements. The first statement is a conventional expression statement, which computes and assigns a value to `retval`, and the second statement is a `return` statement, which causes the function to return to its caller, and also specifies the value which the function returns to its caller.

The `return` statement can return the value of any expression, so we don't really need the local `retval` variable; the function could be condensed to

```
int multbytwo(int x)
{
    return x * 2;
}
```

How do we call a function? We've been doing so informally since day one, but now we have a chance to call one that we've written, in full detail. Here is a tiny skeletal program to call `multby2`:

```
#include <stdio.h>

extern int multbytwo(int);
int main()
{
    int i, j;
    i = 3;
    j = multbytwo(i);
    printf("%d\n", j);
    → return 0;
}
```

This looks much like our other test programs, with the exception of the new line

```
extern int multbytwo(int);
```

This is an *external function prototype declaration*. It is an external declaration, in that it declares something which is defined somewhere else. (We've already seen the defining instance of the function `multbytwo`, but maybe the compiler hasn't seen it yet.) The function prototype declaration contains the three pieces of information about the function that a caller needs to know: the function's name, return type, and argument type(s). Since we don't care what name the `multbytwo` function will use to refer to its first argument, we don't need to mention it. (On the other hand, if a function takes several arguments, giving them names in the prototype may make it easier to remember which is which, so names may optionally be used in function prototype declarations.) Finally, to remind us that this is an external declaration and not a defining instance, the prototype is preceded by the keyword `extern`.

The presence of the function prototype declaration lets the compiler know that we intend to call this function, `multbytwo`. The information in the prototype lets the compiler generate the correct code for calling the function, and also enables the compiler to check up on our code (by making sure, for example, that we pass the correct number of arguments to each function we call).

Down in the body of `main`, the action of the function call should be obvious: the line

```
j = multbytwo(i);
```

calls `multbytwo`, passing it the value of `i` as its argument. When `multbytwo` returns, the return value is assigned to the variable `j`. (Notice that the value of `main`'s local variable `i` will become the value of `multbytwo`'s parameter `x`)

This example is written out in "longhand," to make each step equivalent. The variable `i` isn't really needed, since we could just as well call `j`

```
= multbytwo(3);
```

And the variable `j` isn't really needed, either, since we could just as well call

```
printf("%d\n", multbytwo(3));
```

Here, the call to `multbytwo` is a sub-expression which serves as the second argument to `printf`. The value returned by `multbytwo` is passed immediately to `printf`. (It is interesting to see the flexibility and generality of expressions in C. An argument passed to a function may be an arbitrarily complex sub-expression, and a function call is itself an expression which may be embedded as a sub-expression within arbitrarily complicated surrounding expressions.)

We should say a little more about the mechanism by which an argument is passed down from a caller into a function. Formally, C is *call by value*, which means that a function receives *copies* of the values of its arguments. We can illustrate this with an example. Suppose, in our implementation of `multbytwo`, we had gotten rid of the unnecessary `retval` variable like this:

```
int multbytwo(int x)
{
    x = x * 2;
    return x;
}
```

We might wonder, if we wrote it this way, what would happen to the value of the variable `i` when we called

```
j = multbytwo(i);
```

When our implementation of `multbytwo` changes the value of `x`, does that change the value of `i` up in the caller? The answer is no. `x` receives a copy of `i`'s value, **so when we change `x` we don't change `i`**.

However, there is an exception to this rule. When the argument you pass to a function is not a single variable, but is rather an array, the function does *not* receive a copy of the array, and it therefore *can* modify the array in the caller. The reason is that it might be too expensive to copy the entire array, and furthermore, it can be useful for the function to write into the caller's array, as a way of handing back more data than would fit in the function's single return value. We'll see an example of an array argument (which the function deliberately writes into) in the next section.

Passing arrays to a function

Here we will discuss with the help of an example how to pass an array to a function. To pass a one-dimensional array to a function it is necessary to give the name of the array without the subscripts and the size of the array. For e.g the call `highest(x,n)` will pass the entire array `x` of size `n` to the called function. And the function `highest` can be defined as `int highest(int x[], int size)` if the array comprises of only integers.

Here is an illustrative example of a function `highest` which finds the highest of 5 numbers stored in an array.

```
main()
{
    float highest(float x[], int n); float
    a[5] = {3.6,7.8,2.3,1.2,5.6};
    printf(“%f\n”, highest(a,5));
}

float highest(x[], int n)
{
    int i; float
    high; high =
    x[0];
    for (i = 1; i<n,i++)
        if (high < x[i]
            high = x[i];
    return (high);
}
```

Note that when the array `a` is passed to the function `highest` the values of all the elements of the array `a` are passed to the corresponding elements of array `x` in the called function. It must also be noted that in C passing the name of the array actually passes the address of the first element of the array. So effectively the called function is referring to the same memory locations as the calling function. This also means that any changes that the called function carries out on the array will also affect the array in the calling function. This is also called in C as passing by reference unlike the normal variables which are passed by values.

8.4 FUNCTION PROTOTYPES

In modern C programming, it is considered good practice to use prototype declarations for all functions that you call. As we mentioned, these prototypes help to ensure that the compiler can generate correct code for calling the functions, as well as allowing the compiler to catch certain mistakes you might make.

Strictly speaking, however, prototypes are optional. If you call a function for which the compiler has not seen a prototype, the compiler will do the best it can, assuming that you're calling the function correctly.

If prototypes are a good idea, and if we're going to get in the habit of writing function prototype declarations for functions we call that we've written (such as `multbytvo`), what happens for library functions such as `printf`? Where are their prototypes? The answer is in that boilerplate line

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

we've been including at the top of all of our programs. `stdio.h` is conceptually a file full of external declarations and other information pertaining to the "Standard I/O" library functions, including `printf`. The `#include` directive (which we'll meet formally in a later chapter) arranges that all of the declarations within `stdio.h` are considered by the compiler, rather as if we'd typed them all in ourselves. Somewhere within these declarations is an external function prototype declaration for `printf`, which satisfies the rule that there should be a prototype for each function we call. (For other standard library functions we call, there will be other "header files" to include.) Finally, one more thing about external function prototype declarations. We've said that the distinction between external declarations and defining instances of normal variables hinges on the presence or absence of the keyword `extern`. The situation is a little bit different for functions. The "defining instance" of a function is the function, including its body (that is, the brace-enclosed list of declarations and statements implementing the function). An external declaration of a function, even without the keyword `extern`, looks nothing like a function declaration. Therefore, the keyword `extern` is optional in function prototype declarations. If you wish, you can write

```
int multbyt看wo(int);
```

and this is just as good an external function prototype declaration as

```
extern int multbyt看wo(int);
```

(In the first form, without the `extern`, as soon as the compiler sees the semicolon, it knows it's not going to see a function body, so the declaration can't be a definition.) You may want to stay in the habit of using `extern` in all external declarations, including function declarations, since "extern = external declaration" is an easier rule to remember.

8.5 FUNCTION PHILOSOPHY

What makes a good function? The most important aspect of a good "building block" is that it has a single, well-defined task to perform. When you find that a program is hard to manage, it's often because it has not been designed and broken up into functions cleanly.

Two obvious reasons for moving code down into a function are because:

1. It appeared in the main program several times, such that by making it a function, it can be written just once, and the several places where it used to appear can be replaced with calls to the new function. This reduces the size of the main program and results in efficient use of memory and also speeds up the program. We have referred to this earlier as reusability.

2. The main program was getting too big, so it could be made (presumably) smaller and more manageable by lopping part of it off and making it a function.

These two reasons are important, and they represent significant benefits of well-chosen functions, but they are not sufficient to automatically identify a good function. As we've been suggesting, a good function has at least these two additional attributes:

3. It does just one well-defined task, and does it well.

4. Its interface to the rest of the program is clean and narrow.

Attribute 3 is just a restatement of two things we said above. Attribute 4 says that you shouldn't have to keep track of too many things when calling a function. If you know what a function is supposed to do, and if its task is simple and well-defined, there should be just a few pieces of information you have to give it to act upon, and one or just a few pieces of information which it returns to you when it's done. If you find yourself having to pass lots and lots of information to a function, or remember details of its internal implementation to make sure that it will work properly this time, it's often a sign that the function is not sufficiently well-defined. A poorly-defined function may be an arbitrary chunk of code that was ripped out of a main program that was getting too big, such that it essentially has to have access to all of that main function's local variables.

The whole point of breaking a program up into functions is so that you don't have to think about the entire program at once; ideally, you can think about just one function at a time. We say that a good function is a "black box," which is supposed to suggest that the "container" it's in is opaque--callers can't see inside it and the function inside can't see out. When you call a function, you only have to know **what it does, not how it does it**. When you're *writing* a function, you only have to know what it's supposed to do, and you don't have to know why or under what circumstances its caller will be calling it. When designing a function, we should perhaps think about the callers just enough to ensure that the function we're designing will be easy to call, and that we aren't accidentally setting things up so that callers will have to think about any internal details.

Some functions may be hard to write (if they have a hard job to do, or if it's hard to make them do it truly well), but that difficulty should be compartmentalized along with the function itself. Once a "hard" function is written, as a programmer you should be able to watch it do that hard work on call from the rest of your program. It should be noted how much easier the rest of the program is to write, once the hard work can be deferred to the function.

In fact, if a difficult-to-write function's interface is well-defined, you may be able to get away with writing a quick-and-dirty version of the function first, so that you can begin testing the rest of the program, and then go back later and rewrite the function to do the hard parts. As long as the

function's original interface anticipated the hard parts, you won't have to rewrite the rest of the program when you fix the function.

It is to be noted that functions are important for far more important reasons than just saving typing. Sometimes, we'll write a function which we only call once, just because breaking it out into a function makes things clearer and easier.

If you find that difficulties pervade a program, that the hard parts can't be buried inside black-box functions and then forgotten about; if you find that there are hard parts which involve complicated interactions among multiple functions, then the program probably needs **redesigning**.

For the purposes of explanation, we've been seeming to talk so far only about "main programs" and the functions they call and the rationale behind moving some piece of code down out of a "main program" into a function. But in reality, there's obviously no need to restrict ourselves to a two-tier scheme. Any function we find ourself writing will often be appropriately written in terms of sub-functions, sub-sub-functions, etc. Furthermore, the "main program," `main()`, is itself just a function.

Check Your Progress

1. Write a function which accepts two distinct numbers as arguments and returns the higher of the two.
2. Write a function that takes an integer number as argument and returns its factorial.
3. Write a function that takes accepts two integers as arguments and calculates the GCD of the two numbers.
4. Write a function that takes accepts two integers as arguments and calculates the LCM of the two numbers.
5. Write a function that accepts a lower case string as a parameter and converts it into an upper case string.
6. Write a function that accepts a string as parameter and returns the string after reversing it.
7. Write a function that accepts a string and checks whether it is alphabetic (that it contains no other characters than alphabets).
8. Write a function that accepts a number and checks whether it is prime or not.
9. Write a function that accepts two numbers as parameters and checks whether the first number is divisible by the second number and returns an appropriate message.
10. Write a function that accepts a decimal number and returns its binary equivalent.

8.7 MACRO DEFINITION AND SUBSTITUTION

A preprocessor line of the form

```
#define name text
```

defines a *macro* with the given name, having as its *value* the given replacement text. After that (for the rest of the current source file), wherever the preprocessor sees that name, it will replace it with the replacement text. The name follows the same rules as ordinary identifiers (it can contain only letters, digits, and underscores, and may not begin with a digit). Since macros behave quite differently from normal variables (or functions), it is customary to give them names which are all capital letters (or at least which begin with a capital letter). The replacement text can be absolutely anything--it's not restricted to numbers, or simple strings, or anything.

The most common use for macros is to propagate various constants around and to make them more self-documenting. We've been saying things like

```
char line[100];
```

```
...
```

```
getline(line, 100);
```

but this is neither readable nor reliable; it's not necessarily obvious what all those 100's scattered around the program are, and if we ever decide that 100 is too small for the size of the array to hold lines, we'll have to remember to change the number in two (or more) places. A much better solution is to use a macro:

```
#define MAXLINE 100
```

```
char line[MAXLINE];
```

```
...
```

```
getline(line, MAXLINE);
```

Now, if we ever want to change the size, we only have to do it in one place, and it's more obvious what the words `MAXLINE` through the program mean than the magic numbers 100 did.

Since the replacement text of a preprocessor macro can be anything, it can also be an expression, although you have to realize that, as always, the text is substituted (and perhaps evaluated) later. No evaluation is performed when the macro is defined. For example, suppose that you write something like

```
#define A 2
#define B 3
#define C A + B
```

Then, later, suppose that you write

```
int x = C * 2;
```

If A, B, and C were ordinary variables, you'd expect x to end up with the value 10. But let's see what happens.

The preprocessor always substitutes text for macros exactly as you have written it. So it first substitutes the replacement text for the macro C, resulting in

```
int x = A + B * 2;
```

Then it substitutes the macros A and B, resulting in

```
int x = 2 + 3 * 2;
```

Only when the preprocessor is done doing all this substituting does the compiler get into the act. But when it evaluates that expression (using the normal precedence of multiplication over addition), it ends up initializing x with the value 8!

To guard against this sort of problem, it is always a good idea to include explicit parentheses in the definitions of macros which contain expressions. If we were to define the macro C as

```
#define C (A + B)
```

then the declaration of x would ultimately expand to

```
int x = (2 + 3) * 2;
```

and x would be initialized to 10, as we probably expected.

Notice that there does not have to be (and in fact there usually is *not*) a semicolon at the end of a `#define` line. (This is just one of the ways that the syntax of the preprocessor is different from the rest of C.) If you accidentally type

```
#define MAXLINE 100;
    /* WRONG */
```

then when you later declare

```
char line[MAXLINE];
```

the preprocessor will expand it to

```
char line[100];          /* WRONG */
```

which is a syntax error. This is what we mean when we say that the preprocessor doesn't know anything about the syntax of C--in this last example, the *value* or replacement text for the macro MAXLINE were the 4 characters

1 0 0 ; , and that's exactly what the preprocessor substituted (even though it didn't make any sense).

Simple macros like `MAXLINE` act sort of like little variables, whose values are constant (or constant expressions). It's also possible to have macros which look like little functions (that is, you invoke them with what looks like function call syntax, and they expand to replacement text which is a function of the actual arguments they are invoked with).

8. SCOPE OF FUNCTION VARIABLES

Only a limited amount of information is available within each function. Variables declared within the calling function can't be accessed unless they are passed to the called function as arguments. The only other contact a function might have with the outside world is through global variables.

Local variables are declared within a function. They are created anew each time the function is called, and destroyed on return from the function. Values passed to the function as arguments can also be treated like local variables.

Static variables are slightly different, they don't die on return from the function. Instead their last value is retained, and it becomes available when the function is called again.

Global variables don't die on return from a function. Their value is retained, and is available to any other function which accesses them.

9. RECURSIVE FUNCTIONS

A recursive function is one which calls itself. We shall provide some examples to illustrate recursive functions.

Recursive functions are useful in evaluating certain types of mathematical function. You may also encounter certain dynamic data structures such as linked lists or binary trees. Recursion is a very useful way of creating and accessing these structures.

Here is a recursive version of the Fibonacci function. We saw a non recursive version of this earlier.

```
int fib(int num)
/* Fibonacci value of a number */
{   switch(num) {
    case 0:
        return(0);
        break;
```

```

case 1:
    return(1);
    break;
default: /* Including recursive calls */
    return(fib(num - 1) + fib(num - 2));
    break;
}
}

```

We met another function earlier called power. Here is an alternative recursive version.

```

double power(double val, unsigned pow)
{
    if(pow == 0) /* pow(x, 0) returns 1 */
        return(1.0);
    else
        return(power(val, pow - 1) * val);
}

```

Notice that each of these definitions incorporate a test. Where an input value gives a trivial result, it is returned directly, otherwise the function calls itself, passing a changed version of the input values. Care must be taken to define functions which will not call themselves indefinitely, otherwise your program will never finish.

The definition of fib is interesting, because it calls itself twice when recursion is used. Consider the effect on program performance of such a function calculating the fibonacci function of a moderate size number.

Input Value	Number of times fib is called
0	1
1	1
2	3
3	5
4	9
5	15
6	25
7	41
8	67
9	109
10	177

If such a function is to be called many times, it is likely to have an adverse effect on program performance.

Recursion might seem complex to begin with but once grasped it offers a very effective and efficient way of solving a certain set of programming problems. Recursive functions are sometimes the simplest answer to a calculation. However there is always an alternative non-recursive solution available too. This will normally involve the use of a loop, and may lack the elegance of the recursive solution.

Towers of Hanoi and its C implementation

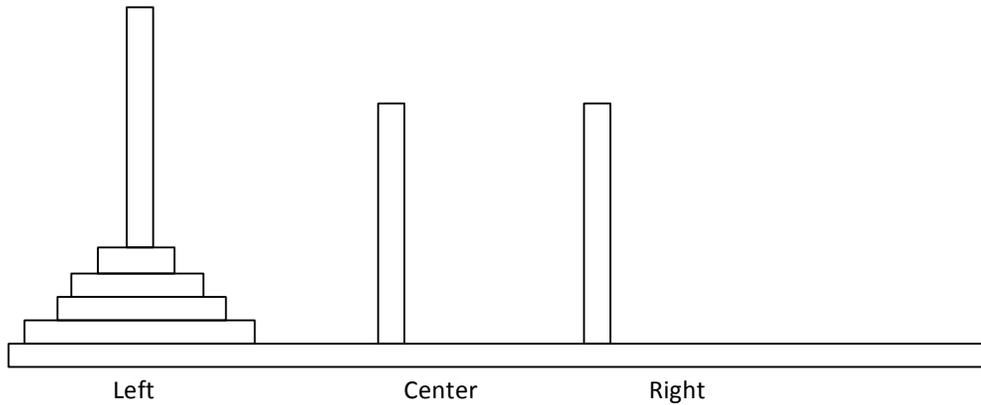
The towers of Hanoi is a well known game for children. It has three poles and a number of different sized discs. Each disc has a hole at the center allowing it to be stacked on any of the three poles. To begin with the disc are all stacked on the first or the leftmost pole in the order of decreasing size. i.e. smallest on top and the largest disc at the bottom.

The aim of the game is to transfer the discs from the leftmost pole to the rightmost pole without ever placing a larger disc on a smaller disc. Only one disc maybe moved at a time and each disc must always be placed around one of the poles. The general approach is to consider one of the poles to be the origin and another to be the destination. The third pole is used for intermediate storage purposes allowing the movement of discs in such a way as to avoid placing the larger disc on a smaller one.

Assuming there are n discs numbered from the smallest to the largest, if the discs are initially stacked on the left pole, the problem of moving n discs to the right pole can be stated recursively as follows.

1. Move the top $n-1$ discs from the left pole to the center pole.
2. Move the n th (largest) disc to the right pole.
3. Move the $n-1$ discs from the center pole to the right pole.

Refer to the diagram below



In order to program this game we first assign labels to the poles. The leftmost pole is labeled as 'L', the center or intermediary pole as 'C' and the rightmost or the destination pole as 'R'. It is now possible to construct a recursive function called **shift** that will shift n disc from one pole to the another. We refer to the individual poles with the char type variables from, to, and temp indicating origin, destination and intermediary poles. Given this information we now write the C code to implement the towers of Hanoi solution.

```
/* Solution of the TOWERS OF HANOI using recursive algorithm */
```

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

```
#include <conio.h>
```

```
void transfer(int n, char from, char to, char temp); /* Function prototype*/
```

```
main()
```

```
{
```

```
int n;
```

```
printf("Towers of Hanoi\n");
```

```
printf("Enter number of discs?");
```

```
scanf("%d",&n);
```

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

```
transfer(n,'L','R','C');
```

```

}

void transfer(int n, char from, char to, char temp)
/* Transfer n discs from one pole to another*/
{
    if (n>0)
    /* Move n-1 discs from origin to temporary */
        transfer(n-1, from, temp, to);

    /* Move nth disc from origin to destination */
        printf("Move disk %d from %c to %c \n", n , from , to);

    /* Move n-1 discs from temporary to destination */
        transfer(n-1, temp, to , from) ;
}
return ;
}

```

It is to be noted that the function transfer receives a different set of values for its arguments each time it is called. These set of values will be pushed onto the stack independently of one another and then popped from the stack at the proper time during execution. It is this ability to store and retrieve these independent sets of values that allows for recursion to work.

Here is an example of using a recursive function called reverse to reverse a number

```
#include<stdio.h>

int main(){

    int num,rev;

    printf("\nEnter a number :");

    scanf("%d",&num);

    rev=reverse(num);

    printf("\nAfter reverse the no is :%d",rev);

    → return 0;

}

int sum=0,r;

reverse(int num){

    if(num){

        r=num%10;

        sum=sum*10+r;

        reverse(num/10);

    }

else

    return sum;

}
```

8.10 UNIT END EXERCISES

Although some of the problems here can be solved without using recursive functions the learner may try to solve them using recursion.

1. Write a program to input a string and check if it is a palindrome. (A palindrome string is a string which is spelt the same forwards and backwards such as the words rotor, ere, etc
2. Write a program to find the permutations and combinations of r things selected out of n things.
3. Write a program to find the sum of all numbers up to n.
4. Write a program to sort an array of 20 numbers.
5. Write a program to find the GCD of two numbers.



Thank You



Introduction to algorithm & C Part-4



THE C STORAGE CLASSES, SCOPE AND MEMORY

Unit Structure

1. Objectives
 2. Introduction
 3. Automatic Variables
 4. External Variables
 5. Static Variables
 6. Register Variables
 7. Definition And Declaration
 8. Unit End Exercises
-

9.0 OBJECTIVES

After learning this chapter the student will be able to

1. Understand the concept of a storage class
2. Understand the different storage classes
3. Understand the concept of scope, visibility and longevity of a variable
4. Understand which storage class should be used under what circumstances
5. Learn the advantages and disadvantages of each storage class

9.1 INTRODUCTION

In the earlier introductory chapters we have already learnt the concept of a variable in C language. Unlike many other programming languages C variables behave differently. In some of the computer languages a variable retains its value throughout the program. In C language whether the variable retains its value throughout the duration of the program depends upon its storage class.

Variables in C language typically have two characteristics. One is the data type such as int, float etc which we have covered earlier and which indicate the type of data that the variable can store and the other is the **storage class** which determines the part of memory where storage is allocated for variables and functions, and how long the storage allocation continues to exist.

