LESSON - 1

CONTENTS

1.0 Aims and Objectives

The objective of this lesson is to make the people to understand about fundamental concept of general programming constructs. After reading this lesson, reader can understand the way of developing programs based on well-defined approach.

1.1 Programming development methodologies

1.1.1 Introduction

Programming development methodologies are generally useful to make a program with easy-to-test and easy-to-change structure. These methodologies include various modules like

Problem Definition (state the problem clearly)
Algorithm Design (use *pseudo co*de)
Desktop Testing (work a simple problem by hand)
Write the Program (in small steps)
Testing (use a variety of data)

1.1.2 PROGRAMMING STYLE

Programming style refers to a set of rules or guidelines used when writing the source code for a computer program. It is often claimed that following a particular programming style will help programmers to quickly read and understand source code conforming to the style as well as helping to avoid introducing faults.

A classic work on the subject was *The Elements of Programming Style*, written in the 1970s, and illustrated largely with examples from the Fortran language prevalent at the time.

The programming style used in a particular program may be derived from the **coding standards** or **code conventions** of a company or other computing organization, as well as the preferences of the author of the code. Programming styles are often designed for a specific programming language (or language family), but some rules are commonly applied to many languages. (Style considered good in C source code might not be appropriate for BASIC source code, and so on.)

ELEMENTS OF GOOD STYLE

Good style, being a subjective matter, is difficult to concretely categorize. However, there are several elements common to a large number of programming styles. The issues usually considered as part of programming style, include the layout of the source code, including indentation; the use of white space around operators and keywords, the capitalization or otherwise of keywords and variable names, the style and spelling of user-defined identifiers, such as function, procedure and variable names, the use and style of comments and the use or avoidance of programming constructs themselves (such as GOTO statements).

CODE APPEARANCE

Programming styles commonly deal with the appearance of source code, with the goal of improving the readability of the program. However, with the advent of software that formats source code automatically, the focus on appearance will likely yield to a greater focus on naming, logic, and higher techniques. As a practical point, using a computer to format source code saves time, and it is possible to then enforce company-wide standards without debates.

INDENTING

Indent styles assist in identifying control flow and blocks of code. In programming languages that use indentation to delimit logical blocks of code, indentation style directly affects the behavior of the resulting program. In other languages, such as those that use brackets to delimit code blocks, the indentation style does not directly affect the product. Instead, using a logical and consistent indentation style makes code more readable. Compare:

```
if (hours < 24 && minutes < 60 && seconds < 60)
{
    return true;
}
else
{
    return false;
}
```

or

```
if (hours < 24 && minutes < 60 && seconds < 60) {
    return true;
} else {
    return false;
}
```

with something like

```
if( hours < 24 && minutes < 60 && seconds < 60 ){
return true;
  }else
  {
return false;
  }
```

The first two examples are probably much easier to read because they are indented in an established way (a "hanging paragraph" style). This indentation style is especially useful when dealing with multiple nested constructs. This example is somewhat contrived, of course; all the above are equivalent to the more concise statement

```
return (hours < 24) && (minutes < 60) && (seconds < 60);
```

VERTICAL ALIGNMENT

It is often helpful to align similar elements vertically, to make typo-generated bugs more obvious. Compare:

```
$search = array('a', 'b', 'c', 'd', 'e');
$replacement = array('foo', 'bar', 'baz', 'quux');
```

with:

```
$search      = array('a',   'b',   'c',   'd',   'e');
$replacement = array('foo', 'bar', 'baz', 'quux');
```

The latter example makes two things intuitively clear that were not clear in the former:

- the search and replace terms are related and match up: they are not discrete variables;
- there is one more search term than there are replacement terms. If this is a bug, it is now more likely to be spotted.

Arguments against vertical alignment generally claim difficulty in maintaining the alignment.

SPACING

The grammars of most free-format languages are mostly unconcerned with the presence or absence of whitespace. Making good use of spacing in code layout is therefore considered good programming style.

Compare the following syntactically equivalent examples of C code.

```
int i;
for(i=0;i<10;++i){
    printf("%d",i*i+i);
}
```

versus

```
int i;
for (i = 0; i < 10; ++i) {
    printf("%d", i*i + i);
}
```

It is also recommended to avoid using tab characters in the middle of a line as different text editors render their width differently.

1.1.3 APPROPRIATE VARIABLE NAMES

Appropriate choices for variable names are seen as the keystone for good style. Poorly-named variables make code harder to read and understand.