C has four storage classes, namely,

1. Automatic Variables
2. External Variables
3. Static Variables
4. Register Variables

For all the above storage classes we will observe three factors. The **scope**, **visibility** and **durability** or **longevity**. The **scope** of a variable refers over what part of the program the variable continues to be available to that program. **Durability** or **longevity** refers to the time period during which the variable retains the value of the variable. The **visibility** refers to the accessibility of the variable from the memory.

Variables are also categorized depending on where in the program they are declared.

Internal variables are declared within the body of a function and **external** variables are declared outside the body of the function. Depending on where they are declared has implications on their accessibility and durability. It is also important to know that depending on the storage classes the values of these variables are physically placed in different parts of the memory of a computer

system such as registers, cache, memory (Random Access Memory) and secondary storage such as magnetic and optical disk.

A beginner programmer may wonder why C allows for storage classes, however it is important to understand that storage classes determine the speed and execution of a program and more specifically programs which have multiple functions.

9.2 AUTOMATIC VARIABLES

They are declared at the start of a program's block such as in the curly braces ({ }). Memory is allocated automatically upon entry to a block and freed automatically upon exit from the block. The scope of automatic variables is local to the block in which they are declared, including any blocks nested within that block. For these reasons, they are also called local variables. No block outside the defining block may have direct access to automatic variables (by variable name) but, they may be accessed indirectly by other blocks and/or functions using pointers. **Automatic** variables may be specified upon declaration to be of storage class `auto`. However, it is not required to use the keyword `auto` because by default, storage class within a block is `auto`. **Automatic** variables declared with initializers are initialized every time the block in which they are declared is entered or accessed.

Example

```
main()
{
    int x;
    - -----;
    - -----;
}
```

In the above program the variable x will be treated as an automatic variable.

The same program could also be rewritten using the word auto.

```
main()

{
    auto int x;
    - -----;
    - -----;
}
```

It is important to note that the value of an auto variable cannot be changed inadvertently by some other function. This allows the programmer certain liberty in naming variables with same names and yet not unintentionally affecting the values of those variables in other functions.

We now show an example to illustrate how the values of auto variables remain unaffected in a program despite the same variable being used in various functions.

In the following example there is a program with two functions func1 and func2. There is a variable x which has been declared and initialized to different values in each of three functions. The main, func1 and func2. x has the value 6 in the main function , and 2 and 4 in the functions func1 and func2.

When executed the main function calls upon the function func2 which in turn calls the function func1. When the main function is active the value of x is 6 but when main calls func2 its value is temporarily put on hold and a new variable x with value 4 gets created in function func2. When func2 calls func1 its value (4) is kept on hold and a third variable also bearing the same name x is created with value 2 and becomes active. When the execution of func1 gets over the value of x (2) gets printed and func2 takes over and the value 4 gets printed and finally the control returns to main function and the value 6 gets printed.

The learner should go through the code carefully to understand how the values get created and printed.

```
void func1(void);

void func2(void);

main()
{
    int x = 6;
    func2();
    printf("%d \n",x);
}

void func1(void)

{
    int x = 2;
    printf("%d \n",x);
}

void func2(void)

{
    int x = 4;
    func1();
    printf("%d \n",x);
```

```
}
```

Output is

2

4

6

The important thing to note here is that the auto variables local to main() remain alive throughout the execution of the program but active only in main. In the subsequent functions these variables get created and destroyed locally whenever the sub program or sub function execution gets completed.

9.3 EXTERNAL VARIABLES

All variables we have seen so far have had limited scope (the block in which they are declared) and limited lifetimes (as for automatic variables). However, in some applications it may be useful to have data which is accessible from within any block and/or which remains in existence for the entire execution of the program. Such variables are called **global** variables, and the C language provides storage classes which can meet these requirements; namely, the external (**extern**) and static (**static**) classes.

Declaration for external variable is as follows:

```
extern int var;
```

External variables may be declared outside any function block in a source code file the same way any other variable is declared; by specifying its type and name (extern keyword may be omitted). Typically if declared and defined at the beginning of a source file, the extern keyword can be omitted. If the program is in several source files, and a variable is defined in let say file1.c and used in file2.c and file3.c then the extern keyword must be used in file2.c and file3.c.

But, usual practice is to collect extern declarations of variables and functions in a separate header file (.h file) then included by using #include directive. Memory for such variables is allocated when the program begins execution, and remains allocated until the program terminates. For most C implementations, every byte of memory allocated for an external variable is initialized to zero.

The scope of external variables is global, i.e. the entire source code in the file following the declarations. All functions following the declaration may access the external variable by using its name. However, if a local variable having the same name is declared within a function, references to the name will access the local variable cell.

The following program example demonstrates storage classes and scope.

```
/* storage class and scope */

#include <stdio.h>

void funct1(void);
void funct2(void);

/* external variable, scope is global to main(), funct1() and funct2(), extern
keyword is omitted here, as there is just one file */

int globvar = 10;

int main()
{
    printf("\n***storage classes and scope***\n");

    /* external variable */
```

```
globvar = 20;

printf("\nVariable globvar, in main() = %d\n", globvar);

funct1();

printf("\nVariable globvar, in main() = %d\n", globvar);

funct2();

printf("\nVariable globvar, in main() = %d\n", globvar);

→ return 0;

}

/* external variable, scope is global to funct1() and funct2() */

int globvar2 = 30;

void funct1(void)
{
/* auto variable, scope local to funct1() and funct1() cannot access the external
globvar */

char globvar;

/* local variable to funct1() */

globvar = 'A';

/* external variable */

globvar2 = 40;
```

```
printf("\nIn funct1(), globvar = %c and globvar2 = %d\n", globvar, globvar2);
}

void funct2(void)
{
/* auto variable, scope local to funct2(), and funct2() cannot access the external
globvar2 */

double globvar2;

/* external variable */

globvar = 50;

/* auto local variable to funct2() */

globvar2 = 1.234;

printf("\nIn funct2(), globvar = %d and globvar2 = %.4f\n", globvar, globvar2);
}
```

And the output will be as follows

******storage classes and scope******

Variable globvar, in main() = 20

In funct1(), globvar = A and globvar2 = 40

Variable globvar, in main() = 20

In funct2(), globvar = 50 and globvar2 = 1.2340

External variables may be initialized in declarations just as automatic variables; however, the initializers must be constant expressions. The initialization is done only once at compile time, i.e. when memory is allocated for the variables.

In general, it is a good programming practice to avoid using external variables as they destroy the concept of a function as a 'black box' or independent module. The black box concept is essential to the development of a modular program with modules. With an external variable, any function in the program can access and alter the variable, thus making debugging more difficult as well. This does not mean that external variables should never be used.

There may be occasions when the use of an external variable significantly simplifies the implementation of an algorithm. Suffice it to say that external variables should be used rarely and with caution.

9.4 STATIC VARIABLES

As we have seen, external variables have global scope across the entire program (provided extern declarations are used in files other than where the variable is defined), and have a lifetime over the entire program run.

Similarly, static storage class provides a lifetime over the entire program, however; it provides a way to limit the scope of such variables, and static storage class is declared with the keyword `static` as the class specifier when the variable is defined.

These variables are automatically initialized to zero upon memory allocation just as external variables are. Static storage class can be specified for automatic as well as external variables such as:

static extern varx;

Static automatic variables continue to exist even after the block in which they are defined terminates. Thus, the value of a static variable in a function is retained between repeated function calls to the same function.

The scope of static automatic variables is identical to that of automatic variables, i.e. it is local to the block in which it is defined; however, the storage allocated becomes permanent for the duration of the program.

Static variables may be initialized in their declarations; however, the initializers must be constant expressions, and initialization is done only once at compile time when memory is allocated for the static variable.

The following example illustrates the use of static variables.

```
/* static storage class program example */  
  
#include <stdio.h>  
  
#define MAXNUM 3  
  
void sum_up(void);  
  
int main()  
{  
    int count;  
  
    printf("\n*****static storage*****\n");
```

```
printf("Key in 3 numbers to be summed ");
for(count = 0; count < MAXNUM; count++)
    sum_up();
printf("\n*****COMPLETED*****\n");

→ return 0;
    }

void sum_up(void)
{
    /* at compile time, sum is initialized to 0 */
    static int sum = 0;

    int num;

    printf("\nEnter a number: ");
    scanf("%d", &num);

    sum += num;

    printf("\nThe current total is: %d\n", sum);
}
```

And the output will be

*******static storage*******

Key in 3 numbers to be summed

Enter a number: 10

The current total is: 10

Enter a number: 10

The current total is: 20

Enter a number: 11

The current total is: 31

The student must note that the function `sum_up()` in the above example is called three times as it appears in the loop. Every time it is called the variable `sum` is accumulating the value of the newly typed in number as it retains the value (due to it being declared as static) between the function calls.

An external static variable is declared outside of all functions and is available to all the functions in that program. The difference between a static external variable and a simple external variable is that the static external variable is available only within the file where it is defined while the simple external variable can be accessed by other files.

The static class is used sometimes to control the scope of a function. For example if we would like a particular function to be accessible only to the functions defined in a file and not to other functions in other files then this can be accomplished by defining that function with the storage class `static`.

9.5 REGISTER VARIABLES

Automatic variables are allocated storage in the main memory of the computer; however, for most computers, accessing data in memory is considerably slower than processing directly in the CPU.

Registers are memory located within the CPU itself where data can be stored and accessed quickly. Normally, the compiler determines what data is to be stored in the registers of the CPU at what times.

However, the C language provides the storage class `register` so that the programmer can suggest to the compiler that particular automatic variables

should be allocated to CPU registers, if possible and it is not an obligation for the CPU to do this.

Thus, register variables provide a certain control over efficiency of program execution.

Variables which are used repeatedly or whose access times are critical may be declared to be of storage class register.

Variables can be declared as a register as follows:

```
register int var;
```

By making the above statement we are requesting the compiler that a variable called var of type int be assigned a place in the register area of the CPU. Although most ANSI standards place no restriction on what type of variable can be assigned to the register type most compilers restrict them to be of either int or character type. It must be noted that only a few variables can be assigned to be of the register storage class due to the inherent limitations on the space of the registers. If this limit is reached register storage variables then get treated as non register variables. The following table summarizes the different storage classes and their scope, visibility and durability for ready reference of a programmer. It also allows the student to compare the different storage classes at a glance.

Storage class	Place of declaration	Visibility	Durability
None	Preceding all functions in a file	Entire file and other files where the variable is declared with the word <code>extern</code>	Entire program (Global)
<code>extern</code>	Preceding all functions in a file	Entire file and other files where the variable is declared with the word <code>extern</code>	Global
<code>static</code>	Preceding all functions in a file	Only in the file where declared	Global
None or <code>auto</code>	Inside a function or block within which declared	Only in that function or block where declaration is made	Until end of function or block
<code>register</code>	Inside a function or block within which declared	Only in that function or block where declaration is made	Until end of function or block
<code>static</code>	Inside a function	Only in that function	Global

9.6 DEFINITION AND DECLARATION

Up until now, we have been using the term *declaration* rather loosely when referring to variables. In this section, we will "tighten" the definition of this term. So far when we have "declared" a variable, we have meant that we have told the compiler *about* the variable; i.e. its type and its name, as well as allocated a memory cell for the variable (either locally or globally). This latter action of the compiler, allocation of storage, is more properly called the *definition* of the variable. The stricter definition of *declaration* is simply to describe information "about" the variable.

So far, we have used declarations to *declare* variable names and types as well as to *define* memory for them. Most of the time these two actions occur at the same time, that is, most declarations are definitions; however, this may not always be the case.

In the chapter on function we have seen the difference between *declaring* and *defining* with functions. The prototype statement for a function *declares* it, i.e. tells the compiler "about" the function - its name, return type, and number and type of its parameters. A similar statement, the function header, followed by the body of the function, *defines* the function - giving the details of the steps to perform the function operation.

For automatic and register variables, there is no difference between definition and declaration. The process of declaring an automatic or a register variable defines the variable name and allocates appropriate memory. However, for external variables, these two operations may occur independently. This is important because memory for a variable must be allocated only once, to ensure that access to the variable always refers to the same cell. Thus, all variables must be defined once and only once. If an external variable is to be used in a file other than the one in which it is *defined*, a mechanism is needed to "connect" such a use with the uniquely defined external variable cell allocated for it. This process of connecting the references of the same external variable in different files, is called *resolving the references*.

As we saw in the previous section, external variables may be defined and declared with a declaration statement outside any function, with no storage class specifier. Such a declaration allocates memory for the variable. A declaration statement may also be used to simply *declare* a variable name with the `extern` storage class specifier at the beginning of the declaration. Such a declaration specifies that the variable is *defined* elsewhere, i.e. memory for this variable is allocated in another file. Thus, access to an external variable in a file other than the one in which it is *defined* is possible if it is *declared* with the keyword `extern`; no new memory is allocated. Such a declaration tells the compiler that the variable is defined elsewhere, and the code is compiled with the external variable left unresolved. The reference to the external variable is resolved during the linking process.

Here are some examples of *declarations* of external variables that are not *definitions*:

```
extern char stack[10];  
    extern int stkptr;
```

These declarations tell the compiler that the variables `stack[]` and `stkptr` are defined elsewhere, usually in some other file. If the keyword `extern` were omitted, the variables would be considered to be new ones and memory would be allocated for them. Remember, access to the same external variable defined in another file is possible only if the keyword `extern` is used in the declaration.

9.7 UNIT END EXERCISES

1. In the program below what will be the output of the program.

```
void stat(void);

main()
{
    int i;
    for (i=1;i<=3;i++)
        stat();
}

void stat(void)
{
    static int x=0;

    x=x+1;
    printf("x= %d\n",x);
}
```

2. What would its output be if the word `static` is omitted in the function definition.
3. In the exercises given in earlier chapters what variables would you declare as `register` and why?
4. In the programs written by you without prior knowledge of storage classes in what way can the efficiency or performance of the programs be improved by your understanding of the concept of storage classes.



STRUCTURES AND UNIONS

Unit Structure

- 10.0 Objectives
- 10.1 Introduction
- 10.2 Why Use Structures Defining
- 10.3 A Structure Declaring
- 10.4 Structure Variables Accessing
- 10.5 Structure Variables Arrays Of
- 10.6 Structures Arrays Within
- 10.7 Structures Structures Within
- 10.8 Structures Structures And
- 10.9 Functions Unions
- 10.10 Unit End Exercise
- 10.11

1. OBJECTIVES

After learning this chapter the student should be able to

1. Understand what are structures and why they are needed
2. Be able to define a structure
3. Be able to read and assign values to elements in a structure
4. Be able to understand the relationship between arrays and structures
5. Be able to define structures within structures
6. Be able to understand the relationship between structures and functions
7. Be able to understand what are unions
8. Write programs involving the use of structures

10.1 INTRODUCTION

In the chapters that we have learned so far we have used different data types such as int, float and char. However when we look at real life data, it rarely comes in such atomized forms, rather we come across entities that are collections of things, each thing having its own attributes, just as the entity we call a 'bill' is a collection of things such as bill number, bill date, item no, item description, quantity purchased, amount and so on. As you can see all this data is dissimilar, for example bill number is a number, whereas item description is a string and so on. For dealing with such collections, C provides a data type called 'structure'. *A structure gathers together, different individual pieces of information that makes up a given entity.* A structure can be looked upon as a tool to handle a group of logically related data. From this point on we can look at the individual pieces of information or we can look at the whole composite entity. We

will see why it is an advantage to be able to view the data in this manner and how it simplifies coding and improves performance.

10.2 WHY USE STRUCTURES

We have seen earlier how ordinary variables can hold one piece of information and we have also learnt about arrays which can hold a number of pieces of information of the same data type. These two data types can handle a great variety of situations. However sometimes we are faced with situations where we have to deal with entities that are not a collection of one homogenous type but of dissimilar data types. For example, suppose we want to store data about a book. We might want to store its name (a string), its price (a float) and number of pages it contains (an int). If data about say 3 such books is to be stored, then we can follow two approaches:

(a) Construct individual arrays, one for storing names, another for storing prices and still another for storing number of pages.

(b) Use a structure variable.

We examine these two approaches. For the sake of programming convenience assume that the names of books would of one character.

```
main()
{
char name[3] ;
float price[3] ;
int pages[3], i ;
printf ( "\nEnter names, prices and no. of pages of 3 books\n" ) ;
for ( i = 0 ; i <= 2 ; i++ )
scanf ( "%c %f %d", &name[i], &price[i], &pages[i] );
printf ( "\nAnd this is what you entered\n" ) ;
for ( i = 0 ; i <= 2 ; i++ )
printf ( "%c %f %d\n", name[i], price[i], pages[i] );
}
```

And here is the sample run...

Enter names, prices and no. of pages of 3 books

A 100.00 354

C 256.50 682

F 233.70 512

And this is the output

A 100.000000 354

C 256.500000 682

F 233.700000 512

This approach allows you to store names, prices and number of pages. But as you must have realized, it is an unwieldy approach that obscures the fact that you are dealing with a group of characteristics related to a single entity—the book.

The program becomes more difficult to handle as the number of items relating to the book go on increasing. For example, we would be required to use a number of arrays, if we also decide to store name of the publisher, date of purchase of book, etc. To solve this problem, C provides a special data type—the structure. A structure contains a number of data types grouped together.

These data types may or may not be of the same type. The following example illustrates the use of this data type.

```
main()
{
struct book
{
char name ;
float price ;
int pages ;
} ; struct book b1, b2, b3 ;
printf ( "\nEnter names, prices & no. of pages of 3 books\n" ) ;
scanf ( "%c %f %d", &b1.name, &b1.price, &b1.pages ) ; scanf
( "%c %f %d", &b2.name, &b2.price, &b2.pages ) ; scanf (
"%c %f %d", &b3.name, &b3.price, &b3.pages ) ; printf (
"\nAnd this is what you entered" ) ;
printf ( "\n%c %f %d", b1.name, b1.price, b1.pages ) ;
printf ( "\n%c %f %d", b2.name, b2.price, b2.pages ) ;
printf ( "\n%c %f %d", b3.name, b3.price, b3.pages ) ;
}
```

And here is the output...

Enter names, prices and no. of pages of 3 books

A 100.00 354

C 256.50 682

F 233.70 512

10.3 DEFINING A STRUCTURE

Unlike arrays which can be declared and used straight away structures have to be defined first and declared later. We consider an example to illustrate how a structure can be defined and variables created.

Consider a database of books containing name of book, author, number of pages and price. We can define a structure which represents the above information as follows.

```

struct book
{
    char title[20];
    char author[10];

    float price;
}

```

The keyword **struct** declares a structure and specifies that it will hold the data fields title, author, pages and price. These fields are called structure elements or members. The name of the structure is book and is also called as the **structure tag**. Each member in the structure may be of different data type. The tag name is used later to declare variables which will have the same structure. It is important to note that the above definition doesn't create any variables and that at this stage no memory has been utilized. It can be seen more as a template or a blue print.

The general format of a structure definition is as follows

```

struct tag-name
{
    datatype element1;
    datatype element2;
    .....
};

```

The following important points must be noted

1. The definition is terminated with a semicolon.
2. Although the whole definition is actually a statement each member is declared independently for its name and type
3. The tag name is used to declare structure variables in the program

10.4 DECLARING STRUCTURE VARIABLES

After specifying the format of the structure it is possible to declare variables of that type and this is done exactly in the same manner in which we have been declaring data types of variables so far. The declaration has the following syntax

1. The keyword **struct**
2. The name of the structure tag
3. List of variables separated by comma and ending with semi-colon

For example with reference to the structure book given above we can declare variables as follows.

```
struct book, book1,book2,book3;
```

The above statement declares three variables namely book1,book2 and book3 each one of these variables will have the four members title, author,

pages and price as mentioned in the book structure at the time when it was defined. It is important for a student to understand that the members of the structure are not the variables and they do not occupy any memory space until they are associated with a particular variable. It is only when the compiler comes across the declaration statement that specific memory

allocated to the variables. It is possible to combine the definition and declaration of a structure.

For example

```

    struct book
    {
        char title[20];
        char author[10];
        int pages; float
        price;
    } book1,book2,book3;

```

Will do both the definition and declaration.

How Structure Elements are Stored

It is important for the student to understand that whatever be the elements of a structure, they are always stored in contiguous memory locations. The following program would illustrate this:

```

/* Memory map of structure elements */
main( )
{
    struct book
    {
        char name ;
        float price ;
        int pages ;
    } ;

    struct book b1 = { 'B', 130.00, 550 } ;
    printf ( "\nAddress of name = %u", &b1.name ) ;
    printf ( "\nAddress of price = %u", &b1.price ) ;
    printf ( "\nAddress of pages = %u", &b1.pages ) ;
}

```

The output of the program would be as follows

```

Address of name = 65518
Address of price = 65519
Address of pages = 65523

```

Actually the structure elements are stored in memory as shown in the figure below

Variable	b1.name	B1.price	b1.pages
Value	'B'	130.00	550
Memory Location	65518	65519	65523

10.5 ACCESSING STRUCTURE VARIABLES

Once a structure is defined and its variables declared they can be accessed in a number of ways. It is to be noted that once again that the member itself such as author is not a variable however author of book1 will mean the variable author belonging to book1. The link between the member and the variable is established by the member operator “.”, which is also known as the ‘dot operator’ or ‘period operator’.

For example book1.pages will refer to the variable representing the number of pages of book1. Variables can be given values in exactly the same way that we have been doing before with the help of an assignment statement.

For e.g. book1.pages = 225;
 book2.price = 200.00

Values can also be read into these variables with the help of the scanf statement

```
scanf(“%f”,&book1.price)
```

Example illustrating the defining, declaring and accessing the variables.

Define a structure type called student that will contain the roll number, name of the student and marks in three subjects. Using this structure write a program that will input the details of the student and print it out on the screen.

The C language program will look as follows.

```
struct student
{
    int rollno;
    char name[15];
    int m1,m2,m3;
};

main()
{
```

```

struct student student1;
printf("Input Values");
scanf("%d %s %d %d %d",
      student.rollno,
      student.name,
      student.m1,
      student.m2,
      student.m3);

printf("%d %s %d %d %d",
      student.rollno,
      student.name,
      student.m1,
      student.m2,
      student.m3);
}

```

The above program accepts the values entered into the structure student through an input statement and prints it out by accessing the structure variables.

It is also possible to **initialize** the structure variables at the time of compilation by giving the variables values immediately upon declaration.

For example it is perfectly valid to do as follows

```

struct student
{
    int rollno;
    char name[15];
    int m1,m2,m3;
} student1 = { 1,"AJAY",67,78,89};

```

Or separately, while declaring the student1 variable as follows.

```

struct student student1 = {1,"AJAY", 67,78,89};

```

However, it must be noted that C language doesn't permit initialization of any variables within the template in which it is defined. The initialization can be done when variables are being declared and not when the members of a structure are being defined.

It is possible to compare and copy the structure variables in exactly the same manner that other C language variables are treated. For e.g. if student1 and student2 are variables with the same structure type then it would be valid to assign the value of one structure variable to another

For e.g. `student1 = student2` will assign all the values of `student2` to `student1`.

However C language doesn't permit any logical operations directly on the structure variables so it would not be valid to compare `student1` with `student2` so statement like `if (student1 == student2)` is not valid however individual members of a structure can be compared for values.

So it would be valid to make a statement `if (student1.m1 == student2.m1)`

The following program illustrates how individual values of structure variables can be compared.

```

struct student
    {
        int rollno;
        char name[15];
        int m1,m2,m3;
    };

main()
{
    struct student student1 = {1,"AJAY", 67,78,89};
    struct student student2 = {2,"GEETA", 57,88,90};
    struct student student3;

    student3 = student2;
    if ( student3.rollno == student2.rollno)
        printf("The students roll numbers are the same");
    else
        printf("The students roll numbers are not the
same");
}

```

An individual member gets treated exactly the same way as other variables in C language once the dot operator specifies the parent structure variable to which the member belongs.

All the operators and expressions which are valid for other variables in C language also remain valid for structure variable members once the dot or member operator is used.

For e.g. it would be valid to make the following statements

```

tot = student1.m1 + student1.m2 + student1.m3;
student1.m1++;
student1.m1 += 10;

```

10.6 ARRAYS OF STRUCTURES

We have seen how structures help us to group variables of different type which are logically related to one another because they comprise parts of a composite entity. In order to efficiently process many members who belong to the same structure we can use arrays which allow us to access each element which belongs to the same structure. The processing of arrays is to be done in the same manner as regular arrays which we have covered in our previous chapters. There is no change in the syntax except that while declaring the type of the array we refer to the structure to which individual array elements will belong.

For example `struct college student[100];`

Will define an array called `student`, each element of which belongs to a structure called `college`.

Another example would be

```
struct result
{
    int m1;
    int m2;
    int m3;
};
```

Having defined the `result` structure, we can give values using an array as follows

```
struct result student[3] = { {
35,55,75},{45,65,76},{76,43,65}};
```

The above statement declares the `student` as an array of three elements, namely `student[0]`, `student[1]`, `student[2]`; and the values of individual variables will be assigned

```
as
student[0].m1 = 35
student[0].m2 = 55
student[0].m3 = 75
student[1].m1 = 45 and so on .
```

We now present an illustration of how arrays can be used with structures.

Example Consider a structure which stores the marks of students in three subjects. It is required to write a program which finds subject wise totals of each of the three subjects. For simplicity sake we will take the data of three students.

```

struct marks
{
    int m1;
    int m2;
    int m3;
};

main()
{
    int i, tm1,tm2,tm3;
    struct marks student[3] = { { 35,55,75 },{45,65,76},{76,43,65} };

    for (i=0,i<=2;i++)
    {
        tm1 = tm1 +student[i].m1

        tm2 = tm2 +student[i].m2;
        tm3 = tm3 +student[i].m3;

    }

    printf("Total of subject %d is equal to %d\n",1,tm1);
    printf("Total of subject %d is equal to %d\n",2,tm2);
    printf("Total of subject %d is equal to %d\n",3,tm3);

}

```

10.7 ARRAYS WITHIN STRUCTURES

C language permits the use of arrays within structures. We have in previous examples used names of students or people which are actually arrays within a structure. However it is possible to use single or multi-dimensional arrays within a structures.

For e.g. the following definition defines an array within a structure.

```

struct marks
{
    int rollno;
    int m[3];
} student[2];

```

In the above example m[3] is an array comprising of elements m[0],m[1], and m[2] which are marks scored by the students in the three subjects. To access the marks scored by the first student in the first subject we will have to refer to the member by specifying student[0].m[0] , and the marks in second subject by student[0].m[1] and so on.

We can rewrite the above program by using an array to represent the marks scored by the student in the three subjects as follows

```

struct marks
{
    int m[3];
};

main()
{
    int i, tm[3]={0,0,0};
    struct marks student[3] = { { 35,55,75},{45,65,76},{76,43,65} };

        for (i=0,i<=2;i++)
        {
            for (j=0;j<=2;j++)
            {
                tm[i] =tm[i] + student[j].m[i];

            }

        }

        for (i=0,i<=2;i++)
        printf("Total of subject %d is equal to %d\n",i,tm[i];

    }

```

10.8 STRUCTURES WITHIN STRUCTURES

Structures can be nested within other structures. We consider the following structure which has all the details of a student joining an institute. We create a structure for all the details of the student and the marks of the student in three subjects constitutes a separate structure within the main structure.

```

struct student
{
    int rollno;
    char name[10];
    char addr1[10];
    char addr2[10];
    struct
    {
        int m1;
        int m2;
        int m3;
    } mark;
} student;

```

Here the main structure gives the details of the student other than the marks in three subjects which have been separately grouped as a logical entity.

The student structure contains the member called mark which itself is a structure with members m1,m2 and m3.

The members contained in the inner structure namely mark can be referred to by referring to the main structure as follows

```
student.mark.m1
```

The innermost member within a structure can be referred to by connecting all the structure variables from the outermost to the innermost. The actual connecting is done by using the dot or the member operator.

However merely mentioning the names of the structure variables such as student.mark has no meaning as it is not referring to any specific member. It is also not valid if one writes student.m1 as m1 belongs to the structure

mark and not the structure student, and the immediate structure mark cannot be omitted.

10.9 STRUCTURES AND FUNCTIONS

In the same way as an ordinary variable, a structure variable can also be passed to a function. We may either pass individual structure elements or the entire structure variable at one go. We examine both the approaches by using suitable programs.

```
/* Passing individual structure elements */
main()
{
    struct student
    {
        int rollno ;
        char name[25] ;
        int m1;
    };
    struct student st = { 111, "AJAY", 90 } ;
    display ( st.rollno, st.name, st.m1 ) ;
}
display ( int x, char *t, int n )
{
    printf ( "\n%d %s %d", x, t, n ) ;
}
```

The output will look as follows

```
111 AJAY 90
```

It must be noted that in the declaration of the structure, **name** has been declared as array. Therefore, when we call the function **display()** using,

```
display ( st.rollno, st.name, st.m1 );
```

we are passing the base address of the array **name** , but the values stored in **rollno and m1**. Thus, this is a mixed call—a call by reference as well as a call by value.

It can be immediately realized that to pass individual elements would become more tedious as the number of structure elements go on increasing. A better way would be to pass the entire structure variable at a time.

This method is shown in the following program.

```
struct student
{
int rollno ;
char name[25] ;

int m1;
};
main()
{
    struct student st = { 111, "AJAY", 90 } ;
    display ( st );
}
display ( struct book b )
{
printf ( "\n%d %s %d", b.rollno, b.name, b.m1 ) ;
}
```

And here is the output...

```
111 AJAY 90
```

We observe that here the calling of function **display()** becomes very simple and elegant,

```
display ( b1 );
```

It should also be noted here that the structure has been defined outside the main() function so that it is available to all functions defined within the main() function.

10. UNIONS

Unions have a great deal of similarities with structures however there is one main difference and that is in the manner of data storage that they implement. In structures each member has its own storage location, in unions all members share a common storage location. The consequence of this is even if a union may contain many members of different data type it can handle only one member at a time. Like structures unions can also be declared with the keyword union.

The following example illustrates the use of union.

```
main() {
    union data
    {
        int integer;
        char character;
    } data_t;

    data_t input;
    input.integer = 2;
    printf("input.integer = %d\n",input.integer);

    input.character = 'A';
    printf("input.character = %c\n",input.character);
}
```

The above example declares a variable data_t of type union data. The union contains two members namely integer and character. Note however that only one of them can be used at a time. This is due to the fact that only one location is allocated for a union variable. The compiler allocates space large enough to hold the largest member in the union. To access a union member the same syntax as structures is used.

11. UNIT END EXERCISES

1. Create a structure to specify data on students given below:
Roll number, Name, Department, Course, Year of joining. Assume that there are not more than 250 students in the college.
 - (a) Write a function to print names of all students who joined in a particular year.
 - (b) Write a function to print the data of a student whose roll number is given.
2. Create a structure to specify data of customers in a bank. The data to be stored is: Account number, Name, Balance in account. Assume maximum of 200 customers in the bank.
 - (a) Write a function to print the Account number and name of each customer with balance below Rs. 100.
 - (b) If a customer requests for withdrawal or deposit, it is given in the form:

Acct. no, amount, code (1 for deposit, 0 for withdrawal) Write a program to give a message, "The balance is insufficient for the specified withdrawal".

3. A record contains name of cricketer, his age, number of test matches that he has played and the average runs that he has scored in each test match. Create an array of structure to hold records of 20 such cricketer and then write a program to read these records and arrange them in ascending order by average runs.
4. There is a structure called **employee** that holds information like employee code, name, date of joining. Write a program to create an array of the structure and enter some data into it.
5. Then ask the user to enter current date. Display the names of those employees whose tenure is 3 or more than 3 years according to the given current date.



Pointers

11

Unit Structure

- 11.0 Objectives
- 11.1 Introduction
 - 1. Fundamentals
 - 2. Pointer Variables
 - 3. Referencing and de-referencing
- 11.2 Chain of Pointers
- 11.3 Pointer Arithmetic
- 11.4 Pointers and arrays
 - 1. Pointers and Strings
 - 2. Array of Pointers
- 11.5 Pointers as function arguments
 - 1. Functions returning pointers
 - 2. Pointers to function
- 11.6 Pointer to structure
 - 11.6.1 Pointers within structure
- 11.7 Dynamic Memory allocation
 - 1. malloc()
 - 2. free()
 - 3. sizeof operator

4. calloc()
5. realloc()
8. Summary
9. List of References
10. Unit End Exercises

11.0 Objectives

After going through this unit, you will be able to:

- Understand the pointers
- Write dynamic programs
- Understand strength of pointers

11.1 Introduction:

Pointer is one of the features of C, which is most difficult to understand. Obviously, people around you might be saying this. But, once you understand basics of Pointers nothing is really hard. Before going to Pointers, it's important to understand organization of Computer Memory. As we know, memory is organized in sequence of bytes and 1 byte = 8 bits. These bytes have given fixed unique numbers (i.e. address) starting from 0 to max capacity of memory as shown in figure 11.1 Consider you have declared a variable as *int a = 100;*

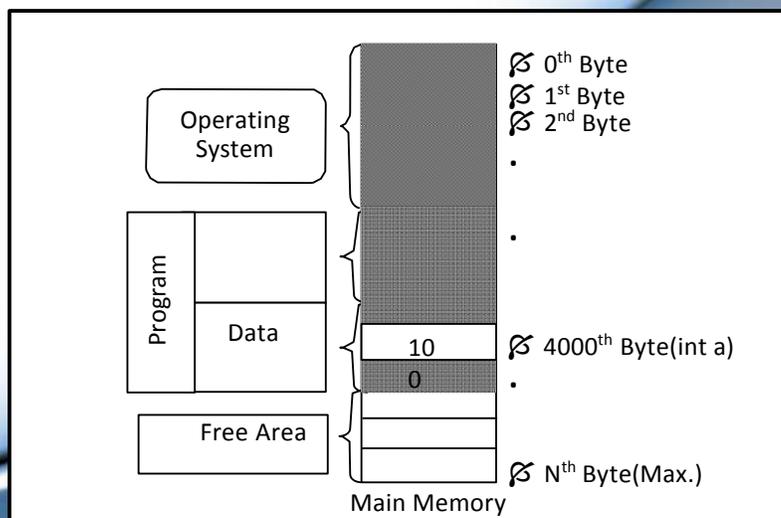


Fig. 11.1: Main Memory with address

When you declare the above variable 'a' it means occupy a space in memory for storing integer value 100 and give the name as 'a' to this space. Here it is stored at 4000th location is just imaginary address. It will be different in actual!

11.1.1 Fundamentals of Pointer

Definition of Pointer:

Pointer variables are defined as variables that contain the memory addresses of data or executable code.

As we know, every variable is stored somewhere into computer memory. It has a unique address wherever it is stored. Now the variable which can store this address is nothing but the pointer. For example: Every person in the world is living at particular location. If you want to store the address of a particular person, you may use address book or digital diary or notebook. We can say all these tools are pointers. As they are locating some address, they are known as Pointers.

11.1.2 Pointer Variables

If you clear with the Pointers concept, next question in your mind will be "how to declare a Pointer variable?"

Syntax: `data_type *pointer_variable_name;`

e.g. `int *abc;`
 `float *pqr;`

Just observe the syntax carefully, first you will find `data_type`, which is the type of data to which you point using pointer variable, followed by `*`, denotes it is a pointer variable and name of the pointer variable. When you declare a pointer variable as integer, that doesn't mean pointer variables type is int, it means it can point to an integer or it has ability to store the address of integer variable. In above examples the pointer variable 'abc' can store the address of an integer variable and 'pqr' can store the address of a float variable. Then what's the type of a pointer variable? It's 'Positive Integer'. Yes! Pointers variables are always type of positive integers. As it stores the address of another variable, it can not be of type float or character, only positive integers. To understand this concept clearly, take a glance at computer memory in fig. 11.2 and forget about Operating System and Code part from the memory. Concentrate only on data part.

```
Example:    int a = 100;

            int *a_ptr = &a;

            float pi = 3.14;

            float *pi_ptr = &pi;
```

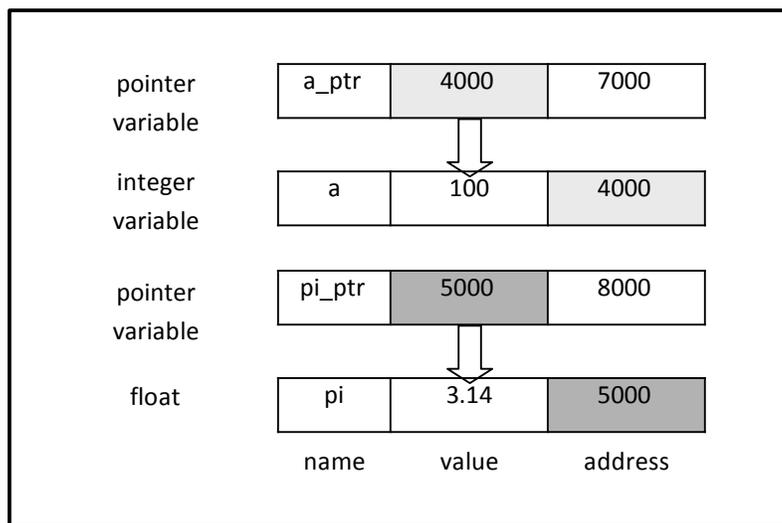


Fig. 11.2: Variables and addresses

In above example, variable a content value 100 and its address is 4000, which is stored in pointer variable a_ptr. Similarly pi content value 3.14 and its address is 5000 which is stored in pointer variable pi_ptr. Here you will find one more new operator, i.e. &(ampersand). It is known as address operator. The & (ampersand) operator gives the address of a variable. E.g '&a' will return the address of variable 'a'.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a=100;      // Integer variable 'a' is declared
    int *a_ptr;    // Pointer variable 'a_ptr' is declared which can point to
integer
    float pi=3.14; // Float variable 'pi' is declared
    float *pi_ptr; // Pointer variable 'pi_ptr' is declared which can point to
Float

    clrscr();

    a_ptr = &a; // address of 'a' is stored in a_ptr;
    pi_ptr = &pi; // address of 'pi' is stored in pi_ptr;

    printf("\n\t Value contain in a => %d",a);
    printf("\n\t Address of a is => %u",a_ptr);
    printf("\n\t Value contain in pi => %f",pi);
    printf("\n\t Address of pi is => %u",pi_ptr);

    printf("\n\t Value contain in a => %d",*a_ptr);
    printf("\n\t Value contain in pi => %f",*pi_ptr);

    getch();
}
```

Using * operator, you can access the value stored at that location. In example 11.1, observe last two printf statements. You can get the value stored in a using a_ptr. When you write *a_ptr, that means you want to access value stored at address in a_ptr. As a_ptr contains 65524, which is address of variable a, so *a_ptr will return you value 100 and a_ptr will return you 65524. It's not necessary that it will always return you 65524; it can be different in your system.

11.1.3 Dereferencing Pointer:

*It is a procedure to obtain or manipulate data in the memory using pointer. It can be achieved using * operator. In this case, * is known as the indirection operator. From previous example, you know that, using * operator, you can access the value stored at the particular address. Similarly you can also manipulate that value using pointer. This process of accessing and manipulating data using pointer is known as Dereferencing Pointer. To understand it clearly just go through the following exa*

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

void main()
{
    int num=400;

    int *p;

    clrscr();

    p=&num; // store the address of num in pointer p
    printf("\n\t Value of num is => %d",num);

    *p=100; // manipulating the value of num using pointer p
    printf("\n\n\t Value of num is => %d",num);

    *p+=100; // similar as num = num + 100;
    printf("\n\n\t Value of num is => %d",num);

    getch();
}
```

In above example, initially value 400 is stored in variable num. Then the address of num is stored in pointer variable p using & operator. Then, value of variable num is changed from 400 to 100 and finally, value of num is incremented by 100 using pointer p.

11.2 Chain of Pointers:

Remember the previous example; you store another person's address in your digital diary or notebook. Now imagine some other person may have stored your address in his diary and other third person may have stored second person's address in his diary. Similarly, if you have stored address of a variable in pointer, you can store this pointer's address in another pointer, this pointer's address in another pointer i.e. a pointer to pointer's pointer's..... You can use unlimited pointers. To get rid of this, execute the following program in your system and remember once again, address of variable may printed different in your system as it depends on available memory in system.

```
#include<stdio.h>

#include<conio.h>

void main()

{

    int i=50;

    int *ptr;

    int **ptr2ptr;

    int ***ptr2ptr2ptr;

    clrscr();

    ptr = &i;

    ptr2ptr = &ptr;

    ptr2ptr2ptr = &ptr2ptr;

    printf("\n Value of i is => %d",i);

    printf("\n\n Adress of i is => %u OR value stored in ptr is =>
%u",&i,ptr);

    printf("\n\n Adress of ptr is => %u OR value stored in ptr2ptr is => %u",
&ptr, ptr2ptr);

    printf("\n\n Adress of ptr2ptr is => %u OR value stored in ptr2ptr2ptr is
=> %u",&ptr2ptr,ptr2ptr2ptr);

    printf("\n\n Value of i is => %d",*ptr);

    printf("\n\n Value of i is => %d",**ptr2ptr);

    printf("\n\n Value of i is => %d",***ptr2ptr2ptr);

    getch();

}
```

```
Turbo C++ IDE
Value of i is => 50
Adress of i is => fff4 OR value stored in ptr is => fff4
Adress of ptr is => fff2 OR value stored in ptr2ptr is => fff2
Adress of ptr2ptr is => fff0 OR value stored in ptr2ptr2ptr is => fff0
Adress of ptr2ptr2ptr is => ffee
Value of i is => 50
Value of i is => 50
Value of i is => 50_
```

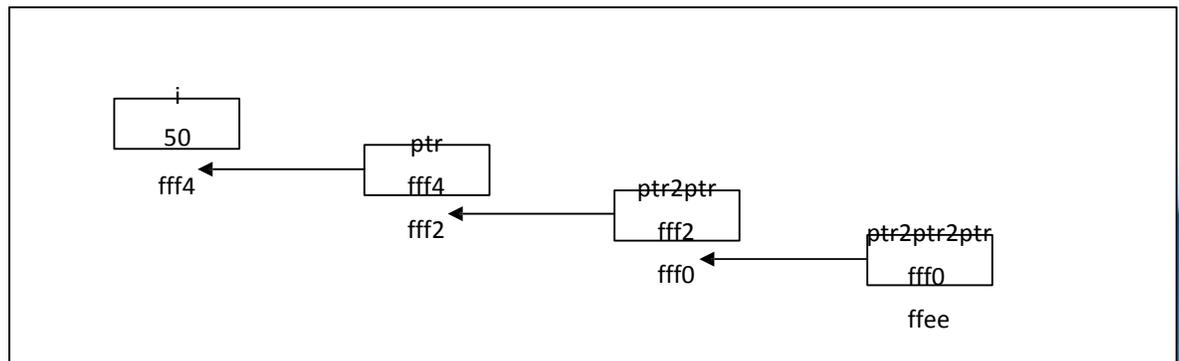


Fig.11.3 Logical View of Memory after executing above program

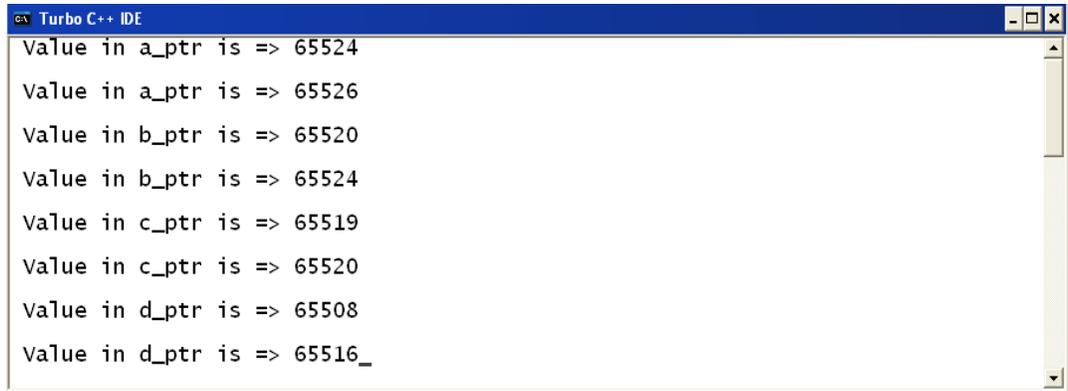
11.3 Pointer Arithmetic:

Yes! C permits you to do arithmetic on Pointers. But be careful while doing that. Cause only you have to take care of output of program. When you increase the pointer by one it immediately shift to next location in memory of the type to which it points. E.g if you have a pointer with address value 4004 which points to an integer. When you increment this pointer, it points to the next integer i.e 4006, because integer requires 2 bytes to store it in memory. Then, how many bytes are accessed by a pointer for other types? Following table illustrate this.