For example, consider the following pseudo code snippet:

```
get a b c
if a < 24 and b < 60 and c < 60
  return true
else
  return false
```

Because of the choice of variable names, the function of the code is difficult to work out. However, if the variable names are made more descriptive:

```
get hours minutes seconds
if hours < 24 and minutes < 60 and seconds < 60
  return true
else
  return false
```

The code's intent is easier to discern, namely, "Given a 24-hour time, true will be returned if it is a valid time and false otherwise".

BOOLEAN VALUES IN DECISION STRUCTURES

Some programmers think decision structures such as the above, where the result of the decision is merely computation of a Boolean value, are overly verbose and even prone to error. They prefer to have the decision in the computation itself, like this:

```
return (hours < 12) && (minutes < 60) && (seconds < 60);
```

The difference is often purely stylistic and syntactic, as modern compilers produce identical object code for both forms.

However, one argument in favour of the former is that some debuggers allow you to step line by line: if your test also caused changes to the variables you were testing, and you wanted to examine the values of all variables after that test, then only the former method would permit that to be debugged. The latter would not allow your debugger to reach a line "after the test" where those variables still existed.

LEFT-HAND COMPARISONS

In languages which use a single = for assignment and a double == for comparison, and where assignments may be made within control structures, it is sometimes considered good programming style to place constants to the left in any comparison.[1] The following lines are functionally identical:

```
if ( true == $a ) { ... }
if ( $a == true ) { ... }
```

However, of the two following, typed lines, the first is a syntax error that will be diagnosed by the interpreter, while the latter is a subtle bug that sets the value of `$a` to be true, then evaluates to true.

```
if ( true = $a ) { ... }
if ( $a = true ) { ... }
```

1.1.4 LOOPING AND CONTROL STRUCTURES

The use of logical control structures for looping adds to good programming style as well. It helps someone reading code to understand the program's sequence of execution (in imperative programming languages). For example, in pseudo code:

```
i = 0
while i < 5
  print i * 2
  i = i + 1
end while
```

The above snippet obeys the naming and indentation style guidelines, but the following use of the "for" construct makes the code much easier to read:

```
for i = 0, i < 5, i=i+1
  print i * 2
```

In many languages, the often used "for each element in a range" pattern can be shortened to:

```
for i = 0 to 5
  print i * 2
print "Ended loop"
```

In curly bracket programming languages, it has become common for style documents to require that even where optional, curly brackets be placed after all control flow constructs.

```
for (i = 0 to 5) {
  print i * 2;
}
print "Ended loop";
```

This prevents program-flow bugs which can be time-consuming to track down, such as where a terminating semicolon is introduced at the end of the construct (a common typo):

```
for (i = 0; i < 5; ++i);
   printf("%d\n", i*2);    /* The incorrect indentation hides the
fact that this line is not part of the loop body. */
puts("Ended loop");
```

where another line is added before the first:

```
      for (i = 0; i < 5; ++i)
         fprintf(logfile, "loop reached %d\n", i);
   printf("%d\n", i*2);    /* The incorrect indentation hides the fact
that this line is not part of the loop body. */
puts("Ended loop");
```

LISTS

Where items in a list are placed on separate lines, it is sometimes considered good practice to add the item-separator after the final item, as well as between each item, at least in those languages where doing so is supported by the syntax (e.g, C):

```
const char *array[] = {
    "item1",
    "item2",
    "item3",  /* still has the comma after it */
};
```

1.2 PROBLEM SOLVING TECHNIQUES

Generally, problem can be solved using several different approaches as follows

1. Divide and conquer: break down large, complex problem into smaller, solvable problems
2. Hill-climbing strategy, (or - rephrased - gradient descent/ascent, difference reduction) - attempting at every step to move closer to the goal situation. The problem with this approach is that many challenges require that you seem to move away from the goal state in order to clearly see the solution.
3. Means-end analysis, more effective than hill-climbing, requires the setting of sub goals based on the process of getting from the initial state to the goal state when solving a problem.
4. Working backwards
5. Trial-and-error
6. Brainstorming
7. Morphological analysis
8. Method of focal objects

9. Lateral thinking
10. George Polya's techniques in *How to Solve It*
11. Research: study what others have written about the problem (and related problems). Maybe there's already a solution?
12. Assumption reversal (write down your assumptions about the problem, and then reverse them all)
13. Analogy: has a similar problem (possibly in a different field) been solved before?
14. Hypothesis testing: assuming a possible explanation to the problem and trying to prove the assumption.
15. Constraint examination: are you assuming a constraint that does not really exist?
16. Incubation: input the details of a problem into your mind, and then stop focusing on it. The subconscious mind will continue to work on the problem, and the solution might just "pop up" while you are doing something else
17. Build (or write) one or more abstract models of the problem
18. Try to prove that the problem cannot be solved. Where the proof breaks down can be your starting point for resolving it
19. Get help from friends or online problem solving community (e.g. 3form)
20. Delegation: delegating the problem to others.
21. Root Cause Analysis

1.3 ALGORITHM

An **algorithm, in general,** is a definite list of well-defined instructions for completing a task; that given an initial state, will proceed through a well-defined series of successive states, eventually terminating in an end-state.