Data Type	No. of bytes
int	2 consecutive bytes
float	4 consecutive bytes
double	8 consecutive bytes
char	1 consecutive byte

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a=200, *a_ptr;
    float b=2.3, *b_ptr;
    char c='J', *c_ptr;
    double d=5.5, *d_ptr;

    clrscr();
    a_ptr=&a;
    printf(" Value in a_ptr is => %u",a_ptr);
    a_ptr++; // increamenting a_ptr
    printf("\n\n Value in a_ptr is => %u",a_ptr);
```



```

Turbo C++ IDE
Value in a_ptr is => 65524
Value in a_ptr is => 65526
Value in b_ptr is => 65520
Value in b_ptr is => 65524
Value in c_ptr is => 65519
Value in c_ptr is => 65520
Value in d_ptr is => 65508
Value in d_ptr is => 65516_

```

In the above example, as you have seen, pointer can be incremented; similarly it can be decremented also.

11.4 Pointers and Arrays:

Array is a collection of similar elements which are stored in adjacent to each other. Whenever you want to store multiple values in single variable of the same type, you can use an Array. Each value in an array has index number and location address.

E.g `int a[5] = {10,20,30,40,50}`

2020	2022	2024	2026	2028
10	20	30	40	50
a[0]	a[1]	a[2]	a[3]	a[4]

In above example, there are 5 elements in the array. Array index always starts from 0th position. Therefore we have array indexes from a[0] a[4]. The starting memory address of an array is known as base address. In above case, it is 2020 and as each element takes 2 bytes to store, next address is 2022 and so on.

You can access element of arrays using Pointers. If you increments the value of Pointer then it points to next location i.e next element of an array. If you have a pointer which points to base address i.e. 1st element a[0], after incrementing the pointer it will point to the next element a[1].

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a[5] = {10,20,30,40,50};
    int *a_ptr,i;
    clrscr();
    a_ptr = &a[0];
    for(i=0;i<=4;i++)
    {
        printf("\n\n Element No.%d, Address is %u, Value is %d",i,&a[i],a[i]);
        printf("\n\n Using Pointer Element No.%d, Address is %u, Value is %d",i,a_ptr,*a_ptr);
        a_ptr++;
    }
    getch();
}
```

```

Turbo C++ IDE

Element No.0, Address is 65516, Value is 10
Using Pointer Element No.0, Address is 65516, Value is 10
Element No.1, Address is 65518, Value is 20
Using Pointer Element No.1, Address is 65518, Value is 20
Element No.2, Address is 65520, Value is 30
Using Pointer Element No.2, Address is 65520, Value is 30
Element No.3, Address is 65522, Value is 40
Using Pointer Element No.3, Address is 65522, Value is 40
Element No.4, Address is 65524, Value is 50
Using Pointer Element No.4, Address is 65524, Value is 50_

```

In the above example, the base of the 0th element has been assigned to the pointer. The same thing, you can achieve in a different way. You can access the base address i.e address of 0th element by specifying just array name. That means, in above case, '&a[0]' is same as 'a'.

```

#include<stdio.h>
#include<conio.h>
void main()
{
    int a[5] = {10,20,30,40,50};
    int i;
    clrscr();
    for(i=0;i<=4;i++)
    {
        printf("\n\n Element No.%d, Address is %u, Value is %d",i, a+i,*(a+i));
    }
    getch();
}

```

11.4.1 Pointers and Strings:

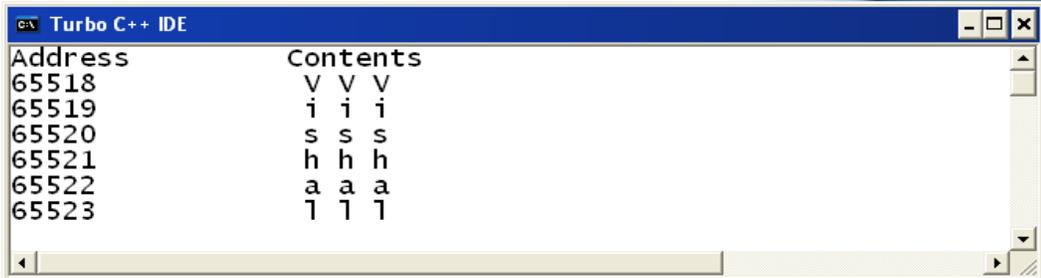
In the previous topic, you have seen how you can use Pointers with arrays. In above example, we have used array of integers. Similarly, you can use array of characters, known as Strings. Strings are set of characters, stored adjacent to each others. Every string terminates with '\0' (null) character, which indicates the end of the String. You don't need to specify this null character. C compiler will take care of it. It will inserted automatically at the end of the string. E.g. `city[] = "Mumbai";`

city[0]	city[1]	city[2]	city[3]	city[4]	city[5]	city[6]
M	u	m	b	a	i	\0
1004	1005	1006	1007	1008	1009	1010

Fig.11.4 Logical View of a String in Memory

In above example, 'city' is array of characters. You can access the base address using variable 'city'. After base address, each location is shifted by one byte. Because, you know now, "character requires one byte to store it in memory". The following program will demonstrate how we can use the strings.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    char name[]="Vishal";
    int i=0;
    clrscr();
    printf("Address\t  Contents");
    while(name[i]!='\0') // Loop will continue until end of the string
    {
        printf("\n%u\t\t %c %c %c", (name+i), *(name+i), i[name], name[i]);
        i++;
    }
    getch();
}
```



Turbo C++ IDE

Address	Contents
65518	V V V
65519	i i i
65520	s s s
65521	h h h
65522	a a a
65523	l l l

Here, first you will find address of each character, which is printed using 'name+i' (base address + value of i). Next is character itself using *value at the operator*, `*(name+i)`. Again the same character is printed twice using `'i[name]'` and `'name[i]'`. Yes, both statements will result same. Because compiler will interpret `'*(i+name)'` and `'*(name+i)'`.

There are some commonly used library functions to work on string. Most of them manipulate string using pointers. Some of them are: `strlen()`, `strcat()`, `strcpy()` and `strcmp()`.

`strlen()` function:

This function is used to find out the length of the string. You need to pass the base address of the string to this function. It will return the length of the string.

`strcat()` function:

Thus function is used to concatenate two strings. That means, if you have one string as "Hello" and another string as "brother", this function will generate new string as "helloworldbrother".

`strcpy()` function:

This function is used to copy one string to the other string. It requires two arguments, pointer to the first character of destination string and pointer to the first character of source string.

`strcmp()` function:

This function is used to compare two strings, whether they are equal or not. If both the strings are equal, it will return an integer value 0. The function compares two strings character by character. If it finds the different characters, it will return the difference between ASCII values of these characters.

The following program will demonstrate, how above functions can be used with strings?

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

void main()
{
    char str1[10], str2[]="Hello ", str3[]="Mumbai",str4[10];

    int len,d;

    clrscr();
    printf("Enter your name => ");
    scanf("%s",str1);

    len=strlen(str1);// Returns the length of str1
    printf("\nLength of str1 is %d",len);

    strcat(str2,str1); // Concatenates two string
    printf("\nContents of str2 => %s",str2);

    strcpy(str4,str3); // Copy str3 into str4
    printf("\nContents of str3 => %s",str3);
    printf("\nContents of str4 => %s",str4);

    if(strcmp(str3,str4)==0) // Compares two strings
    {
        printf("\nstr3 and str3 are equal.");
    }

    getch();
}
```

```

Turbo C++ IDE
Enter your name => Makarand

Length of str1 is 8
Contents of str2 => Hello Makarand
Contents of str3 => Mumbai
Contents of str4 => Mumbai
str3 and str4 are equal.

```

11.4.2 Array of Pointers:

As you know, an array is a collection of similar data type elements. Then what's the array of Pointers? It is the collection of Pointers. Pointers can store the address. So, collection of such consecutive addresses into single variable is

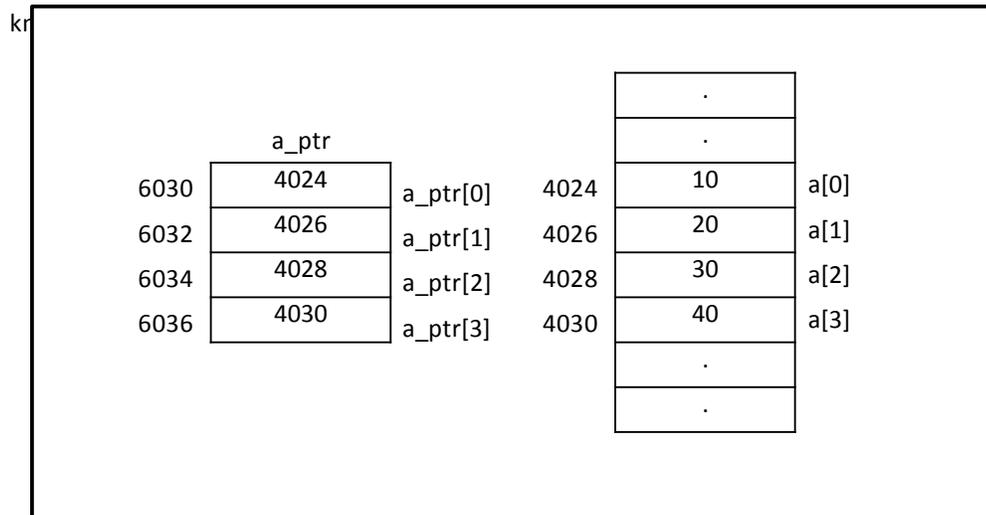


Fig. 11.5: Array of Pointers

In fig.11.5, there are two arrays. One is array of integers and the other one is array of Pointers. In array 'a', we have stored 4 integer numbers (10, 20, 30, 40). In array of pointers, we have store addresses of each element of an integer array. This can be illustrated using next program.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a[]={10,20,30,40,50};
    int *a_ptr[5],i;

    clrscr();
    for(i=0;i<5;i++)
    {
        a_ptr[i]=&a[i];
    }
    printf("Variable Name  Contents  Address");
    for(i=0;i<5;i++)
    {
        printf("\n a[%d] \t\t %d \t  %u",i,a[i],&a[i]);
    }
    printf("\nPointer Name  Contents  Address");
    for(i=0;i<5;i++)
    {
        printf("\n a_ptr[%d] \t %u \t  %u",i,a_ptr[i],&a_ptr[i]);
    }
    getch();
}
```

Thank You



Introduction to algorithm & C Part-5



Variable Name	Contents	Address
a[0]	10	65516
a[1]	20	65518
a[2]	30	65520
a[3]	40	65522
a[4]	50	65524
Pointer Name	Contents	Address
a_ptr[0]	65516	65506
a_ptr[1]	65518	65508
a_ptr[2]	65520	65510
a_ptr[3]	65522	65512
a_ptr[4]	65524	65514

5. Pointers as Function Arguments:

There are two ways to pass arguments to the function.

1. Call by Value
2. Call by Reference

However, if you use the first method, the actual arguments in the calling method will be copied into formal arguments of the called function. That means, it will create one more copy of each argument in memory for called function. Therefore, if you make changes into formal arguments of called method, it will not affect to the actual arguments in calling method.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int w=10, h=4;
    clrscr();
    printf("Before Calling: w = %d, h = %d ",w,h);
    ifun(w,h);
    printf("\n After Calling: w = %d, h = %d ",w,h);
    getch();
}
ifun(int w,int h)
{
    w++; h++;
}
```

If you have executed above program, you can easily find that we are passing two variables w and h to the function 'ifun'. We have incremented values of w and h in function 'ifun'. But as C compiler, creates two new copies of these variable for function 'ifun', the changes made in function 'ifun' won't be reflected in values of calling function 'main'. Therefore we get same result after calling 'ifun' function.

To solve above problem, we can use the second method. i.e. Call By Reference. In this method, instead of passing actual values of arguments, we send reference (addresses) of arguments. Now, whatever changes made in called function, gets reflected in the calling function also. It can illustrated using following program.

In this program, we are passing addresses of variable w and h to the function 'ifun', which are collected into the pointer variables w and h in the function

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

void main()
{
    int w=10, h=4;

    clrscr();

    printf("Before Calling: w = %d, h = %d ",w,h);

    ifun(&w,&h);

    printf("\n After Calling: w = %d, h = %d ",w,h);

    getch();
}

ifun(int *w,int *h)
{
    *w=*w+1;

    *h=*h+1;
}
```

'ifun'. Then we have incremented the value of w and h using *(value at



```
Turbo C++ IDE
Before Calling: w = 10, h = 4
After Calling: w = 11, h = 5 _
```

operator). Therefore the changes also gets reflected into the function 'main'

11.5.1 Functions Returning Pointers:

Now, you are master in passing values and reference to the function. But note one thing, you can also return pointers to the functions. The only condition is that, it should be mentioned explicitly in the calling function and in the function declaration. The next program will illustrate you how you can return pointer from a function.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int *p;

    int* ifun();    // Function Declaration

    clrscr();

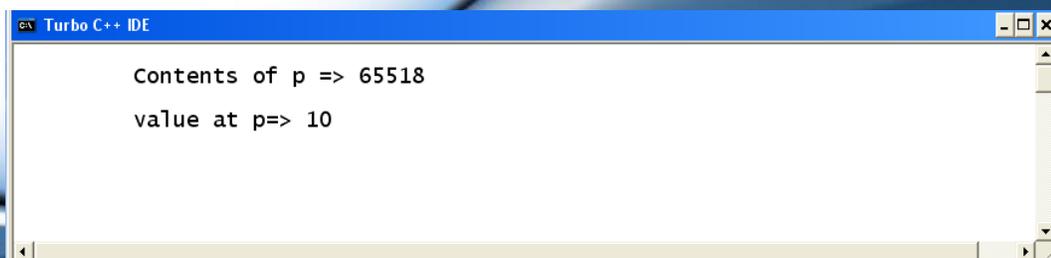
    p=ifun();

    printf("\n\t Contents of p => %u \n\n\t value at p=> %d",p,*p);

    getch();
}

int* ifun()
{
    int num=10;

    return(&num);
}
```



```
Turbo C++ IDE
Contents of p => 65518
value at p=> 10
```

11.5.2 Pointers to Function:

Looking at the topic name, there must be a question in your mind; can we store address of a function? Yes! This is one of the powerful capabilities of C that it allows you to store address of a function into pointer through which you can also invoke the function.

So, what to do for accessing functions using pointers? Nothing much! To get the address of a function, you need to mention only name of the function. Declaration of the pointer variable is as follows:

```
<return type> (*pointer_name)();
```

Here, 'return type' is the data type, which is returned by the function to which it points. The next program will illustrate this:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int num;

    int sum();

    int (*funptr)();

    clrscr();

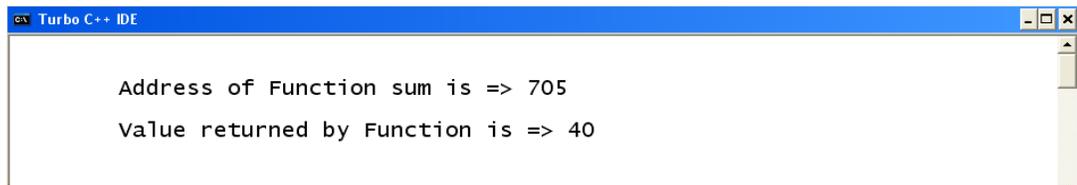
    funptr=sum; // Address of sum is assigned to funptr
    printf("\n\n\tAddress of Function sum is => %u",funptr);

    num = (*funptr()); //Invoking function using pointer
    printf("\n\n\tValue returned by Function is => %d",num);

    getch();
}

int sum()
{
    int a=10,b=30;

    return(a+b);
}
```



```

Turbo C++ IDE
Address of Function sum is => 705
Value returned by Function is => 40

```

11.6 Pointer to a Structure:

There are numerous data types available in C like int, float, char, etc. But are these data types sufficient to store information? If you want to store information of an employee which contains employee id, name, age etc. Well, don't worry. C provides user defined types known as Structure in which different data type can be group together.

e.g.

```

struct employee
{
    int eno;
    char ename[20];
    int age;
};

employee e1;

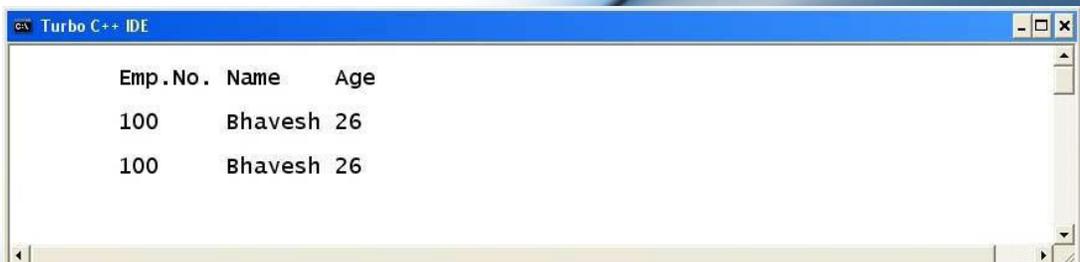
```

In above example, we have defined employee structure. 'e1' is a variable name of the type employee. If you want to access 'eno', you can use e1.eno, similarly e1.ename for 'ename' and e1.age for age.

We have seen, how to point to an int or float, can we point to structures also. Yes. Same as normal data type we can point to structure as well. But the accessing of structure elements using pointers is different. To access structure element using structure variable we use '.'(dot) operator. While to access structure element using pointer variable you have to use '→' operator. That means, on the left hand side of '.'(dot), it should structure variable and on the left hand side of '→', it should be pointer to the structure.

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

void main()
{
    struct employee
    {
        int eno;
        char ename[20];
        int age;
    };
    struct employee e1={100,"Bhavesh",26};
    struct employee *p;
    p=&e1; // assign address of structure to pointer
    clrscr();
    printf("\n\tEmp.No.\tName\tAge");
    printf("\n\n\t%d\t%s\t%d",e1.eno,e1.ename,e1.age);//using structure
    printf("\n\n\t%d\t%s\t%d",p->eno,p->ename,p->age);//using pointer
    getch();
}
```



```
c:\ Turbo C++ IDE
```

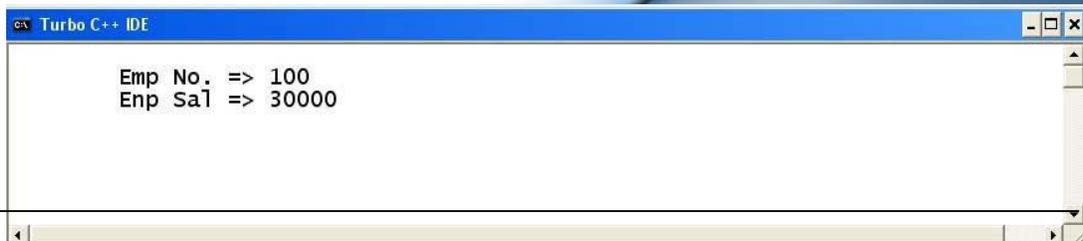
Emp.No.	Name	Age
100	Bhavesh	26
100	Bhavesh	26

11.6.1 Pointers within Structure:

Various data types like int, float, char etc. can be group together using Structures. Can we group Pointer also with these structures? Yes. Similar to other data types, you also add Pointer variables inside the structures.

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

void main()
{
    struct emp
    {
        int eno;
        int *p;
    };
    struct emp e1;
    int sal=30000;
    clrscr();
    e1.eno=100;
    e1.p=&sal;
    printf("\n\tEmp No. => %d \n\tEmp Sal => %d",e1.eno,*e1.p);
    getch();
}
```



The screenshot shows a Turbo C++ IDE window with the following output:

```
Emp No. => 100
Emp Sal => 30000
```

11.7 Dynamic Memory Allocation:

Consider you are developing a C program for payroll system. You have declared an array of 50 integers to store employee numbers. C compiler will automatically reserve 100 bytes of memory. Because data type is integer and one integer takes 2 bytes. But at the runtime of program, you have stored information for 30 employees only. That means, you are wasting 40 bytes of memory. Or, it may happen, you want to store information for more than 50 employees. In that case, you can not increase the array size at run time. Don't worry. C has solution for this problem also – Dynamic memory allocation! You can allocate memory space at the runtime from free space. There are some standard library functions in C, which allows you to allocate and de-allocate memory space, `malloc()`, `calloc()`, `realloc()` and `free()` are important ones. We will how these functions work in details:

11.7.1 malloc()

The function `malloc()` is used to allocate a block of memory from the free space. It request for a block to the operation system. If it gets the block of memory, it returns the starting pointer of the block; otherwise it returns the value `NULL`.

```
Prototype: void *malloc(size_t size);
```

2. free()

Whenever you allocate memory using 'malloc()' or 'calloc()' function, C compiler reserves that much space for you. After using this space, you should de-allocate this space for future use. It can be achieved using `free()` function.

```
Prototype: void free(void *ptr);
```

3. sizeof Operator

'sizeof' is a unary operator which is used to calculate the size of any data type. It can be used with all primitive data types like `int`, `float`, `long`, etc.

The following program will illustrate use of `malloc()`, `free()` functions and `sizeof()` operator.

```
#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

void main()

{

    char *str;

    int i,len;

    clrscr();

    printf("Enter the length of the string => ");    scanf("%d",&len);

    str = (char*)malloc(sizeof(char)*len);

    if(str==NULL)

        exit(0);

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

    {

        str[i] = rand()%26+'a';

    }

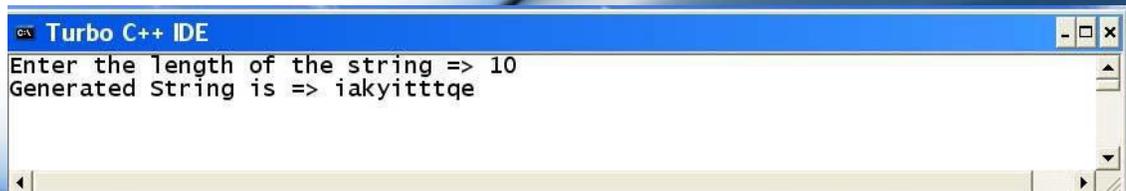
    str[i]='\0';

    printf("Generated String is => %s",str);

    free(str);

    getch();

}
```



The screenshot shows a Turbo C++ IDE window with a blue title bar. The text area contains the output of the program: "Enter the length of the string => 10" followed by "Generated String is => iakyitttqe". The IDE has a standard Windows-style interface with a scroll bar on the right and a status bar at the bottom.

11.7.4 calloc()

The function 'calloc()' is same as the function 'malloc()' but it requires two argument. First is number of variables and second is size required for each variable.

```
Prototype : void *calloc(size_t num, size_t size);
```

11.7.4 realloc()

If you allocate memory using function malloc() or calloc(), you can reallocate this size using realloc() function. That means you can expand or reduce the amount of bytes allocated using realloc() function.

8. Summary

- A pointer is a variable which can store memory address.
- A pointer can store address of all primitive data types and user defined data types.
- A pointer can store address of another pointer variable.
- An array of character is known as string.

9. List of References

- Understanding Pointers In C – Yashavant Kanetkar
- Mastering C – K. R. Venugopal, S. R. Prasad

10. Unit End Exercises

- Write a program, which accepts a string and converts all characters into upper or lower case.
- What is meant by dereferencing?
- What do you understand by a pointer to a pointer?
- Explain the difference between an array and a pointer.



FILE HANDLING

Unit Structure

1. Objectives
2. Introduction
 1. Opening files- fopen() function
 2. Writing to File – fprintf() function
 3. Closing File – fclose() function
3. Types of Files
4. File Handling Function
 1. fgetc()
 2. fputc()
 3. fgets()
 4. fputs()
 5. fscanf()
 6. getw()
 7. putw()
 8. fread()
 9. fwrite()
 10. fseek()
5. Summary
6. List of References
7. Unit End Exercises

12.0 OBJECTIVES

After going through this unit, you will be able to,

- Store data in files
 - Read data from files
 - Understand File Handling Functions
-

12.1 INTRODUCTION

Whenever you run your program, where does C stores your data? Definitely, in Main Memory(RAM). But, as soon as you close the program, the entire data will be erased from Main Memory. Then, what to do for storing this data permanently? Is there any solution for this? Yes! You can store this data on hard drive using Files. C provides very efficient method to store your data into files. There are numerous standard library functions in C for file handling. They are easy to use and efficient to handle data.

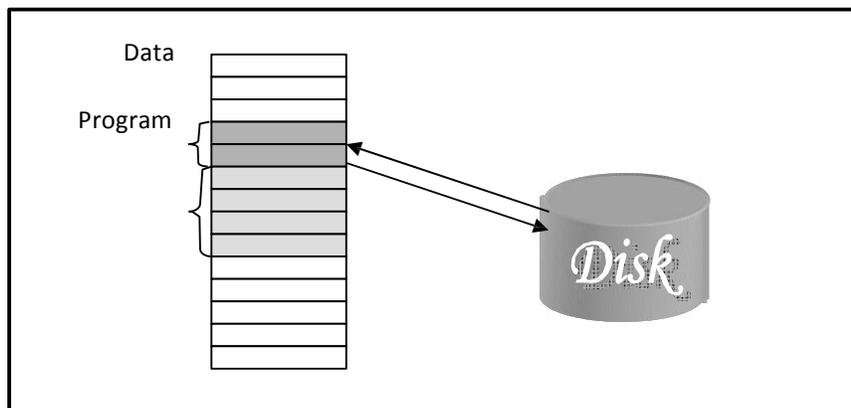


Fig. 12.1 File I/O Operation

C compiler deals with two types of I/O function, Console I/O and File I/O. In general, Console I/O functions refer to read from the keyboard and write to the display screen. The main difference between these two types of I/O function is, program can read/write in the same file.

To store data into files, any C program has to follow the sequence:

- Open the file
- Read or Write to the file
- Close the file

12.1.1 Opening File- fopen() function:

If you want to perform any file I/O operation, you must open the file using function 'fopen()'. The function takes two parameters as arguments. One is name of the file and the second is 'mode' in which you want to open the file. 'Mode' specifies whether you want to open file as 'Read-Only', 'Write-Only', 'Both Read and Write', 'Append', 'Overwrite'.

For example:

```
FILE *fp;  
  
fp = fopen("test.txt","w");
```

The file structure is defined in the file stdio.h file. The first line in the example is declaration of the pointer to the file structure. The function fopen() will open the file 'test.txt' in write mode. If it opens the file successfully, it returns Pointer the given file. This Pointer will be used for other operations on file such as reading or writing.

12.1.2 Writing to File – fprintf() function

Once you have opened file, you are ready to do operations such as reading or writing. To write into file, 'fprintf()' function is used. It requires two parameters. First is pointer to file structure and second is the message or text that you want to write.

For example:

```
fprintf(fp,"Hello World!");
```

12.1.3 Closing File – fclose() function

Once you finished your work (reading/writing) with files, you should close the file. The standard C library has provided 'fclose()' function for closing the file.

For example:

```
fclose(fp);
```

The following program will illustrate above 3 functions:

```
#include<stdio.h>

void main()

{

    FILE *fp;

    fp = fopen("new.txt","w");

    if(fp==NULL)

    {

        printf("Unable to Open...");

        exit();

    }

    fprintf(fp,"Your birth is a mistake, you'll

    spend your whole life trying to correct");

    fclose(fp);

}
```

12.2 TYPES OF FILES

If you are using DOS system, it supports two types of file modes- text and binary. The default is text mode. When you specify the file type as binary, the file I/O functions do not interpret files content but in case of text files, it has to interpret it.

- When you read from a text file, it considers the ASCII '0x1a' as the end of file character.
- In DOS systems, the sequence 0x0d 0x0a on the disk is stored as the newline character for text files.

Accessing files in binary mode is faster, compared to text mode. You can see the files exactly as it is on disk, bytes by bytes.

12.3 FILE HANDLING FUNCTIONS:

12.3.1 fgetc()

The function fgetc() returns the next characters in the file stream. At the end of file, it returns EOF. The syntax of the function is as follows:

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

12.3.2 fputc()

The function fputc() is used to write a character in the file. Syntax is:

```
int fputc(int ch, FILE *fp);
```

12.3.3 fgets()

The function fgets() is used to read entire line(string) from a file. Its syntax is:

```
char * fgets(char * str, int n, FILE *fp);
```

The function requires 3 parameters. First is pointer to an array of characters, where you want to store string. Second is, maximum number of characters to be read. Third is pointer to the FILE.

```
#include <stdio.h>
void main()
{
    FILE * fp;
    char mystring [100];
    fp = fopen ("test.txt" , "r");
    if (fp == NULL) perror ("Error opening file");
    else {
        fgets (mystring , 100 , fp);
        puts (mystring);
        fclose (fp);
    }
}
```

12.3.4 fputs()

The function fputs() is used to write an entire line(string) into the file. The syntax is:

```
int fputs(const char * str, FILE *fp);
```

12.3.5 fscanf()

Using fscanf() function, you can read the data into a specific format. Its syntax is:

```
int fscanf(FILE *fp, const char * format,...);
```

For Example:

```
int num;
```

```
fscanf(fp,"%d" ,&num);
```

12.3.6 getw()

The function `getw()` returns the next integer in the file. Avoid use of this function with text files. The syntax is:

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

12.3.7 putw()

The function is used to write an integer into the file. The syntax is:

```
int putw(int num, FILE *fp);
```

12.3.8 fread()

The function `fread()` is used to read an array of elements of specified size. The syntax is:

```
size_t fread(void * ptr, size_t size, size_t count, FILE *fp);
```

12.3.9 fwrite()

This function is used to write an array of elements to the file. Syntax is as follows:

```
size_t fwrite(const void * ptr, size_t size, size_t count, FILE *fp);
```

12.3.10 fseek()

The function is used to set the pointer of file to the specified position. The syntax of the function is:

```
int fseek(FILE *fp, long int offset, int origin );
```

Here 'offset' refers to the number of bytes from origin. The following constants are defined for the origin.

SEEK_SET	Beginning of the file.
SEEK_CUR	Current position of the file Pointer
SEEK_END	End of the File.

For Example

```
#include <stdio.h>
int main ()
{
    FILE * fp;
    fp = fopen ( "test.txt" , "w" );
    fputs ( "Hello World." , fp);
    fseek ( fp , 9 , SEEK_SET );
    fputs ( " Sagar" , fp);
    fclose ( fp );
    return 0;
}
```

12.4 SUMMARY

- DOS systems can handle two types of file, text and binary.
- Working with binary files is faster than text files.
- Functions like fgetc(), fgets(), fread() etc. are used to read data from the file.
- Functions like fputc(), fputs(), fwrite() etc. are used to write data to the file.

12.5 LIST OF REFERENCES

- Mastering C – K. R. Venugopal, S. R. Prasad
- www.cplusplus.com

12.6 UNIT END EXERCISES:

- What is a file?
- What are three steps that are followed while accessing a file?
- Write a program to convert the case of alphabets in a text file. Uppercase letters should be converted to lower case and vice versa.



LINEAR LINK LIST

Unit Structure

1. Objectives
2. Introduction
- 13.3 Description of Linked Lists
 1. Array Review
 2. Linked Lists
- 13.4 Basic Operations
 - 13.4.1 Traversing the Link List
 - 13.4.2 Searching the Link List
 - 13.4.3 Inserting a node into the Link List
 - 13.4.4 Removing a node from the Link List
- 13.5 Implementation using C
 - 13.5.1 Pointer Refresher
 - 13.5.2 Allocating and Freeing Dynamic variables
 - 13.5.3 Creating a node
 - 13.5.4 Inserting a node
 - 13.5.5 Removing a node
 - 13.5.6 Traversing a list
 - 13.5.7 Searching a list
 - 13.5.8 Implementation of Linear Linked List
- 13.6 Summary
- 13.7 Unit End Exercises
 - 13.7.1 Questions
 - 13.7.2 Programming Projects

1. OBJECTIVES

After going through this chapter you will be able to

- Define a Linear Link List and list its features.
 - Understand the advantages & shortcomings of link list over an array.
 - Differentiate between Link List & Array.
 - Write & Explain the basic operations of Linear Link List.
 - Understand how to implement a link list.
 - Write a program in C to implement linear link list.
-

2. INTRODUCTION

Linked lists are some of the most fundamental data structures. Linked lists consist of a number of elements grouped, or *linked*, together in a specific order. They are useful in maintaining collections of data, similar to the way arrays are often used. However, linked lists offer important advantages over arrays in many cases. Specifically, linked lists are considerably more efficient in performing insertions and deletions. Linked lists also make use of dynamically allocated storage, which is storage at runtime. Since in many applications the size of data is not known at compile time, this can be a nice attribute as well.

Some applications of linked lists are:

- ✓ *Mailing lists*: Lists such as ones found in email applications. Since it is difficult to predict how long a mailing list may be, a mailer might build a linked list of addresses before sending a message.
- ✓ *Polynomials*: An important part of mathematics not inherently supported as a datatype by most languages. If we let each element of a linked list store one term, linked lists are useful in representing polynomials (Such as $3x^2 + 2x + 1$)
- ✓ *Memory Management*: An important role of operating systems. An operating system must decide how to allocate and reclaim storage for processes running on the system. A linked list can be used to keep track of portions of memory that are available for allocation.
- ✓ *LISP*: An important programming language in artificial intelligence. LISP, an acronym for LIS Processor, makes extensive use of linked lists in performing symbolic processing.
- ✓ *Linked allocation of files*: A type of file allocation that eliminates external fragmentation on a disk but is good only for sequential access. Each block of a file contains a pointer to the file's next block.
- ✓ *Other data structures*: Some data structures whose implementations depend on linked lists are stacks, queues, sets, hash tables, and graphs.

13.3 DESCRIPTION OF LINKED LISTS

Linked lists and arrays are similar since they both store collections of data. The terminology is that arrays and linked lists store "elements" on behalf of "client" code. The specific type of element is not important since essentially the same structure works to store elements of any type. One way to think about linked lists is to look at how arrays work and think about alternate approaches.

13.3.1 Array Review

Arrays are probably the most common data structure used to store collections of elements. In most languages, arrays are convenient to declare and provide the handy [] syntax to access any element by its index number. The following example shows some typical array code and a drawing of how the array might look in memory. The code allocates an array *int scores[100]*, sets the first three elements set to contain the numbers 1, 2, 3 and leaves the rest of the array uninitialized.

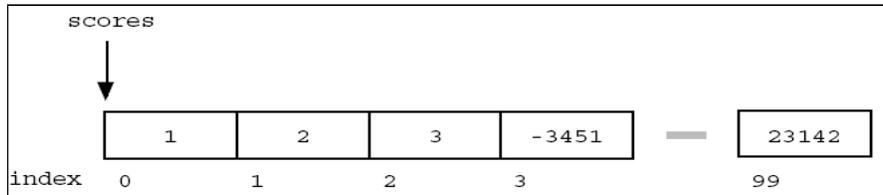
```
void ArrayTest()
{
    int scores[100];
    // operate on the elements of the scores array...

    scores[0] = 1;

    scores[1] = 2;

    scores[2] = 3;
}
```

Here is a drawing of how the scores array might look like in memory. The key point is that the entire array is allocated as one block of memory. Each element in the array gets its own space in the array. Any element can be accessed directly using the [] syntax.



The **disadvantages** of arrays are:

1) The size of the array is fixed — 100 elements in this case. Most often this size is specified at compile time with a simple declaration such as in the example above. With a little extra effort, the size of the array can be deferred until the array is created at runtime, but after that it remains fixed. You can go to the trouble of dynamically allocating an array in the heap and then dynamically resizing it with `realloc()`, but that requires some real programmer effort.

2) Because of (1), the most convenient thing for programmers to do is to allocate arrays which seem "large enough" (e.g. the 100 in the scores example). Although convenient, this strategy has two disadvantages: (a) most of the time there are just 20 or 30 elements in the array and 70% of the space in the array really is wasted. (b) If the program ever needs to process more than 100 scores, the code breaks. A surprising amount of commercial code has this sort of naive array allocation which wastes space most of the time and crashes for special occasions.

3) Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.

13.3.2 Linked Lists

Linked lists are probably the second most commonly used general purpose storage structures after arrays. You can use a linked list in many cases in which you use an array, unless you need frequent random access to individual items using an index. Linked lists have their own strengths and weaknesses, but they happen to be strong where arrays are weak. The array's features all follow from its strategy of allocating the memory for all its elements in one block of memory. Linked lists use an entirely different strategy.

As we will see, linked lists allocate memory for each element separately and only when necessary.

An array allocates memory for all its elements lumped together as one block of memory. In contrast, a linked list allocates space for each element separately in its own block of memory called a "linked list element" or "node". The list gets its overall structure by using pointers to connect all its nodes together like the links in a chain.

Each node contains two fields: a "data" field to store whatever element type the list holds for its client, and a "next" field which is a pointer used to link one node to the next node. The next pointer of the last element is set to NULL, a convenient sentinel marking at the end of the list. The element at the start of the list is its *head*; the element at the end of the list is its *tail* (see Figure 13-1).

Each node is allocated in the heap with a call to `malloc()`, so the node memory continues to exist until it is explicitly deallocated with a call to `free()`. The front of the list is a pointer to the first node. To access an element in a linked list, we start at the head of the list and use the next pointers of successive elements to move from element to element until the desired element is reached. A linear linked list can be traversed in only one direction – from head to tail – because each element contains no link to its predecessor. Therefore, if we start at head and move to some element, and then wish to access an element preceding it, we must start over at the head.

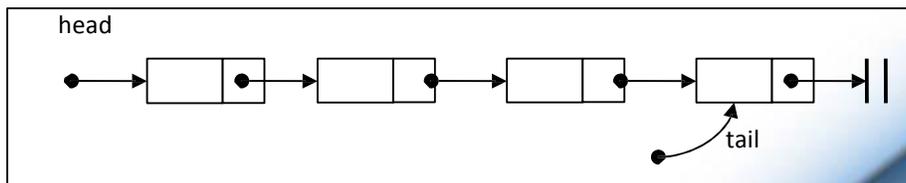


Figure 13-1. Elements linked together to form a linked list

Conceptually, one thinks of a linked list as a series of contiguous elements. However, because these elements are allocated dynamically (using `malloc` in C), it is important to remember that, in actuality, they are usually scattered about in memory (see Figure 13-2). The pointers from element to element therefore are the only means by which we can ensure that all elements remain accessible. With this mind, we will see later that special care is required when it comes to maintaining the links. If we mistakenly drop one link, it becomes impossible to access any of the elements from that point on in the list. Thus, the expression "You are only as strong as your weakest link" is particularly fitting for linked lists.

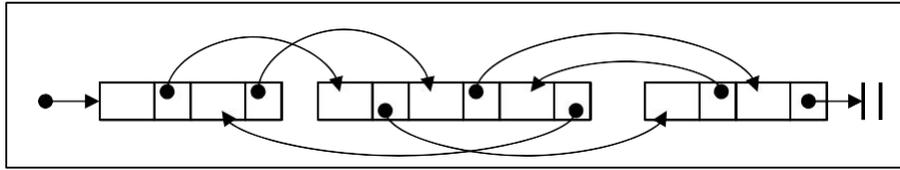


Figure 13-2. Elements of a linked list linked but scattered about an address space.

4. BASIC OPERATIONS

A list is a dynamic data structure. The number of nodes on a list may vary dramatically as elements are inserted and removed. The dynamic nature of a list may be contrasted with the static nature of an array, whose size remains constant.

The basic operations performed when using a linear link list are

- ❖ Traversing or iterating the link list
- ❖ Searching the link list
- ❖ Inserting a node into the link list
- ❖ Removing a node from the link list

13.4.1 Traversing the link list

In order to traverse the link list we need to begin at the head of the list and then move from node to node till we reach the last node which has the next pointer set to NULL. This operation is generally used to count the number of nodes in the list or/and to display the value stored in the data field of the node.

This operation can be implemented as follows

```

node = list.firstNode
count = 0
while node not null
{
    display node.data
    count++
    node = node.next
}

```

```

}

return count

```

In the above algorithm, we initialize 2 variables, node and count, node points to the first node in the list and count is the counter variable used to count the number of nodes in the list. The while loop tests for the end of the list condition. If the list is empty then the value of node will be NULL on the first iteration and the while loop will exit just before the first iteration. If the list is not empty it will display the value in the data field and increment the counter variable. At the bottom of the while loop, node = node.next advances the node pointer to the next node in the list till it reaches the value NULL.

13.4.2 Searching the link list

To search an element in a linked list, we start at the head of the list and use the next pointers of successive elements to move from element to element until the desired element is reached. A linear linked list can be traversed in only one direction – from head to tail – because each element contains no link to its predecessor. If the node we are searching is the last node in the list then we will have to traverse the entire list in order to reach that node as we cannot access any node directly. The search operation is quite similar to the traversing operation, the only change is in place of displaying or counting the nodes we check the condition of equality.

```

node = list.firstNode

info = 13

while node not null
{
    if(info == node.data)
    {
        display node
        exit
    }

    node = node.next
}

```

In the while loop, we check for the data value we are searching (in this case 13). When found we display the node and exit the loop.

13.4.3 Inserting a node into the link list

A node can be inserted either at the beginning of the link list or after a particular node. In order to insert a node in the link list we need the next node pointer of the new node to point the next node pointer of the existing node and the next node pointer of the existing node to point to the new node. The diagram below (Figure 13-3) shows how it works. Inserting a node before an existing node cannot be done; instead you have to locate it while keeping track of the previous node.

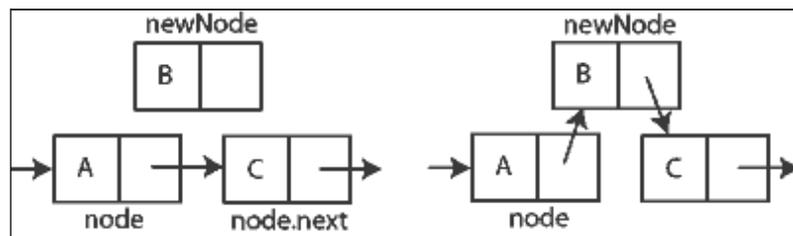


Figure 13-3: Inserting new node after an existing node

```
function insertAfter(Node node, Node newNode)
{
    // insert new node after node

    newNode.next = node.next

    node.next = newNode
}
```

A new node can also be inserted at the end of the list using the above function. However, inserting a node at the beginning of the link list requires changing the head of the current node to the new node.

```
function insertBeginning(List list, Node newNode)
```

```

{
  // insert node before current first node

  newNode.next = list.firstNode

  list.firstNode = newNode
}

```

13.4.4 Removing a node from the link list

A node can be removed from the beginning of the list or after a given node. The functions are quite similar to that of inserting a new node. The diagram below (Figure 13-4) demonstrates removing a node after a given node. To find and remove a particular node, one must again keep track of the previous node.

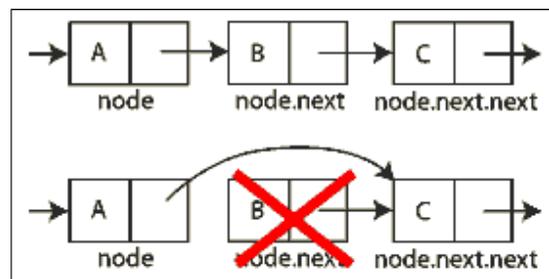


Figure 13-4: Removing an existing node.