Generally, algorithm can be classified in to following categories

(1). Recursive – an algorithm called itself. It may be direct or indirect recursive

(2). Iterative - some portion of algorithm will be repeatedly executed

(3). Deterministic – Output of the algorithm can be defined uniquely.

(4). Non deterministic – Output can not defined uniquely but we can expect some choice

(5). Heuristic algorithm. – (i)This algorithm usually produce a good solution even though such a solution need not be an optimum one.

(ii). These algorithms can be modified easily.

For example , Consider a general task like you are receiving a friend arriving at the airport, and your friend needs to get from the airport to your house. Here are four different algorithms that you might give your friend for getting to your home:

- **The taxi algorithm**:
    1. Go to the taxi stand.
    2. Get in a taxi.
    3. Give the driver my address.
- **The call-me algorithm**:
    1. When your plane arrives, call my cell phone.
    2. Meet me outside baggage claim.
- **The rent-a-car algorithm**:
    1. Take the shuttle to the rental car place.
    2. Rent a car.
    3. Follow the directions to get to my house.
- **The bus algorithm**:
    1. Outside baggage claim, catch bus number 70.
    2. Transfer to bus 14 on Main Street.
    3. Get off on Elm street.
    4. Walk two blocks north to my house.

All four of these algorithms accomplish exactly the same goal, but each algorithm does it in completely different way. Each algorithm also has a different cost and a different travel time. Taking a taxi, for example, is probably the fastest way, but also the most expensive. Taking the bus is definitely less expensive, but a whole lot slower. You choose the algorithm based on the circumstances.

In computer programming, there are often many different ways -- algorithms -- to accomplish any given task. Each algorithm has advantages and disadvantages in different situations.

1.4 PSEUDO CODE

Pseudo code is a kind of structured English for describing algorithms. It allows the designer to focus on the logic of the algorithm without being distracted by details of language syntax. At the same time, the pseudo code needs to be complete. It describe the entire logic of the algorithm so that implementation becomes a rote mechanical task of translating line by line into source code.

In general the vocabulary used in the pseudo code should be the vocabulary of the problem domain, not of the implementation domain. The pseudo code is a narrative for someone who knows the requirements (problem domain) and is trying to learn how the solution is organized. E.g.,

```
Extract the next word from the line (good)
set word to get next token (poor)
```

```
Append the file extension to the name (good)
name = name + extension (poor)

FOR all the characters in the name (good)
FOR character = first to last (ok)
```

Note that the logic must be decomposed to the level of a single loop or decision. Thus "Search the list and find the customer with highest balance" is too vague because it takes a loop AND a nested decision to implement it. It's okay to use "Find" or "Lookup" if there's a predefined function for it such as `String.indexOf()`.

Each textbook and each individual designer may have their own personal style of pseudocode. Pseudocode is not a rigorous notation, since it is read by other people, not by the computer. There is no universal "standard" for the industry, but for instructional purposes it is helpful if we all follow a similar style. The format below is recommended for expressing your solutions in our class.

The "structured" part of pseudo code is a notation for representing six specific structured programming constructs: SEQUENCE, WHILE, IF-THEN-ELSE, REPEAT-UNTIL, FOR, and CASE. Each of these constructs can be embedded inside any other construct. These constructs represent the logic, or flow of control in an algorithm.

It has been proven that three basic constructs for flow of control are sufficient to implement any "proper" algorithm.

SEQUENCE is a linear progression where one task is performed sequentially after another.

WHILE is a loop (repetition) with a simple conditional test at its beginning.

IF-THEN-ELSE is a decision (selection) in which a choice is made between two alternative courses of action.

Although these constructs are sufficient, it is often useful to include three more constructs:

REPEAT-UNTIL is a loop with a simple conditional test at the bottom.

CASE is a multiway branch (decision) based on the value of an expression. CASE is a generalization of IF-THEN-ELSE.

FOR is a "counting" loop.

SEQUENCE

Sequential control is indicated by writing one action after another, each action on a line by itself, and all actions aligned with the same indent. The actions are performed in the sequence (top to bottom) that they are written.

Example 1.1 (non-computer)

```
Brush teeth
Wash face
Comb hair
Smile in mirror
```

Example 1.2

```
READ height of rectangle
READ width of rectangle
COMPUTE area as height times width
```

Common Action Keywords

Several keywords are often used to indicate common input, output, and processing operations.