```

function removeAfter(Node node)
{
  // remove node past this one

  obsoleteNode = node.next

  node.next = node.next.next

  destroy obsoleteNode
}

function removeBeginning(List list)
{

```

```

// remove first node

obsoleteNode = list.firstNode

list.firstNode = list.firstNode.next //point past deleted node

destroy obsoleteNode

}

```

Notice that `removeBeginning()` will set the `list.firstNode` to `null` when removing the last node in the list.

Kindly note, any of the special cases of linked list operations, like `insertBeginning()` and `removeBeginning()`, can be eliminated by including a *dummy element* at the front of the list. This ensures that there are no special cases for the beginning of the list and renders both `insertBeginning()` and `removeBeginning()` unnecessary. In this case, the first useful data in the list will be found at `list.firstNode.next`. A dummy node is one that does not contain any data. The front pointer in a linked list will point to the dummy node. It is easier to code the insertion and deletion operations if the first node on the list contains no data. Otherwise, the code to insert a new first node or to delete the current first node must be different from the code to handle the normal cases.

13.5 IMPLEMENTATION USING C

How can linear lists be represented in C? Recall that a list is simply a collection of nodes, and so it is possible to use an array of nodes. However, there are two disadvantages in using this approach. First, the number of nodes often cannot be predicted when a program is written. Usually, the data with which the program is executed determines the number of nodes necessary. Thus no matter how many elements the array of nodes contains, it is always possible that the program will be executed with the input that requires a larger number.

The second disadvantage of the array approach is that whatever number of nodes are declared must remain allocated to the program throughout its execution. For example, if 500 nodes of a given type are declared, the amount of storage required for those 500 nodes is reserved for that purpose. If the program actually uses only 100 or even 10 nodes in its execution the additional nodes are still reserved and their storage cannot be used for any other purpose.

The solution to this problem is to allow nodes that are dynamic, rather than static. That is, when a node is needed, storage is reserved for it, and when it is no longer needed, the storage is released. Thus the storage for nodes that are no longer in use is available for another purpose. Also, no predefined limit on the number of nodes is established. As long as sufficient storage space is available to the job as a whole, part of that storage can be reserved for use as a node.

13.5.1 Pointer Refresher

Here is a quick review of the terminology and rules for pointers. The linked list code to follow will depend on these rules.

- ❖ *Pointer/Pointee*: A "pointer" stores a reference to another variable sometimes known as its "pointee". Alternately, a pointer may be set to the value NULL which encodes that it does not currently refer to a pointee. (In C and C++ the value NULL can be used as a boolean false).
- ❖ *Dereference*: The dereference operation on a pointer accesses its pointee. A pointer may only be dereferenced after it has been set to refer to a specific pointee. A pointer which does not have a pointee is "bad" (below) and should not be dereferenced.
- ❖ *Bad Pointer*: A pointer which does not have an assigned pointee is "bad" and should not be dereferenced. In C and C++, a dereference on a bad sometimes crashes immediately at the dereference and sometimes randomly corrupts the memory of the running program, causing a crash or incorrect computation later. That sort of random bug is difficult to track down. **In C and C++, all pointers start out with bad values**, so it is easy to use bad pointer accidentally. Correct code sets each pointer to have a good value before using it. Accidentally using a pointer when it is bad is the most common bug in pointer code. In Java and other runtime oriented languages, pointers automatically start out with the NULL value, so dereferencing one is detected immediately. Java programs are much easier to debug for this reason.
- ❖ *Pointer assignment*: An assignment operation between two pointers like $p=q$; makes the two pointers point to the same pointee. It does not copy the pointee memory. After the assignment both pointers will point to the same pointee memory which is known as a "sharing" situation.

- ❖ *malloc()*: *malloc()* is a system function which allocates a block of memory in the "heap" and returns a pointer to the new block. The prototype for *malloc()* and other heap functions are in *stdlib.h*. The argument to *malloc()* is the integer size of the block in bytes. Unlike local ("stack") variables, heap memory is not automatically deallocated when the creating function exits. *malloc()* returns NULL if it cannot fulfill the request.
- ❖ *free()*: *free()* is the opposite of *malloc()*. Call *free()* on a block of heap memory to indicate to the system that you are done with it. The argument to *free()* is a pointer to a block of memory in the heap — a pointer which some time earlier was obtained via a call to *malloc()*.

13.5.2 Allocating and Freeing Dynamic Variables

In C a pointer variable to an integer can be created by the declaration

```
int *p;
```

Once a variable *p* has been declared as a pointer to a specific object, it must be possible to dynamically create an object of that specific type and assign its address to *p*.

This may be done in C by calling the standard library function *malloc(size)*. *malloc* dynamically allocates a portion of memory of size *size* and returns a pointer to an item of type *char*.

The statements

```
pi = (int *) malloc(sizeof (int));
pr = (float *) malloc(sizeof (float));
```

dynamically create the integer variable **pi* and the float variable **pr*. These variables are called **dynamic variables**. In executing these statements, the operator **sizeof** returns the size, in bytes, of its operand. *malloc* can then create an object of that size. Thus *malloc(sizeof(int))* allocates storage for an integer, whereas *malloc(sizeof(float))* allocates storage for a floating-point number. *malloc* also returns a pointer to the storage it allocates. This pointer is to the first byte of that storage and is of the type **char ***. To coerce this pointer so that it points to an integer or real, we use the cast operator **(int *)** or **(float *)**.

The function `free` is used in C to free storage of a dynamically allocated variable. The statement

```
free(p);
```

makes any future references to the variable `*p` illegal (unless, of course, a new value is assigned to `p` by an assignment statement or by a call to `malloc`). Calling `free(p)` makes the storage occupied by `*p` available for reuse, if necessary.

13.5.3 Creating a node

Each node in the link list has two fields: a data pointer and a pointer to the next node in the list. The data pointer is of type `void *`, so that it could point to any data. A node with `next == NULL` is the last node on the list. The structure for a single node is

```
typedef struct node
{
    void *data;
    struct node *next;
}LLIST;
```

Creating a new node is simple. The memory needed to store the node is allocated, and the pointers are set up.

```
LLIST *list_create(void *data)
{
    LLIST *node;
    if(!(node=malloc(sizeof(LLIST)))) return NULL;
    node->data=data;
    node->next=NULL;
```

```

        return node;
    }

```

13.5.4 Inserting a node

In a singly-linked list, there is no efficient way to insert a node before a given node or at the end of the list, but we can insert a node after a given node or at the beginning of the list. The following code creates and inserts a new node after an existing node.

```

LLIST *list_ins_aft(LLIST *node, void *data)
{
    LLIST *newnode;
    newnode=list_create(data);
    newnode->next = node->next;
    node->next = newnode;
    return newnode;
}

```

The above code cannot insert at the beginning of the list, so we have a separate function for this. This code creates and inserts the new node and returns the new head of the list:

```

LLIST *list_ins_begin(LLIST *list, void *data)
{
    LLIST *newnode;
    newnode=list_create(data);
    newnode->next = list;
    return newnode;
}

```

13.5.5 Removing a node

Removing a node before an existing one cannot be done; instead, you have to locate it while keeping track of the previous node. The following function enables to remove the node *after* a given node, and remove a node from the beginning of the list. To find and remove a particular node, one must again keep track of the previous element.

```
int list_remove(LLIST *list, LLIST *node)
{
    while(list->next && list->next!=node)
        list=list->next;

    if(list->next)
    {
        list->next=node->next;
        free(node);
        return 0;
    }
    else return -1;
}
```

13.5.6 Traversing a list

The following function will print the data field of the node while traversing the link list.

```
int list_traverse(LLIST *node)
{
    while(node)
    {
        printf("%s\n", (char *)node->data);
        node=node->next;
    }
}
```

```

    }

    return 0;
}

```

13.5.7 Searching a list

Searching a node containing particular data requires the data in the list and the data to be searched for to be of similar datatype. In some cases it may not be possible as the user may need to search for number (int datatype) but the programmer has stored all the data in the list in char datatype. To overcome these problems we add a function that takes the data in the list and data to be searched in a function and then compares them for equality.

```

int findstring(void *listdata, void *searchdata)
{
    return strcmp((char*)listdata, (char*)searchdata)?0:1;
}

```

The above function tests to see if the value of a given node matches some string.

The searching function will return the first node to which the supplied function pointer returns a positive number.

```

LLIST *list_find(LLIST *node, int(*func)(void*,void*), void *data)
{
    while(node)
    {
        if(func(node->data, data)>0) return node;
        node=node->next;
    }
    return NULL;
}

```

```
}

```

13.5.8 Implementation of Linear Linked List

Example 13-1. Header for the Linear Link List Abstract Datatype

```
/*
 *   File: llist.h
 *   Topic: Linear Link List Implementation
 *   -----
 *
 */

#ifndef LLIST_H
#define LLIST_H

/*
 * Type: LLIST
 * -----
 * This is the type for a linear link list, i.e., it is a type that
 * holds the information necessary to keep track of the nodes.
 * It has a 'data' pointer so that it can point to any data and a
 * 'next' pointer that points to the next node in the list.
 */

typedef struct node
{
    void *data;
    struct node *next;

```

```
}LLIST;
```

```
/*
```

```
* Function: list_create
```

```
* Usage: list_create(&data);
```

```
* -----
```

```
* A new link list is created and the data field of the head node is
```

```
* given.
```

```
*/
```

```
LLIST *list_create(void *data)
```

```
/* Functions: list_ins_aft, list_ins_begin
```

```
* Usage: list_ins_aft(&node, &data), list_ins_begin(&list, &data);
```

```
* -----
```

```
* The first function inserts a node after a particular node.
```

```
* The second function inserts a node at the beginning of the list..
```

```
*/
```

```
LLIST *list_ins_aft(LLIST *node, void *data)
```

```
LLIST *list_ins_begin(LLIST *list, void *data)
```

```
/* Function: list_remove
```

```
* Usage: list_remove(&list, &node)
```

```
* -----
```

```

* This function removes a node after a particular node.
* also this function removes a node at the beginning of the list..
*/

```

```
int list_remove(LLIST *list, LLIST *node)
```

```

/*
* Functions: list_traverse, list_find
* Usage: list_traverse(&node); list_find(&node, int func, &data);
* -----
* These functions operate on an entire list
* list_traverse function is used to traverse the link list and print
* the data field of each node.
* list_find function is used to search for the data of a particular
* node and uses a simple function to compare the data.
*/

```

```
int list_traverse(LLIST *node)
```

```
LLIST *list_find(LLIST *node, int(*func)(void*,void*), void *data)
```

```
#endif /* not defined LLIST_H */
```

Example 14-2. Implementation of the Linear Link List Abstract Datatype.

```

/*
* File: llist.c
* Topic: Linear Link List
* -----

```

```
*  
*/  
  
#include <stdio.h>  
  
#include <stdlib.h> /* for dynamic allocation */  
  
#include "l1ist.h"  
  
/***** Function Definitions *****/  
  
LLIST *list_create(void *data)  
{  
    LLIST *node;  
    if(!(node=malloc(sizeof(LLIST)))) return NULL;  
    node->data=data;  
    node->next=NULL;  
    return node;  
}  
  
LLIST *list_ins_aft(LLIST *node, void *data)  
{  
    LLIST *newnode;  
    newnode=list_create(data);  
    newnode->next = node->next;  
    node->next = newnode;  
    return newnode;  
}
```

```
LLIST *list_ins_begin(LLIST *list, void *data)
{
    LLIST *newnode;

    newnode=list_create(data);

    newnode->next = list;

    return newnode;
}

int list_remove(LLIST *list, LLIST *node)
{
    while(list->next && list->next!=node)
        list=list->next;

    if(list->next)
    {
        list->next=node->next;

        free(node);

        return 0;
    }

    else return -1;
}

int list_traverse(LLIST *node)
{
    while(node)
    {
        printf("%s\n", (char *)node->data);

        node=node->next;
    }
}
```

```
    }  
    return 0;  
}  
  
int findstring(void *listdata, void *searchdata)  
{  
    return strcmp((char*)listdata, (char*)searchdata)?0:1;  
}  
  
LLIST *list_find(LLIST *node, int(*func)(void*,void*), void *data)  
{  
    while(node)  
    {  
        if(func(node->data, data)>0) return node;  
        node=node->next;  
    }  
    return NULL;  
}
```

13.6 SUMMARY

- Linked lists consist of a number of elements grouped, or *linked*, together in a specific order. Each element is called a node.
- Each node contains two fields: a "data" field to store information, and a "next" field which is a pointer to the next node.
- The next pointer of the last element is set to NULL, to mark the end of the list.
- The element at the start of the list is its *head*; the element at the end of the list is its *tail*.
- Each node is dynamically allocated using `malloc()`, so the node memory continues to exist until it is explicitly deallocated with a call to `free()`.
- A linear linked list can be traversed in only one direction – from head to tail – because each element contains no link to its predecessor.
- To traverse a linked list, you start at head and then go from link to link, using each link's next field to find the next link.
- To search an element in a linked list, we start at the head of the list and use the next pointers of successive elements to move from element to element until the desired element is reached.
- A node can be inserted either at the beginning of the link list or after a particular node.
- Inserting an item at the beginning of a linked list involves changing the new link's next field to point to the old first link and changing first to point to the new item.
- A node can be removed from the beginning of the list or after a given node.
- Deleting an item at the beginning of a list involves setting first to point to `first.next`.

13.7 UNIT END EXERCISES

13.7.1 Questions

These questions are intended as a self-test for readers.

1. State the features and uses of linear linked list?
2. Mention the advantages and shortcomings of linear link list with respect to an array?
3. Explain the basic operations of link list?
4. Explain the terms `malloc()` and `free()`?

13.7.2 Programming Projects

Writing programs that solve the Programming Projects helps to solidify your understanding of the material and demonstrates how the chapter's concepts are applied.

1. Write a program `testlinkedlist.c` that tests the above given program `l1.c`.
2. Write separate C routines for removing a node from the link list, removing the head node from the link list and destroying the entire link list.
3. Write a menu driven program that allows the user to create a list, insert a node, delete a node, count the number of nodes, display the list. The program should display the menu as follows:

PROGRAM TO IMPLEMENT SINGLY LINKED LIST

=====

1. Create
2. Insert Node at Beginning
3. Insert Node in Middle
4. Deleting a Node
5. Counting the Number of Nodes
6. Displaying the list
7. Exit

Once the list has been created and if the user selects the menu of creating the list again, it should mention that a list has been created and append the list.

[Hint: Use switch case]

4. Write a C function `search(l, x)` that accepts a pointer `l` to a list of integers and an integer `x` and returns a pointer to a node containing `x`, if it exists, and the null pointer otherwise. Write another function, `srchins(l, x)`, that adds `x` to `l` if it is not found and always returns a pointer to a node containing `x`.



. **STACK****Unit Structure**

1. Objectives
2. Introduction
- 14.3 Description of Stacks
 1. Abstraction
 2. Order produced by Stack
4. Array Representation of Stacks
5. Understanding Primitive Operations
 1. Implementing the Pop operation
 2. Implementing the Push operation
6. Implementation using C
 1. Implementation and Analysis of Stacks
7. Summary
8. Unit End Exercises

14.1 OBJECTIVES

After going through this chapter you will be able to

- Define a stack and its features.
- Write Algorithms for the basic operations of Stack.
- Understand the difference between Stack & Array.
- Understand how an Array is used to implement a Stack.
- Write a program in C to implement Stack.

14.2 INTRODUCTION

Often it is important to store data so that when it is retrieved later, it is automatically presented in some prescribed order. A common way to retrieve data is in the opposite order as it was stored. For example, consider the data blocks a program maintains to keep track of function calls as it runs. These blocks are called *activation records*. For a set of functions $\{f_1, f_2, f_3\}$ in which f_1 calls f_2 and f_2 calls f_3 , a program allocates one activation record each time one of the functions is called. Each record persists until its function returns. Since functions return in the opposite order as they were called, activation records are retrieved and relinquished in the opposite order as they were called. Stacks are simple data structures that help in such common situations.

Some applications of Stacks are:

- ✓ *Function calls in C*: An essential part of modular programming. When we call a function in a C program, an activation record containing information about the call is pushed onto a stack called the *program stack*. When a function terminates, its activation record is popped off the stack. A stack is the perfect model for this because when functions call one another, they return in the opposite order as they were called.

- ✓ *Abstract stack machines*: An abstraction used by compilers and hand-held calculators to evaluate expressions.

14.3 DESCRIPTION OF STACKS

A *stack* is an ordered collection of items into which items can be inserted or deleted from one end. The end into which items can be inserted or from which existing items can be deleted is called the *TOP* of the stack.

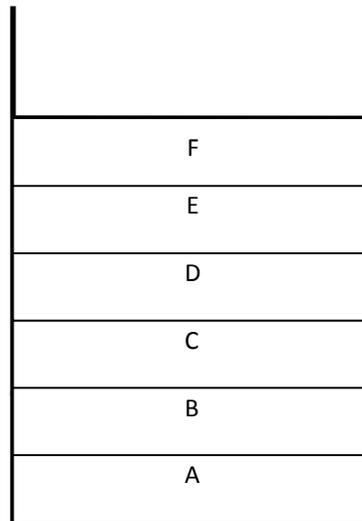


Figure 14.1: A Stack containing stack terms.

A convenient way to think of a stack is a can of tennis balls. As we place balls in the can, the can is filled up from the bottom to the top. When we remove the balls, the can is emptied from the top to the bottom. Furthermore, if we want a ball from the bottom of the can, we must remove each of the balls above it. In computing, to place an element on top of a stack, we *push* it; to remove an element from the top, we *pop* it.

The distinguishing characteristic of a stack is that it stores and retrieves data in a *last-in, first-out*, or *LIFO*, manner. This means that the last element placed on the stack is the first to be removed.

14.3.1 Abstraction

Now, let's think about a stack in an abstract way i.e., it doesn't hold any particular kind of thing (like tennis balls) and we aren't restricting ourselves to any particular programming language or any particular implementation of a stack.

Stacks hold objects, usually all of the same type. Most stacks support just a simple set of operations; and thus, the main property of a stack is that objects go on and come off of the *top* of the stack.

Here are the minimal operations we'd need for an abstract stack (and their typical names):

Push: Places an object on the *top* of the stack.

Pop: Removes an object from the *top* of the stack and produces that object.

IsEmpty: Reports whether the stack is empty or not.

Because we think of stacks in terms of the physical analogy, we usually draw them vertically (so the **top** is really *on top*).

14.3.2 Order produced by a Stack

Stacks are linear data structures. This means that their contexts are stored in what looks like a line (although vertically). This linear property, however, is not sufficient to discriminate a stack from other linear data structures. For example, an array is a sort of linear data structure. However, you can access any element in an array, which is not true for a stack, since you can only deal with the element at its top.

One of the distinguishing characteristics of a stack, and the thing that makes it useful, is *the order* in which elements come out of a stack.

Suppose we have a stack that can hold letters, call it *stack*.

We begin with *stack* empty:

stack

Now, let's perform `Push(stack, A)`, giving:

| A | <-- top

stack

Again, another push operation, `Push(stack, B)`, giving:

| B | <-- top

| A |

stack

Again, another push operation, `Push(stack, C)`, giving:

| C | <-- top

| B |

| A |

stack

Now let's remove an item, `letter = Pop(stack)`, giving:

```

-----
| B | <-- top | C |
-----
| A |          letter
-----
Stack

```

Now let's remove another item, `letter = Pop(stack)`, giving:

```

-----
| A | <-- top | B |
-----
stack        letter

```

And finally, one more addition, `Push(stack, D)`, giving:

```

-----
| D | <-- top
-----
| A |
-----
stack

```

You'll notice that the stack enforces a certain *order* to the use of its contents, i.e., *Last In First Out*. Thus, we say that a stack enforces **LIFO** order.

Thank You



Introduction to algorithm&

C

Part-6



14.4 Array Representation of Stacks

There are several ways to represent a stack in C. The simplest representation is representing a stack using an *array*. Another way of representing is using a *Linked List*.

A stack is an ordered collection of items, and C already contains a data type that is an ordered collection of items: the array. Whenever a problem solution calls for the use of stack, therefore it is tempting to begin a program by declaring a variable `stack` as an array. However, a stack and an array are two entirely different things. The number of elements in an array is fixed and is assigned by the declaration for the array. In general, the user cannot change this number. A stack, on the other hand, is fundamentally a dynamic object whose size is constantly changing as items are popped and pushed.

However, although an array cannot be a stack, it can be the home of a stack. That is, an array can be declared large enough for the maximum size of the stack. The stack can grow and shrink during the course of the program, within the space reserved for it. One end of the array is the fixed bottom of the stack, while the top of stack constantly shifts as items are popped and pushed. Thus, another field is needed that, at each point during program execution keeps track of the current position of the top of the stack.

Consider an **example**:

Suppose the stack has (A, B) in it already...

stack (made up of 'contents' and 'top')

```

-----
| A | B | | | | 1 |
-----
0 1 2 3 top
contents

```

Since **B** is at the top of the stack, the value *top* stores the index of **B** in the array (i.e., 1).

Now, suppose we push something on the stack, `Push(stack, 'C')`, giving:

stack (made up of 'contents' and 'top')

```

-----          ----
| A | B | C | |   | 2 |
-----          ----
0 1 2 3   top
      contents

```

(Note that both the *contents* and *top* part have to change.)

So, a sequence of pops produce the following effects:

```
letter = Pop(stack)
```

stack (made up of 'contents' and 'top')

```

-----          ----  ----
| A | B | | |   | 1 | | C |
-----          ----
0 1 2 3   top letter
      contents

```

```
tter = Pop(stack)
```

stack (made up of 'contents' and 'top')

```

-----          ----  ----
| A | | | |   | 0 | | B |
-----          ----
0 1 2 3   top letter
      contents

```

```
letter = Pop(stack)
```

stack (made up of 'contents' and 'top')

```

-----          ----  ----
| | | | |           | -1 | | A |
-----          ----  ----

0 1 2 3   top  letter
contents

```

so that you can see what value *top* should have when it is empty, i.e., -1.

14.5 UNDERSTANDING PRIMITIVE OPERATIONS

For an array implementation of stack, we need to keep track of (at least) the array *contents* and a *top* index.

Also we need to determine, during the course of execution, whether or not a stack is empty. Although the push operation is applicable to any stack, the pop operation cannot be applied to the empty stack because such a stack has no elements to pop. Therefore before applying the pop operator to a stack, we must ensure that the stack is not empty. The *operation empty(s)* determines whether or not a stack *s* is empty. If the stack is empty, *empty(s)* returns the value TRUE; otherwise it returns the value FALSE.

```

if (empty(s))
    return TRUE
else
    i = pop(s)
    print (i)
endif

```

We can write a function to implement the operation `empty(s)` that returns `TRUE` if the stack is empty and `FALSE` if it is not empty, as follows

```

empty(ps)

struct stack *ps;
{
    if(ps -> top == -1)
        return(TRUE);
    else
        return(FALSE);
}    /* end empty */

```

Once this function exists, a test for the empty stack is implemented by the statement

```

if (empty(&s))
    /* stack is empty */
else
    /* stack is not empty */

```

Note the difference between the syntax of the call to `empty` in the algorithm and in the program segment above. In the algorithm, `s` represents a stack and the call to `empty` was expressed as `empty(s)`.

In the program we are concerned with the actual implementation of the stack and its operations in C. Since parameters in C are passed by value, the only way to modify the argument passed to a function is to pass the address of the argument rather than the argument itself, .i.e. passing by reference. Further, the original definition of C (by Kernighan-Ritchie) and many older C compilers do not allow a structure to be passed as an argument even if its value remains unchanged. Thus, in functions such as `pop` and `push` (which modify their structure arguments), as well as `empty` (which does not), we adopt the

convention that we pass the address of the stack structure rather than the stack itself. (This restriction has been omitted from many newer compilers.)

14.5.1 Implementing the Pop operation

The possibility of underflow must be considered in implementing the *pop* operation, as the user may attempt to pop an element from an empty stack. If such an attempt is made the user should be informed of the underflow condition. We therefore introduce a function *pop* that performs the following three actions:

1. If the stack is empty, print a warning message and halt execution.
2. Remove the top element from the stack.
3. Decrement the top value.
4. Return this element to the calling program.

This operation begins by checking whether or not the stack is empty. If the stack is empty, a warning message stating that the stack is empty is printed and the execution is stopped. If the stack is not empty, the top element on the stack is removed and returned to the calling program. We also need to decrement the value of the top of the stack by 1 as we have removed the top element and returned it to the calling program.

The above conditions can be checked using an if statement

```
BEGIN
    if (top == -1)
        print(stack underflow)
    else
        print(popped element is stack arr[top])
        top=top-1
    end if
END
```

Here we check whether the array is empty before attempting to pop an element from the stack. The array is empty if $top = -1$.

Here we must understand that underflow indicates that the pop operation cannot be performed on the stack and may indicate an error in the algorithm or the data. No other implementation or representation of the stack will cure the underflow condition. Rather, the entire problem must be rethought.

We assume that the stack consists of integers, so that the pop operation can be implemented as a function. However, if a stack consists of a more complex structure (for example, a structure or a union), the pop operation would either be implemented as returning a pointer to a data element of the proper type (rather than the data element itself), or the operation would be implemented with the popped value as a parameter (in which case the address of the parameter would be passed rather than the parameter, so that the pop function could modify the actual argument).

```
pop(ps)
struct stack *ps;
{
    if (empty(ps))
    {
        print("Stack Underflow");
        exit(1);
    } /* end if */
    return(ps->items[ps->top--]);
} /* end pop */
```

Note that `ps` is already a pointer to a structure of type `stack`; therefore, the address operator `&` is not used in calling `empty`.

Let us look at the pop function more closely. If the stack is not empty, the top element of the stack is retained as the return value. This element is then removed from the stack by the expression $ps \rightarrow top--$.

14.5.2 Implementing the Push operation

Let us now examine the push operation. Adding an element to the stack should be quite easy using the array representation of stack. We introduce a function *push* that performs the following actions:

1. Increment the top value
2. Add the new element to top of the stack.

This function makes room for the new element to be pushed onto the stack by incrementing the top by 1 and then inserts the new element into the array.

Although this function implements the push operation, yet it is incorrect. It allows a subtle error to creep in, caused by using the array representation of the stack. Recall that a stack is a dynamic structure that is constantly allowed to grow and shrink and thus change its size. An array, on the other hand, is a fixed object of predetermined size. Thus, it is quite conceivable that a stack may outgrow the array that was set aside to contain it. This occurs when the array is full, .i.e., when the stack contains as many elements as the array and an attempt is made to push yet another element onto the stack. The result of such an attempt is called an *overflow*. In such a situation an error message needs to be produced that relates to the cause of the error.

The push procedure must therefore be revised so that it reads as follows:

1. If stack is full, print warning message and halt execution.
2. If stack is not full, increment the top value.
3. Add new element to the top of the stack.

The above conditions can be checked using if statement.

Begin

```
if (top == max - 1)
    print (stack overflow)
    exit
else
    top = top + 1
    stack arr[top] = pushed item
endif
```

End

Here we check whether the array is full before attempting to push another element onto the stack. The array is full if $top == max - 1$.

Here, we need to understand that *overflow* is not a condition that is applicable to a stack as an abstract data structure. Abstractly, it is always possible to push an element onto a stack. A stack is an ordered set, and there is no limit to the number of elements that such a set can contain. The possibility of overflow is introduced when a stack is implemented by an array with only a finite number of elements, thereby prohibiting the growth of the stack beyond that number. It may very well be that the algorithm that the programmer used is correct, just that the implementation of the algorithm did not anticipate that the stack would become so large. Thus, in some cases an overflow condition can be corrected by changing the value of the constant array size so that the array field item contains more elements.

Based on the above algorithm, the push procedure when implemented reads as follows:

```
push(ps, x)
struct stack *ps;
int x;
```

```

{
    if (ps->top == STACKSIZE-1)
    {
        printf("Stack Overflow");
        exit(1);
    }
    else
        ps->items[++(ps->top)] = x;
    return;
} /* end push */

```

Here we check whether the array is full before attempting to push another element onto the stack. The array is full if $ps \rightarrow top == stacksize - 1$.

You should again note that if and when overflow is detected in push, execution halts immediately after an error message is printed.

14.6 IMPLEMENTATION USING C

For an array implementation of stack, we need to keep track of (at least) the array *contents* and a *top* index.

A stack in C may therefore be declared as a structure containing two objects: an array to hold elements of the stack, and an integer to indicate the position of the current stack top within the array. This may be done for a stack of integers by the declarations

```

#define MAXSIZE 100

struct stack
{
    int top;

```

```
int items[MAXSIZE];  
} stackT;
```

Here we assume that the elements of the stack `stackT` contained in the array `stackT.items` are integers and that the stack will at no time contain more than 100 integers.

There is, of course, no reason to restrict a stack to contain only integers; a stack can as easily contain float elements or char elements, or whatever other type we might wish to give to the elements of the stack. In fact, should the need arise, a stack can contain objects of different types by using C unions.

In order to implement a stack in C, let's think the functions we need...

First, we need the standard stack operations:

`Stack_Push()`

`Stack_Pop()`

`Stack_IsEmpty()`

We'll use the convention of placing the data structure name at the beginning of the function name (e.g., `Stack_IsEmpty`). That will help us down the line. For example, suppose we use 2 different data structures in a program, both with *IsEmpty* operations--our naming convention will prevent the 2 different `IsEmpty` functions from conflicting.

We'll also need 2 extra operations:

`Stack_Init()`

`Stack_Destroy()`

They are not part of the *abstract* concept of a stack, but they are necessary for setup and cleanup when writing the stack in C.

Finally, while the array that holds the contents of the stack will be dynamically-allocated, it still has a maximum size. So, this stack is unlike the abstract stack in that it can get full. We should add something to be able to test for this state:

Stack_IsFull()

Thus, we need the following functions:

- Stack_Push()
- Stack_Pop()
- Stack_IsEmpty()
- Stack_Init()
- Stack_Destroy()
- Stack_IsFull()

14.6.1 Implementation and Analysis of Stacks

Example 14-1. Header for the Stack Abstract Datatype

```

/*
 *   File: stack.h
 *
 *   Topic: Stack - Array Implementation
 *
 *   -----
 *
 *
 * This is the interface for a stack of characters.
 */

#ifndef _STACK_H
#define _STACK_H

/*
 * Type: stackElementT

```

```
* -----
```

```
* This is the type of the objects entered in the stack.
```

```
* Edit it to change the type of things to be placed in
```

```
* the stack.
```

```
*/
```

```
typedef char stackElementT;
```

```
/*
```

```
* Type: stackT
```

```
* -----
```

```
* This is the type for a stack, i.e., it is a type that
```

```
* holds the information necessary to keep track of a stack.
```

```
* It has a pointer `contents' to a dynamically-allocated
```

```
* array (used to hold the contents of the stack), an integer
```

```
* `maxSize' that holds the size of this array (i.e., the
```

```
* maximum number of things that can be held in the stack),
```

```
* and another integer `top,' which stores the array index of
```

```
* the element at the top of the stack.
```

```
*/
```

```
typedef struct
```

```
{
```

```
    stackElementT *contents;
```

```
    int maxSize;
```

```
    int top;
```

```
} stackT;
```

```
/*  
 * Function: Stack_Init  
 * Usage: Stack_Init(&stack, maxSize);  
 * -----  
 * A new stack variable is initialized. The initialized  
 * stack is made empty. MaxSize is used to determine the  
 * maximum number of character that can be held in the  
 * stack.  
 */  
  
void Stack_Init(stackT *stackP, int maxSize);  
  
/* Function: Stack_Destroy  
 * Usage: Stack_Destroy(&stack);  
 * -----  
 * This function frees all memory associated with the stack.  
 * The `stack' variable may not be used again unless  
 * Stack_Init(&stack, maxSize) is first called on the stack.  
 */  
  
void Stack_Destroy(stackT *stackP);  
  
/*  
 * Functions: Stack_Push, Stack_Pop  
 * Usage: Stack_Push(&stack, element); element = Stack_Pop(&stack);  
 * -----  
 * These are the fundamental stack operations that add an element to
```

```

* the top of the stack and remove an element from the top of the
* stack.
* A call to Stack_Pop on an empty stack or to StackPush on a full
* stack is an error. Make use of Stack_IsEmpty()/Stack_IsFull()
* (see below) to avoid these errors.
*/

```

```
void Stack_Push(stackT *stackP, stackElementT element);
```

```
stackElementT Stack_Pop(stackT *stackP);
```

```

/*
* Functions: Stack_IsEmpty, Stack_IsFull
* Usage: if (Stack_IsEmpty(&stack)) ...
* -----
* These return a true value if the stack is empty
* or full (respectively).
*/

```

```
int Stack_IsEmpty(stackT *stackP);
```

```
int Stack_IsFull(stackT *stackP);
```

```
#endif /* not defined _STACK_H */
```

➤ **Stack_Init():**

The first function we'll implement is `Stack_Init()`. It will need to set up a `stackT` structure so that it represents an *empty stack*.

Here is what the prototype for `Stack_Init()` looks like...

```
void Stack_Init(stackT *stackP, int maxSize);
```

It needs to change the stack passed to it, so the stack is passed by reference (`stackT *`). It also needs to know what the maximum size of the stack will be (i.e., `maxSize`).

Now, the body of the function must:

1. Allocate space for the contents.
2. Store the maximum size (for checking fullness).
3. Set up the top.

Here is the full function:

```
void Stack_Init(stackT *stackP, int maxSize)
{
    stackElementT *newContents;

    /* Allocate a new array to hold the contents. */

    newContents = (stackElementT *)malloc(sizeof(stackElementT)*
maxSize);

    if (newContents == NULL)
    {
        fprintf(stderr, "Insufficient memory to initialize stack.\n");
        exit(1); /* Exit, returning error code. */
    }

    stackP->contents = newContents;
```

```

stackP->maxSize = maxSize;

stackP->top = -1; /* i.e., empty */

}

```

Note how we make sure that space was allocated (by testing the pointer against `NULL`). Also, note that if the stack was not passed by reference, we could not have changed its fields.

➤ **Stack_Destroy():**

The next function we'll consider is the one that cleans up a stack when we are done with it. It should get rid of any dynamically-allocated memory and set the stack to some *reasonable state*.

This function only needs the stack to operate on:

```
void StackDestroy(stackT *stackP);
```

and should reset all the fields set by the initialize function:

```

void StackDestroy(stackT *stackP)
{
    /* Get rid of array. */
    free(stackP->contents);

    stackP->contents = NULL;

    stackP->maxSize = 0;

    stackP->top = -1; /* i.e., empty */
}

```

➤ **Stack_IsEmpty() and Stack_IsFull():**

Let's look at the functions that determine emptiness and fullness. Now, it's not necessary to pass a stack by reference to these functions, since they do not change the stack. So, we could prototype them as:

```
int Stack_IsEmpty(stackT stack);
```

```
int Stack_IsFull(stackT stack);
```

However, then some of the stack functions would take pointers (e.g., we need them for `StackInit()`, etc.) and some would not. It is more *consistent* to just pass stacks by reference (with a pointer) all the time. Furthermore, if the struct `stackT` is large, passing a pointer is more efficient (since it won't have to copy a big struct).

So, our prototypes will be:

```
int Stack_IsEmpty(stackT *stackP);
```

```
int Stack_IsFull(stackT *stackP);
```

Emptiness

Now, testing for emptiness is an easy operation. We've said that the *top* field is -1 when the stack is empty. Here's a simple implementation of the function.

```
int Stack_IsEmpty(stackT *stackP)
{
    return stackP->top < 0;
}
```

Fullness

Testing for fullness is only slightly more complicated. Let's look at an example stack.

Suppose we asked for a stack with a maximum size of 1 and it currently contained 1 element (i.e., it was full)...

stack (made up of 'contents' and 'top')

```

-----
| A |   | 0 |
-----
0      top

```

contents

We can see from this example that when the *top* is equal to the *maximum size minus 1* (e.g., $0 = 1 - 1$), then it is full. Thus, our fullness function is.

```

int Stack_IsFull(stackT *stackP)
{
    return stackP->top >= stackP->maxSize - 1;
}

```

This illustrates the importance of keeping the maximum size around in a field like `maxSize`.

➤ **Stack_Push():**

Now, pushing onto the stack requires the stack itself as well as *something to push*. So, its prototype will look like:

```
void Stack_Push(stackT *stackP, stackElementT element);
```

The function should place an element at the correct position in the *contents* array and update the *top*. However, before the element is placed in the array, we should make sure the array is not already full...Here is the body of the function:

```

void Stack_Push(stackT *stackP, stackElementT element)
{
    if (Stack_IsFull(stackP))
    {
        fprintf(stderr, "Can't push element on stack: stack is full.\n");
        exit(1); /* Exit, returning error code. */
    }

    /* Put information in array; update top. */

    stackP->contents[++stackP->top] = element;
}

```

Note how we used the *prefix* ++ operator. It increments the *top* index before it is used as an index in the array (i.e., where to place the new element).

Also note how we just reuse the `StackIsFull()` function to test for fullness.

➤ **Stack_Pop():**

Finally, popping from a stack only requires a stack parameter, but the value popped is typically returned. So, its prototype will look like:

```
stackElementT Stack_Pop(stackT *stackP);
```

The function should return the element at the top and update the *top*. Again, before an element is removed, we should make sure the array is not empty. Here is the body of the function:

```

stackElementT Stack_Pop(stackT *stackP)
{
    if (Stack_IsEmpty(stackP))
    {
        fprintf(stderr, "Can't pop element from stack: stack is
empty.\n");
        exit(1); /* Exit, returning error code. */
    }

    return stackP->contents[stackP->top--];
}

```

Note how we had the sticky problem that we had to update the *top* before the function returns, but we need the *current value of top* to return the correct array element. This is accomplished easily using the *postfix --* operator, which allows us to use the current value of *top* before it is decremented.

Example 14-2. Implementation of the Stack Abstract Datatype.

```

/*
 *   File: stack.c
 *
 *   Topic: Stack - Array Implementation
 *
 * -----
 *
 * This is an array implementation of a character stack.
 */

#include <stdio.h>

#include <stdlib.h> /* for dynamic allocation */

#include "stack.h"

```

```
/****** Function Definitions *****/
```

```
void Stack_Init(stackT *stackP, int maxSize)
```

```
{
```

```
    stackElementT *newContents;
```

```
    /* Allocate a new array to hold the contents. */
```

```
    newContents = (stackElementT *)malloc(sizeof(stackElementT) * maxSize);
```

```
    if (newContents == NULL)
```

```
    {
```

```
        fprintf(stderr, "Insufficient memory to initialize stack.\n");
```

```
        exit(1); /* Exit, returning error code. */
```

```
    }
```

```
    stackP->contents = newContents;
```

```
    stackP->maxSize = maxSize;
```

```
    stackP->top = -1; /* i.e., empty */
```

```
}
```

```
void Stack_Destroy(stackT *stackP)
```

```
{
```

```
    /* Get rid of array. */
```

```
    free(stackP->contents);
```

```
stackP->contents = NULL;

stackP->maxSize = 0;

stackP->top = -1; /* i.e., empty */
}

void Stack_Push(stackT *stackP, stackElementT element)
{
    if (Stack_IsFull(stackP))
    {
        fprintf(stderr, "Can't push element on stack: stack is full.\n");
        exit(1); /* Exit, returning error code. */
    }

    /* Put information in array; update top. */

    stackP->contents[++stackP->top] = element;
}

stackElementT Stack_Pop(stackT *stackP)
{
    if (Stack_IsEmpty(stackP))
    {
        fprintf(stderr, "Can't pop element from stack: stack is empty.\n");
        exit(1); /* Exit, returning error code. */
    }