Input: READ, OBTAIN, GET
Output: PRINT, DISPLAY, SHOW
Compute: COMPUTE, CALCULATE, DETERMINE
Initialize: SET, INIT
Add one: INCREMENT, BUMP

IF-THEN-ELSE

Binary choice on a given Boolean condition is indicated by the use of four keywords: IF, THEN, ELSE, and ENDIF. The general form is:

```
IF condition THEN
sequence 1
ELSE
sequence 2
ENDIF
```

The ELSE keyword and "sequence 2" are optional. If the condition is true, sequence 1 is performed, otherwise sequence 2 is performed.

Example 1.3

```
IF HoursWorked > NormalMax THEN
```

```
Display overtime message
ELSE
Display regular time message
ENDIF
```

WHILE

The WHILE construct is used to specify a loop with a test at the top. The beginning and ending of the loop are indicated by two keywords WHILE and ENDWHILE. The general form is:

```
WHILE condition
      sequence
ENDWHILE
```

The loop is entered only if the condition is true. The "sequence" is performed for each iteration. At the conclusion of each iteration, the condition is evaluated and the loop continues as long as the condition is true.

Example 1.4

```
WHILE Population < Limit
Compute Population as Population + Births - Deaths
ENDWHILE
```

Example 1.4

```
WHILE employee.type NOT EQUAL manager AND personCount <
numEmployees
INCREMENT personCount
CALL employeeList.getPerson with personCount RETURNING employee
ENDWHILE
```

CASE

A CASE construct indicates a multiway branch based on conditions that are mutually exclusive. Four keywords, CASE, OF, OTHERS, and ENDCASE, and conditions are used to indicate the various alternatives. The general form is:

```
CASE expression OF
condition 1 : sequence 1
condition 2 : sequence 2
...
condition n : sequence n
```

```
OTHERS:
default sequence
ENDCASE
```

The OTHERS clause with its default sequence is optional. Conditions are normally numbers or characters

indicating the value of "expression", but they can be English statements or some other notation that specifies the condition under which the given sequence is to be performed. A certain sequence may be associated with more than one condition.

Example 1.6

```
CASE  Title  OF
    Mr     : Print "Mister"
    Mrs    : Print "Missus"
    Miss   : Print "Miss"
    Ms     : Print "Mizz"
    Dr     : Print "Doctor"
ENDCASE
```

Example 1.7

```
CASE  grade  OF
        A       : points = 4
        B       : points = 3
        C       : points = 2
        D       : points = 1
        F       : points = 0
ENDCASE
```

REPEAT-UNTIL

This loop is similar to the WHILE loop except that the test is performed at the bottom of the loop instead of at the top. Two keywords, REPEAT and UNTIL are used. The general form is:

```
REPEAT
    sequence
UNTIL condition
```

The "sequence" in this type of loop is always performed at least once, because the test is performed after the sequence is executed. At the conclusion of each iteration, the condition is evaluated, and the loop repeats if the condition is false. The loop terminates when the condition becomes true.

FOR

This loop is a specialized construct for iterating a specific number of times, often called a "counting" loop. Two keywords, FOR and ENDFOR are used. The general form is:

```
FOR iteration bounds
        sequence
ENDFOR
```

In cases where the loop constraints can be obviously inferred it is best to describe the loop using problem domain vocabulary.

Example 1.8

```
FOR each month of the year (good)
FOR month = 1 to 12 (ok)

FOR each employee in the list (good)
FOR empno = 1 to listsize (ok)
```

NESTED CONSTRUCTS

The constructs can be embedded within each other, and this is made clear by use of indenting. Nested constructs should be clearly indented from their surrounding constructs.

Example 1.9

```
SET total to zero
REPEAT
READ Temperature
IF Temperature > Freezing THEN
    INCREMENT total
END IF
UNTIL Temperature < zero
Print total
```

In the above example, the IF construct is nested within the REPEAT construct, and therefore is indented.

INVOKING SUBPROCEDURES

Use the `CALL` keyword. For example:

```
CALL AvgAge with StudentAges
CALL Swap with CurrentItem and TargetItem
CALL Account.debit with CheckAmount
CALL getBalance RETURNING aBalance
CALL SquareRoot with orbitHeight RETURNING nominalOrbit
```

EXCEPTION HANDLING

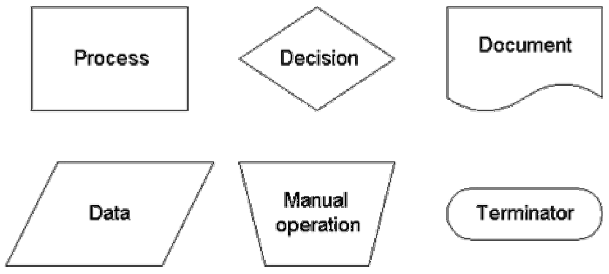
```
    BEGIN
        statements
    EXCEPTION
        WHEN exception type
            statements to handle exception
        WHEN another exception type
            statements to handle exception
    END
```

1.5 FLOW CHART

A flow chart is defined as a pictorial representation describing a process being studied or even used to plan stages of a project. Flow charts tend to provide people with a common language or reference point when dealing with a project or process.