    return stackP->contents[stackP->top--];
}
```

```
}

int Stack_IsEmpty(stackT *stackP)
{
    return stackP->top < 0;
}

int Stack_IsFull(stackT *stackP)
{
    return stackP->top >= stackP->maxSize - 1;
}
```

14.7 SUMMARY

- Stacks are data structures usually used to simplify certain programming operations.
- In this data structure, only one data item can be accessed.
- A *stack* is an ordered collection of items into which items can be inserted or deleted from one end.
- A stack allows access to the last item inserted.
- The important stack operations are pushing (inserting) an item onto the top of the stack and popping (removing) the item that's on the top.
- A stack is said to be a Last-In-First-Out (LIFO) storage mechanism because the last item inserted is the first one to be removed.

- An array is a sort of linear data structure containing a fixed number of elements which is assigned at time of declaration. A stack, on the other hand, is fundamentally a dynamic object whose size is constantly changing as items are popped and pushed.
 - You can access any element in an array, which is not true for a stack, since you can only deal with the element at its top.
 - There are several ways to represent a stack in C. The simplest representation is representing a stack using an array. Another way of representing is using a Linked List.
-

14.8 UNIT END EXERCISES

1. Questions

These questions are intended as a self-test for readers.

1. Suppose you push 10, 20, 30, and 40 onto the stack. Then you pop three items. Which one is left on the stack?
2. What does LIFO mean?
3. State the differences between a Stack and an Array?
4. Define a Stack and state its basic operations?

2. Programming Projects

Writing programs that solve the Programming Projects helps to solidify your understanding of the material and demonstrates how the chapter's concepts are applied.

1. Write a program to implement a stack of integers in C.
2. Given a sequence of push and pop operations and an integer representing the size of an array in which a stack is to be implemented, design an algorithm for determining whether or not overflow and underflow occurs. Implement the algorithm as a C program.

3. Write a program to implement a stack containing elements of different data types .i.e. int, float, char, string.

(Hint: Use unions and use switch case when popping and pushing elements)

4. Write a program to implement a stack using linked list.



QUEUES

Unit Structure

1. Objectives
2. Introduction
- 15.3 Description of Queues
- 15.4 Abstraction
- 15.5 Understanding Primitive Operations
 1. Checking for Emptiness
 2. Inserting an element
 3. Removing an element
- 15.6 Implementation using C
 - 15.6.1 Array Implementation of Queues
 - 15.6.2 Linked List Implementation of Queues
- 15.7 Summary
- 15.8 Unit End Exercises

15.1 OBJECTIVES

After going through this chapter you will be able to

- Define a queue and state its features.
- State the applications that use queues.
- State the basic operations of a queue.
- Differentiate between straight queue and circular queue.
- Implement queues using arrays and linked lists.

15.2 INTRODUCTION

Often it is important to store data so that when it is retrieved later, it is automatically presented in some prescribed order. A common way to retrieve data is in the opposite order as it was stored. For such situations **stacks** are used. Another common way to retrieve data is in the same order as it was stored. For example, this might be useful with a bunch of things to do; often we want to do the first item first and the last item last. **Queues** are simple data structures that help in such common situations. The data structures like stacks and queues are more often used as programmer's tools. They are primarily conceptual aids rather than full-fledged storage devices. Their lifetime is typically shorter than that of the database-type structures (link lists, trees). They are created and used to carry out a particular task during the operation of a program; when the task is completed, they're discarded.

Some applications of queues are:

- ✓ *Semaphores*: Programmatic devices for synchronizing access to shared resources. When a process encounters a semaphore, it performs a test to determine whether someone else is currently accessing the resource the semaphore protects. If so, the process blocks and waits until another process signals that the resource is available. Since many processes may be waiting on a resource, some implementations of semaphores use a queue to determine who is next to go.
- ✓ *Event Handling*: A critical part of real-time programming. In real-time systems, events frequently occur when the system is not quite ready to handle them. Therefore, a queue keeps track of events so that they can be processed at a later time in the order they were received.
- ✓ *X Window System*: A network-based, graphical window system in which graphics are displayed on servers under the direction of client programs. X is a specific example of a system that does event handling. To manage events in real time, it uses a queue to store events until they can be processed.
- ✓ *Producer-consumer problem*: A generalization for modelling cooperating processes wherein one process, the *producer*, writes to a queue shared by another process, the *consumer*, which reads from it. The producer-consumer problem is a classic one to study because many applications can be described in terms of it.

15.3 DESCRIPTION OF QUEUES

The word *queue* is British for *line* (the kind you wait in). In Britain, to “queue up” means to get in line. In computer science a queue is a data structure that is somewhat like a stack, except that in a queue the first item inserted is the first to be removed (First-In-First-Out, FIFO), while in a stack, as we’ve seen, the last item inserted is the first to be removed (LIFO). A queue works like the line at the movies: The first person to join the rear of the line is the first person to reach the front of the line and buy a ticket. The last person to line up is the last person to buy a ticket (or—if the show is sold out—to fail to buy a ticket). The figure 15.1 shows how such a queue looks.

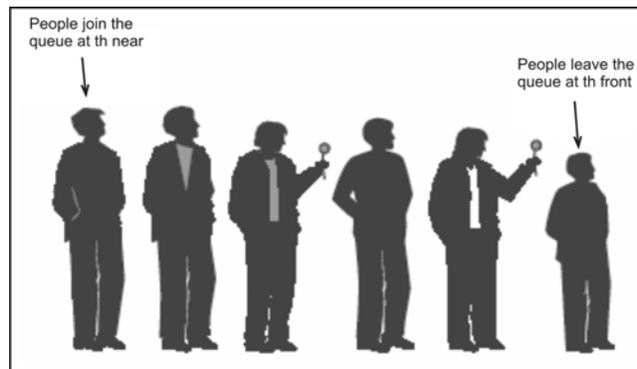


Figure 15.1: A queue of people

A queue can be defined as 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). Queues are efficient data structures useful for storing and retrieving data in a first-in, first-out, or FIFO, order. This allows us to retrieve data in the same order as it was stored.

In an array, any item can be accessed, either immediately (if its index number is known) or by searching through a sequence of cells until it’s found. In a queue, however, access is restricted: only one item can be read or removed at a given time. The interface of a queue is designed to enforce this **restricted access**. Access to other items is (in theory) not allowed.

Queues (like stacks) are more **abstract entities** than arrays and many other data storage structures. They're defined primarily by their interface: the permissible operations that can be carried out on them. The underlying mechanism used to implement them is typically not visible to their user.

The underlying mechanism for a queue, for example, can be an array, or it can be a linked list. The underlying mechanism for a linked list can be an array or a special kind of tree called a *heap*.

Queues also used to model real-world situations such as people waiting in line at a bank, airplanes waiting to take off, or data packets waiting to be transmitted over the Internet. There are various queues quietly doing their job in your computer's (or the network's) operating system. There's a printer queue where print jobs wait for the printer to be available. A queue also stores keystroke data as you type at the keyboard. This way, if you're using a word processor but the computer is briefly doing something else when you hit a key, the keystroke won't be lost; it waits in the queue until the word processor has time to read it. Using a queue guarantees the keystrokes stay in order until they can be processed.

In computing, to place an element at the rear of a queue, we enqueue it; to remove an element from the front of a queue, we dequeue it. Figure 15.2 illustrates a queue containing 5 elements. 59 is at the front of the queue and 80 is at the rear. In the figure, an element is inserted into the queue. Since an element is inserted from the rear in the queue, it changes the value at the rear from 80 to 6 (value of new element inserted).

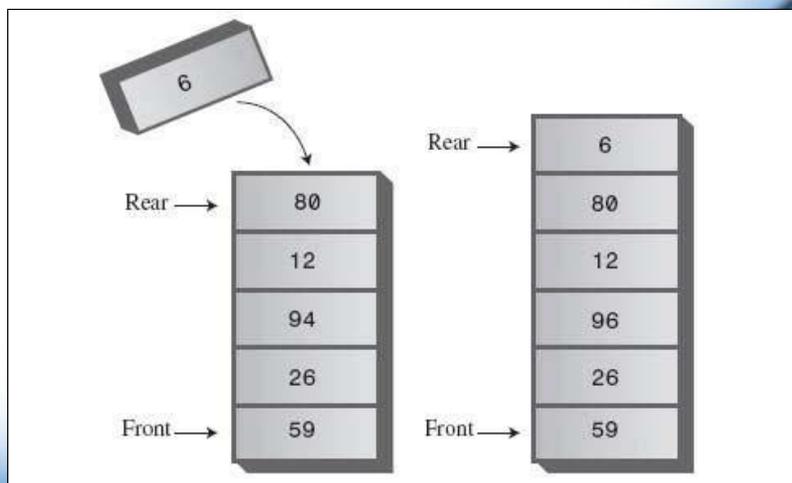


Figure 15.2: New item inserted at rear of queue

In figure 15.3 two elements have been deleted from the queue. Since elements may be deleted only from the front of the queue and one element at a time, 59 is removed first and 26 is now at the front. When 26 is deleted from the queue, 94 is now at the front.

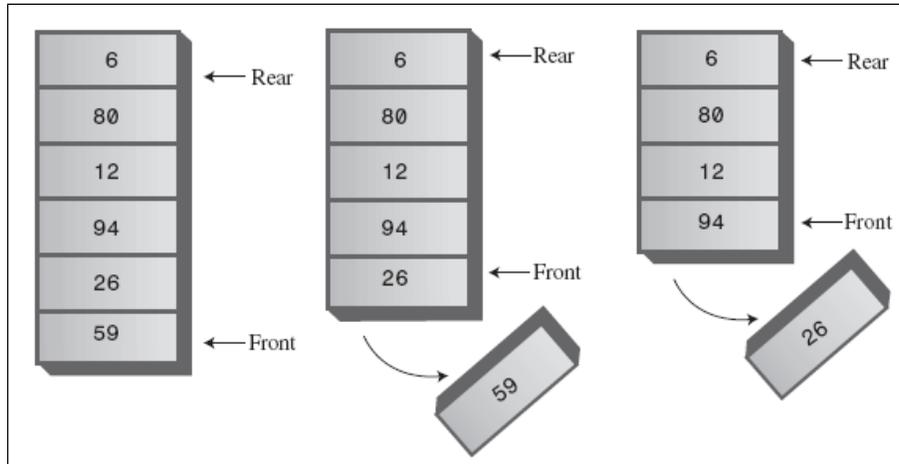


Figure 15.3: Two items removed from front of queue

15.4 ABSTRACTION

Now, let's think about a queue in an abstract way i.e., it doesn't hold any particular kind of thing and we aren't restricting ourselves to any particular programming language or any particular implementation of a queue.

Queues hold objects, usually all of the same type. Most queues support just a simple set of operations; and thus, the main property of a queue is that objects are inserted at the rear and removed from the front of the queue.

Here are the minimal operations we'd need for an abstract queue (and their typical names):

Insert: Places an object at the *rear* of the queue.

Remove: Removes an object from the *front* of the queue and produces that object.

IsEmpty: Reports whether the queue is empty or not.

Because we think of queues in terms of the physical analogy, we usually draw them horizontally, so there is always a **front** and **rear** of the queue.

As we are thinking about a queue in an abstract way, the *insert* operation can always be performed, since there is no limit to the number of elements a queue may contain. The *remove* operation, however, can be applied only if the queue is nonempty; there is no way to remove an element from a queue containing no elements. The result of an illegal attempt to remove an element from an empty queue is called **underflow**. The *empty* operation is, of course, always applicable.

15.5 UNDERSTANDING PRIMITIVE OPERATIONS

Let us try to understand the three primitive operations that can be applied to a queue without restricting ourselves to any programming language or any particular implementation of the queue.

15.5.1 Checking for Emptiness

Checking for emptiness is a simple operation which returns false or true depending on whether or not the queue contains any elements. Checking for emptiness is mandatory in case of removing an element as it is not possible to remove any element from an empty queue. Checking for emptiness in case of inserting an element enables to set the element to the front or rear of the queue depending on whether the queue is empty or not.

15.5.2 Inserting an element

As mentioned earlier, that in an abstract data type, the insert operation can always be performed, without any limit for the number of insertions. We introduce the insert operation for a queue which performs the following actions:

1. Check if queue is empty.
2. If queue is empty, set the rear to point to new element.
3. If queue is not empty, add element to the rear of the queue and set the rear to the new element.

15.5.3 Removing an element

The possibility of underflow must be considered when removing an element from a queue. Removing an element from an empty queue is not possible and therefore, we need to check if the queue is empty or not before trying to remove an element.

1. Check if queue is empty.
2. If queue is empty, display appropriate message and exit.
3. If queue is not empty, delete element from the front of the queue and set the front to the next element after the deleted element.

We also need to remember that if the queue contains only a single element then after removing that element we must set the rear to NULL.

15.6 IMPLEMENTATION USING C

Theoretically, one characteristic of a queue is that it does not have a specific capacity. Regardless of how many elements are already contained, a new element can always be added. It can also be empty, at which point removing an element will be impossible until a new element has been added again.

A practical implementation of a queue, e.g. with pointers, of course does have some capacity limit, that depends on the concrete situation it is used in. For a data structure the executing computer will eventually run out of memory, thus limiting the queue size. Queue **overflow** results from trying to add an element onto a full queue and queue **underflow** happens when trying to remove an element from an empty queue.

15.6.1 Array Implementation of Queues

How shall a queue be represented in C? One idea is to use an array to hold the elements of the queue and to use two variables, *front* and *rear*, to hold the positions within the array of the first and last elements of the queue.

Based on this idea let us see the following program

Example 15-1. Header for the Queue

```
/*  
 *   File: queue.h  
 *   Topic: Queue - Array Implementation  
 *   -----  
 *  
 */  
  
void insert(int *q,int *rear, int element);  
  
int remove(int *q,int *front);  
  
int full(int rear,const int size);  
  
int empty(int front,int rear);  
  
void init(int *front, int *rear);
```

Example 15-2. Implementation of the Queue

```
/*
```

```
*      File: queue.c
*
*      Topic: Queue - Array Implementation
*
* -----
*
*/

#include <stdio.h>

#include <stdlib.h>

#include "queue.h"

/***** Function Definitions *****/

/*initialize queue pointer*/
void init(int *front, int *rear)
{
    *front = *rear = 0;
}

/* insert an element
   precondition: the queue is not full
*/
void insert(int *q,int *rear, int element)
{
    q[(*rear)++] = element;
}

/* remove an element
```

```
    precondition: queue is not empty
*/
int remove(int *q,int *front)
{
    return q[(*front)++];
}

/* report queue is full nor not
   return 1 if queue is full, otherwise return 0
*/
int full(int rear,const int size)
{
    return rear == size ? 1 : 0;
}

/* report queue is empty or not
   return 1 if the queue is empty, otherwise return 0
*/
int empty(int front, int rear)
{
    return front == rear ? 1 : 0;
}
```

Example 15-3. Testing Implementation of the Queue

```
/*
 *   File: testqueue.c
 *   Topic: Queue - Array Implementation
```

```
* -----  
*  
*/  
  
#include <stdio.h>  
#include <stdlib.h>  
#include "queue.h"  
#define size 10  
  
void main()  
{  
    int front,rear,element;  
    int queue[size];  
  
    // initialize queue  
    init(&front,&rear);  
  
    // insert elements  
    while(full(rear,size) != 1)  
    {  
        element = rand();  
        printf("insert element %d\n",element);  
        insert(queue,&rear,element);  
        printf("front=%d,rear=%d\n",front,rear);  
        //press enter to insert more  
        getchar();  
    }  
}
```

```
printf("queue is full\n");

// remove elements from
while(!empty(front,rear))
{
    element = remove(queue,&front);
    printf("remove element %d \n",element)
    //press enter to pop more
    getchar();
}

printf("queue is empty\n");
getchar();}
```

In the above program, we use arrays to implement a queue. We have defined an array of size 10. You cannot insert more than 10 elements. If we try to insert, a message is printed stating that the queue is full. Similarly, a message stating that the queue is empty is printed when all 10 elements are removed.

At a stage when we have inserted 8 elements and removed 3, the queue will look like as shown in the figure below (Figure 15.4). Unlike the situation in a stack, the items in a queue don't always extend all the way down to index 0 in the array. After some items are removed, front will point at a higher index as shown in figure 15.4. It is possible to reach the absurd situation where the queue is empty, yet no new element can be inserted (see if you can come up with a sequence of insertions and deletions to reach that situation). Clearly, the array representation outlined in the foregoing is unacceptable.

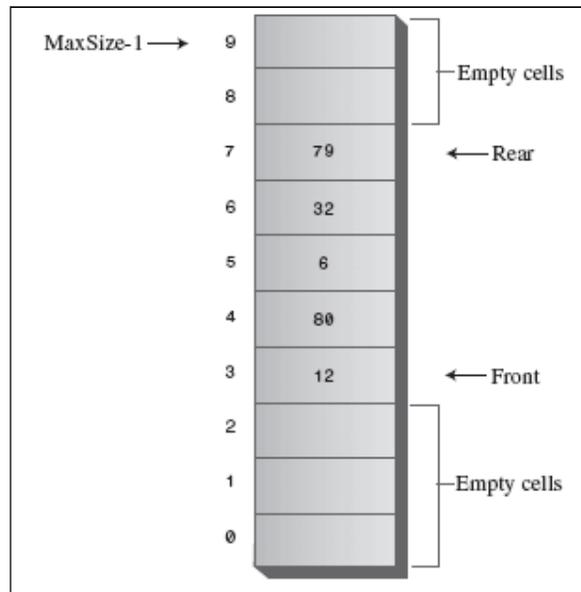


Figure 15.4: A queue with some items removed.

One solution is to modify the *remove* operation so that when an item is deleted, the entire queue is shifted to the beginning of the array. The remove operation would then be modified to

```
x = q.items[0];
for(i=0; i<q.rear; i++)
    q.items[i] = q.items[i+1];
q.rear--;
```

The queue need no longer contain a front field, since the element at position 0 of the array is always at the front of the queue.

This method, however, is too inefficient. Each deletion involves moving every remaining element of the queue. If a queue contains 500 or 1000 elements, this is clearly a high price to pay. Further, the operation of removing an element from the queue logically involves manipulation of only one element: the one currently at the front of the queue. The implementation of that operation should reflect this and should not involve a host of extraneous operations (see exercise 15.8.2 problem 4 for more efficient alternative).

Another solution is to view the array that holds the queue as a circle rather than as a straight line. That is, we imagine the first element of the array (that is, the element at position 0) as immediately following its last element. This implies that even if the last element is occupied, a new value can be inserted behind it in the first element of the array as long as that first element is empty. This can be clearly understood in the figure given below (figure 15.5).

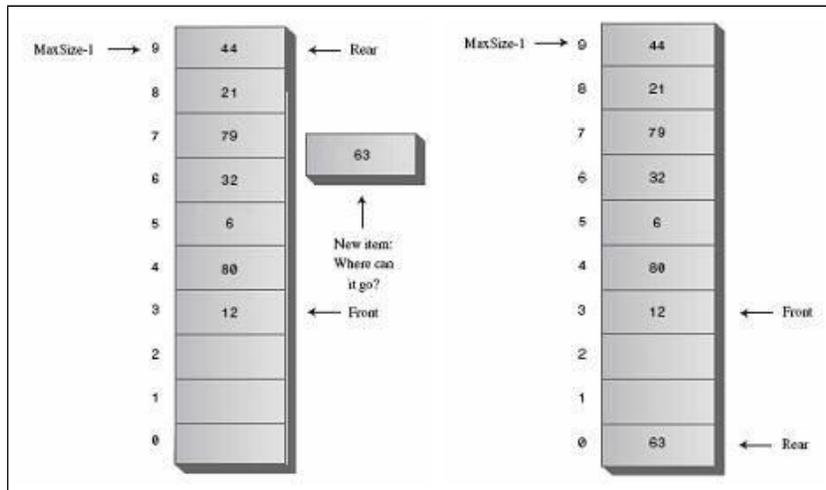


Figure 15.5: Circular Queue

In a straight queue, we can check for emptiness by evaluating the condition whether rear is less than front (i.e. $q.rear < q.front$). Unfortunately, it is difficult under this representation to determine when the queue is empty, as the mentioned condition is no longer valid. Figure 15.5 illustrates this situation.

One way of solving this problem is to establish the convention that the value of $q.front$ is the array index immediately preceding the first element rather than the index of the first element itself. Thus since $q.rear$ is the index of the last element, the condition $q.front = q.rear$ implies that the queue is empty.

Now, a queue of integers can be declared and initialized by

```
#define MAXQUEUE 100

struct queue
{
    int items[MAXQUEUE];
    int front, rear;
};

struct queue q;

q.front = q.rear = MAXQUEUE - 1;
```

Since $q.rear = q.front$, the queue is initially empty. The *empty* function can be coded as

```
empty(pq)
struct queue *pq;
{
    return ((pq->front == pq->rear) ? TRUE : FALSE);
}/* end empty */
```

The Remove Operation

The operation *remove* may be coded as

```
remove(pq)
struct queue *pq;
{
    if (empty(pq))
    {
```

```
        printf("queue underflow");  
        exit(1);  
    } /* end if */  
  
    if (pq->front == MAXQUEUE - 1)  
        pq->front = 0;  
    else  
        (pq->front)++;  
  
    return (pq->items[pq->front]);  
} /* end remove */
```

The Insert Operation

The insert operation involves testing for overflow, which occurs when the entire array is occupied by items of the queue and an attempt is made to insert yet another element into the queue. When an array is full, this situation is indicated by the fact that *q.front* equals *q.rear*, which is precisely the indication for underflow (.i.e. an empty queue). It seems that there is no way to distinguish between the empty queue and the full queue under this implementation. Such a situation is clearly unsatisfactory.

One solution is to sacrifice one element of the array and to allow a queue to grow only as large as one less than the size of the array. Thus, in an array of 100 elements can only contain 99 elements. An attempt to insert a hundredth element causes an overflow.

The *insert* routine may be written as follows:

```

insert(pq, x)
struct queue *pq;
int x;
{
    /* make room for new element */
    if (pq->rear == MAXQUEUE - 1)
        pq->rear = 0;
    else
        (pq->rear)++;

    /* check for overflow */
    if (pq->rear == pq->front)
    {
        printf("queue overflow");
        exit(1);
    } /* end if */

    pq->items[pq->rear] = x;
    return;
} /* end insert */

```

Note that the test for overflow in the *insert* operation occurs after *pq->rear* has been adjusted, whereas the test for underflow in the *remove* operation occurs immediately upon entering the routine, before *pq->front* is updated.

15.6.2 Linked List Implementation of Queues

Let us now examine how to represent a queue as a linked list. Recall that linked lists consist of a number of elements grouped, or *linked*, together in a specific order. Each element is called a node. Each node contains two fields: a "data" field to store information, and a "next" field which is a pointer to the next node.

In a queue items are deleted from the front of a queue and inserted at the rear. Let a pointer to the first element of a list represent the front of the queue. Another pointer to the last element of the list represents the rear of the queue, as shown in the figure 15.6.

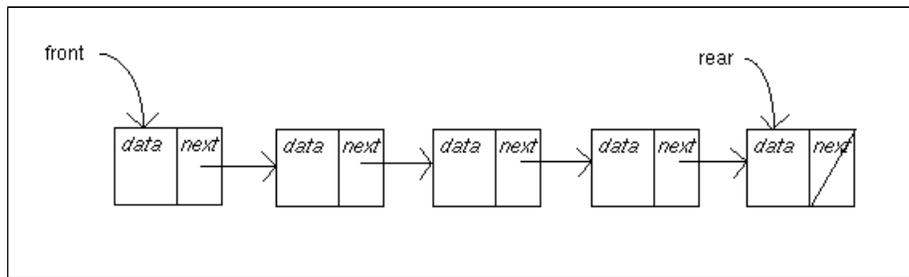


Figure 15.6: A Queue as a Linked List

Under the list representation, a queue consists of a list and two pointers, $q.front$ and $q.rear$. The algorithms for remove operation and insert operation are given below.

The operation $x = remove(q)$ is implemented by

```

if (empty(q))
{
    printf("queue underflow");
    exit(1);
}
p = q.front;
x = info(p);

```

```
q.front = next(p);  
  
if (q.front == null)  
    q.rear = null;  
  
freenode(p);  
return(x);
```

The operation *insert(q, x)* is implemented by

```
p = getnode();  
info(p) = x;  
next(p) = null;  
  
if (q.rear == null)  
    q.front = p;  
else  
    next(q.rear) = p;  
  
q.rear = p;
```

There are some disadvantages of representing a queue by a link list. Clearly, a node in a linked list occupies more storage than a corresponding element in an array, since two pieces of information per element are necessary in a list node (*info* and *next*), whereas only one piece of information is needed in the array implementation.

Another disadvantage is the additional time spent in managing the available list. Each addition and deletion of an element from a queue involves a corresponding deletion or addition to the available list.

An advantage of using linked list is that there are no restrictions on the number of insertions as in the case of arrays where the number of elements is pre-defined. Also storage is reserved as and when required, thus, enabling memory to be used for other purposes if required.

As a further illustration of how the C list implementations are used, we present C routines for manipulating a queue represented as a linear list. A queue is represented as a structure:

```
struct queue
{
    NODEPTR front, rear;
}q;
```

In the above declaration, *front* and *rear* are pointers to the first and last nodes of a queue represented as a list. The empty queue is represented by *front* and *rear* both equalising the null pointer. The function *empty* need check only one of these pointers, since in a nonempty queue neither *front* nor *rear* will be *NULL*.

```
empty(pq)
struct queue *pq;
{
    return ((pq->front == NULL) ? TRUE : FALSE);
}/* end empty */
```

The routine to insert an element into a queue may be written as follows:

```
insert(pq, x)
struct queue *pq
int x;
{
    NODEPTR p;
    p = getnode();
    p->info = x;
    p->next = NULL;
    if (pq->rear == NULL)
        pq->front = p;
    else
        (pq->rear)->next = p;
    pq->rear = p;
} /* end insert */
```

The function remove deletes the first element from the queue and returns its value:

```
remove(pq)
struct queue *pq;
{
    NODEPTR p;
    int x;
    if (empty(pq))
    {
        printf("queue underflow");
        exit(1);
    }
}
```

```
p = pq->front;
x = pq->info;
pq->front = p->next; if
(pq->front == NULL)
    pq->rear == NULL;
freenode(p);
return(x);
}/* end remove */
```

15.7 SUMMARY

- ✓ Queues are data structures usually used to simplify certain programming operations
- ✓ A queue allows access to only one data item .i.e. the first item that was inserted.
- ✓ The important queue operations are inserting an item at the rear of the queue and removing the item from the front of the queue.
- ✓ A queue can be implemented as a circular queue, which is based on an array in which the indices wrap around from the end of the array to the beginning.
- ✓ A queue can be implemented as a linked list, wherein the head of the list is the front of the queue and the tail of the list is the rear of the queue.

15.8 UNIT END EXERCISES

Questions

These questions are intended as a self-test for readers.

5. State the features and applications of queues?
6. Explain the basic operations of a queue?
7. Explain the concept of circular queue?
8. Mention the advantages and shortcomings of using a linear link list for a queue instead of an array?
9. Explain the concept of (with respect to queues)
 - a. FIFO
 - b. Overflow
 - c. Underflow
10. What set of conditions is necessary and sufficient for a sequence of insert and remove operations on a single empty queue to leave the queue empty without causing underflow?

Programming Projects

Writing programs that solve the Programming Projects helps to solidify your understanding of the material and demonstrates how the chapter's concepts are applied.

5. Write a program using the array implementation for circular queue.
6. Write a program using linked list implementation for queues.
7. Show how a sequence of insertions and removals from a queue represented by a linear array can cause overflow to occur upon an attempt to insert an element into an empty queue.
8. If an array holding a queue is not considered circular, the text suggests that each remove operation must shift down every remaining element of a queue. An alternative method is to postpone shifting until rear equals the last index of the array. When that situation occurs an attempt is made to insert an element into the queue, the entire queue is shifted down, so that

Thank You