Four particular types of flow charts have proven useful when dealing with a process analysis: top-down flow chart, detailed flow chart, work flow diagrams, and a deployment chart. Each of the different types of flow charts tend to provide a different aspect to a process or a task. Flow charts provide an excellent form of documentation for a process, and quite often are useful when examining how various steps in a process work together.

When dealing with a process flow chart, two separate stages of the process should be considered: the finished product and the making of the product. In order to analyze the finished product or how to operate the process, flow charts tend to use simple and easily recognizable symbols. The basic flow chart symbols below are used when analyzing how to operate a process.



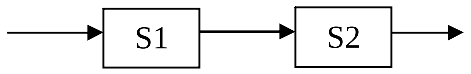
1.5 Structured Flowchart

A flow chart which has been constructed using only the following one-entry one-exit control structures is called structured flowchart, The three basic structures are,

(i). Sequential
(ii). Selection
(iii). Iteration

1.5.1 Sequential construct

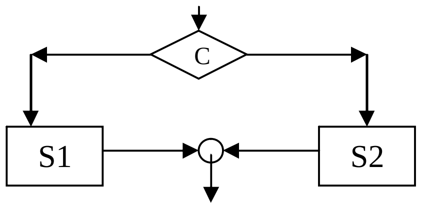
When statements are executed in sequence, they are said to form sequential construct.



Where, S1 and S2 are statement blocks, which may consist of either single or more than one statement.

1.5.2 Selection construct

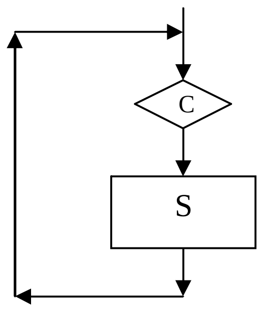
This construct is also known as conditional structure and is used to indicate decision in a program. This construct will be represented diagrammatically as follows,



If condition 'C' is true then it executes S1 otherwise it executes S2.

1.5.3 Iterative

This construct operates a sequence of statements while some condition is satisfied. For example, the While structure can be represented as follows,



The statement block will be executed until condition become false.

1.6 Let us Sum Up

In This lesson, We

- Described about the way of writing programs
- Various program development methodologies
- Pseudo Code representation
- Effective coding techniques
- Structured flow chart

1.7 Points for discussion

Can we use selection construct in iteration construct?
Describe the elements of structured programming.
Do we need different style of coding?

1.8 Check your progress

(i). Define structured program.

If the program using any one of structured construct (ie., sequence, selection and iteration) then the program is referred as structured programming.

(ii). What is meant by flow chart?

The flow chart is pictorial representation of a problem.

1.9 Lesson-end Activities

1. What are all the symbols we are using to construct a flow chart?
2. Why we need structured programming approach?
3. What do you mean by algorithm?

1.10 References

1. Peter Aitken, Teach Yourself C in 21 Days, Fourth Edition, Sams Publisher.
2. Brian W. Kernighan and Dennis M. Ritchie, The C programming Language, Prentice-Hall in 1988
3. E.Balagursamy, Programming in Ansi C, TATA MCGraw Hill
4. http://www.cs.cf.ac.uk/Dave/C
5. http://imada.sdu.dk/~svalle/courses

LESSON : 2 Fundamentals of “C” Programming

CONTENTS

2.0 Aims and Objectives.

This lesson will briefly elucidate basic concepts of “C” programming, which helps readers to understand about the structure of program as well as to know about fundamental concepts used in “C”. After reading this lesson, I am sure that, you will be able to implement small programs using ‘C’.

2.1 Introduction to C

C was created by Dennis Ritchie at the Bell Telephone Laboratories in 1972. The language wasn't created for the fun of it, but for a specific purpose: to design the UNIX operating system. From the beginning, C was intended to be useful to allow busy programmers to get things done. Because C is such a powerful and flexible language, its use quickly spread beyond Bell Labs. Programmers everywhere began using it to write all sorts of programs. Soon, however, different organizations began utilizing their own versions of C, and subtle differences between implementations started to cause programmers headaches. In response to this problem, the American National Standards Institute (ANSI) formed a committee in 1983 to establish a standard definition of C, which became known as ANSI Standard C. With few exceptions, every modern C compiler has the ability to adhere to this standard.

C is a relatively ``low-level" language. This characterization is not pejorative; it simply means that C deals with the same sort of objects that most computers do, namely characters, numbers, and addresses. These may be combined and moved about with the arithmetic and logical operators implemented by real machines.

C is widely promoted as ideal portable and effective language. This characterization is deserved when C is considered for systems-level programs such as compilers, or for mass-market products such as word processing or spreadsheet programs. C was designed as a reasonably transportable replacement for assembly language that would add some high-level language constructs, but would retain almost all the low-level procedural capabilities found at the machine instruction level.

2.2 Basic concepts

2.2.1 Importance of C

In computer programming, there are many high-level languages, such as C, Pascal, BASIC, and Java. These are all excellent languages suited for most programming tasks. Even so, there are several reasons why many computer professionals feel that C is at the top of the list:

- C is a powerful and flexible language. C is used for projects as diverse as operating systems, word processors, graphics, spreadsheets, and even compilers for other languages.

- C is a popular language preferred by professional programmers. As a result, a wide variety of C compilers and helpful accessories are available.

- C is a portable language. *Portable* means that a C program written for one computer system (an IBM PC, for example) can be compiled and run on another system (a DEC VAX system, perhaps) with little or no modification. Portability is enhanced by the ANSI standard for C, the set of rules for C compilers.

- C is a language of few words, containing only a handful of terms, called *keywords*, which serve as the base on which the language's functionality is built. We might think that a language with more keywords (sometimes called *reserved words*) would be more powerful. This isn't true. As we program with C, We will find that it can be programmed to do any task.

- C is modular. C code can (and should) be written in routines called *function*s. These functions can be reused in other applications or programs. By passing pieces of information to the functions, you can create useful, reusable code.

2.2.2 Hurdles in C

The fundamental problem with C is that it doesn't hide enough machine-level details. A good example is the central role that pointer variables play in C programs. C pointers were designed to provide machine-independent address arithmetic; and, for the most part, pointers do make it easier to write system programs that transport across machines. (Even this advantage is qualified, however, because pointers don't always transport easily between machines with flat addresses — e.g., Vax — and machines with segmented addresses — e.g., Intel 808x.). But at an application level, C pointers are a burden and a danger. They're burdensome because the programmer has to attend to details that a compiler can readily handle. For example, in C, to use a function (procedure) parameter as an output parameter (i.e., one that changes a value in the calling function), you have to pass the address of the variable that is to receive the value.

2.2.3 Structure of C Programming

C program may contain one or more sections as shown in the figure 1.

```
Documentation Section
Link Section
Definition section
Global Declaration Section
Main ( )
{
        Declaration Part;
        Executable part;
}
Subprogram Section ( )

Function 1
Function 2
Function 3
.                ( User-defined functions )
.
.
Function n
```

The documentation section consists of a set of comment lines giving the name of the program, author, date of written the program and other details which programmer would like to use. This section is also referred as comment section. Information in this section should be enclosed with /*………*/. The compiler will simply skip the information between /* and */.

Example 2.1

```
/* Program for Inventory Control System, Date 01-10-2007 */
```

The link section provides instructions to the compiler to link functions from system library. Generally all the header files are mentioned in link section.

Example 2.2

```
#include <stdio.h>
#include <dos.h>
#include <conio.h>
#include <math.h>
#include <string.h>
```

The definition section contains all the symbolic constants. These constants can not be changed during the execution. These constants are automatically recognized by the system and no need to declare the respective labels.

Example 2.3

```
#define limit 5
#define pi 3.14.
```

Some variables are used in more than one function. Such variables are called global variables and are declared in global declaration section that is outside of all the functions.

Example 2.4

```
int a ; char name[16];
int bignumber,bigsum;
char letter;
main()
 {

 }
```

Every C program must have one main() function section. This section contains two parts, declaration part and executable part. The declaration part declares all the variables used in the executable part. There is at least one statement in the executable part. These two parts must appear between two the opening and closing braces. All the statements in the declaration part and executable part must be ended with semicolon.

The subprogram section contains user defined functions that are called from main function. These functions are generally placed immediately after main in any order.

```
#include <stdio.h>         include information about standard library
main()                     define a function called main
```

```
{                         that received no argument values
                          statements of main are enclosed in braces
printf("hello, world\n"); main calls library function printf
                          to print this sequence of characters
}                         \n represents the newline character
```

Example 2.5 The first C program

2.2.4 Character Set Delimiters

Characters play a central role in Standard C. C program can be represent as one or more **source files**. The translator reads a source file as a text stream consisting of characters that you can read when you display the stream on a terminal screen or produce hard copy with a printer. You often manipulate text when a C program executes. The program might produce a text stream that people can read, or it might read a text stream entered by someone typing at a keyboard or from a file modified using a text editor. This document describes the characters that you use to write C source files and that you manipulate as streams when executing C programs.

Every character set contains a distinct code value for each character in the **basic C character set**. A character set can also contain additional characters with other code values. For example:

- The **character constant** `'x'` becomes the value of the code for the character corresponding to `x` in the target character set.
- The **string literal** `"xyz"` becomes a sequence of character constants stored in successive bytes of memory, followed by a byte containing the value zero: `{'x', 'y', 'z', '\0'}`

A string literal is one way to specify a **null-terminated string**, an array of zero or more bytes followed by a byte containing the value zero.

Visible graphic characters in the basic C character set:

Form	**Members**
letter	A B C D E F G H I J K L M
	N O P Q R S T U V W X Y Z
	a b c d e f g h i j k l m
	n o p q r s t u v w x y z
digit	0 1 2 3 4 5 6 7 8 9
underscore	_
punctuation	! " # % & ' ( ) * + , - . / :
	; < = > ? [\ ] ^ { | } ~

Additional graphic characters in the basic C character set:

Character	**Meaning**
space	**leave blank space**
BEL	**signal an alert (BELl)**
BS	**go back one position (Backspace)**
FF	**go to top of page (Form Feed)**
NL	**go to start of next line (NewLine)**
CR	**go to start of this line (Carriage Return)**
HT	**go to next Horizontal Tab stop**
VT	**go to next Vertical Tab stop**

The code value zero is reserved for the **null character** which is always in the target character set. Code values for the basic C character set are positive when stored in an object of type *char*. Code values for the digits are contiguous, with increasing value. For example, `'0' + 5` equals `'5'`. Code values for any two letters are *not* necessarily contiguous.

2.2.5 Keywords

Every C word is classified as either a keyword or an identifier. All keywords have fixed meanings and these meanings cannot be changed. **In general all the key** words must be written in lower case. The C language uses the following keywords:

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

2.2.6 Identifiers.

Identifiers refer to the names of variables, functions and arrays. These are user defined names and consist of sequence of letters and digits, with a letter as first character. Both upper case and lower case letters are permitted, although lower case letters are commonly used. The underscore character is also permitted in identifiers. It is usually used as a link between two words in long identifiers.

2.3 Constants

Constant in C refers to the values that can not be changed during the execution of the program. The constants in “C” can be classified as follows.

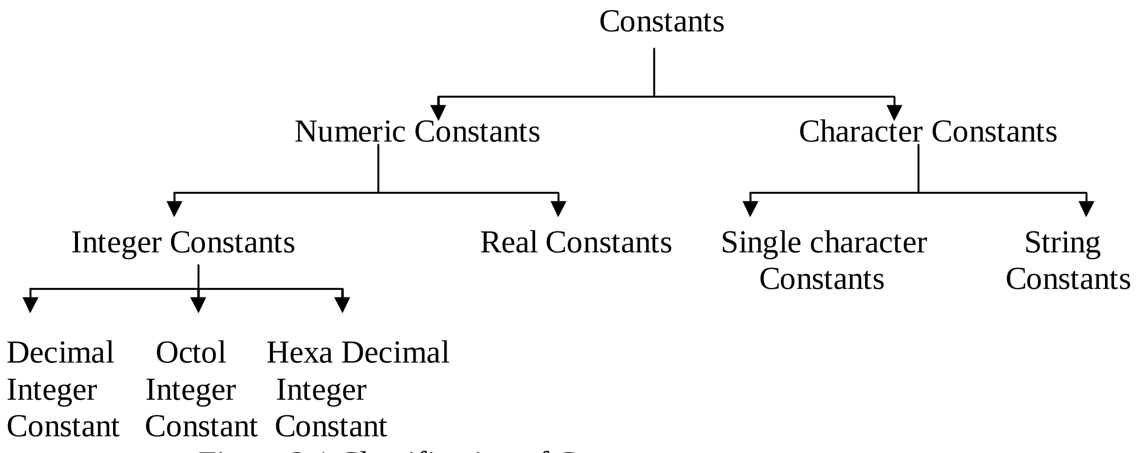


Figure 2.1 Classification of Constants

Integer constants

The normal integral constants are obvious: things like 1, 1034 and so on. You can put l or L at the end of an integer constant to force it to be long. To make the constant unsigned, one of u or U can be used to do the job.

Integer constants can be written in hexadecimal by preceding the constant with 0x or 0X and using the upper or lower case letters a, b, c, d, e, f in the usual way. Be careful about octal constants. They are indicated by starting the number with 0 and only using the digits 0, 1, 2, 3, 4, 5, 6, 7. It is easy to write 015 by accident, or out of habit, and not to realize that it is not in decimal. The mistake is most common with beginners, because experienced C programmers already carry the scars.

The Standard has now invented a new way of working out what type an integer constant is. In the old days, if the constant was too big for an int, it got promoted to a long (without warning). Now, the rule is that a plain decimal constant will be fitted into the first in this list,

int   long   unsigned long
that can hold the value.

Plain octal or hexadecimal constants
will use this list

int   unsigned int   long   unsigned long
If the constant is suffixed by u or U:

unsigned int   unsigned long
If it is suffixed by l or L:

long   unsigned long and finally, if it suffixed by both u or U and l or L, it can only be an unsigned long. All that was done to try to give you ‘what you meant’; what it does mean is that it is hard to work out exactly what the type of a constant expression is if you don't know something about the hardware. Hopefully, good compilers will warn when a constant is promoted up to another length and the U or L etc. is not specified.

A nasty bug hides here:

```
        printf("value of 32768 is %d\n", 32769);
```

On a 16-bit two's complement machine, 32769 will be a long by the rules given above. But Printf is only expecting an int as an argument (the %d indicates that). The type of the argument is just wrong. For the ultimate in safety-conscious programming, you should cast such cases to the right type:

```
        printf("value of 32768 is %d\n", (int)32769);
```

It might interest you to note that there are no negative constants; writing -23 is an expression involving a positive constant and an operator.

Character Constants

Single character constants

Single character constants are formed by using a single character from C character set enclosed in single quotes. Like wise String constants are formed by using either a single character or more than one character enclosed with double quotes.

Character constants actually have type int (for historical reasons) and are written by placing a sequence of characters between single quote marks:

```
'e'
't'
'like that'
```

Wide character constants are written just as above, but prefixed with L:

```
L'e'
L't'
L'like that'
```

Regrettably it is valid to have more than one character in the sequence, giving a machine-dependent result. Single characters are the best from the portability point of view, resulting in an ordinary integer constant whose value is the machine representation of the single character. The introduction of extended characters may cause you to stumble over this by accident; if '<a>' is a multibyte character (encoded with a shift-in shift-out around it) then '<a>' will be a plain character constant, but containing several characters, just like the more obvious 'abcde'. This is bound to lead to trouble in the future; let's hope that compilers will warn about it.

A string constant is a sequence of characters enclosed in *double* quotes. The characters may be letters, numbers, special characters and blank space. Examples are:

"Hello'"
"1027"
"WELL COME"
“a*c”

Remember that a character constant (e.g., 'X') is not equivalent to the single character String constant (e.g., "X"). Further, a single character string constant does not have an equivalent integer value while a character constant has an integer value. Character strings are often used in programs to build meaningful programs

To ease the way of representing some special characters that would otherwise be hard to get into a character constant (or hard to read; does `' '` contain a space or a tab?), there is what is called an escape sequence which can be used instead. Table 2.1 shows the *escape sequences* defined in the Standard.

Sequence	**Represents**
`\a`	audible alarm
`\b`	backspace
`\f`	form feed
`\n`	newline
`\r`	carriage return
`\t`	tab
`\v`	vertical tab
`\\`	backslash
`\'`	quote
`\"`	double quote
`\?`	question mark

Table 2.1. C escape sequences

It is also possible to use numeric escape sequences to specify a character in terms of the internal value used to represent it. A sequence of either `\ooo` or `\xhhhh`, where the `ooo` is up to three octal digits and `hhhh` is any number of hexadecimal digits respectively. A common version of it is `'\033'`, which is used by those who know that on an ASCII based machine, octal `33` is the ESC (escape) code. Beware that the hexadecimal version will absorb any number of valid following hexadecimal digits; if you want a string containing the character whose value is hexadecimal `ff` followed by a *letter* `f`, then the safe way to do it is to use the string joining feature:

```
"\xff" "f"
```

The string

```
"\xfff"
```

only contains one character, with all three of the `fs` eaten up in the hexadecimal sequence. Some of the escape sequences aren't too obvious, so a brief explanation is needed. To get a single quote as a character constant you type `'\''`, to get a question mark you may have to use `'\?'`; not that it matters in that example, but to get two of them in there you can't use `'??'`, because the sequence `??'` is a trigraph! You would have to use `'\?\?'`. The escape `\"` is only necessary in strings, which will come later.

There are two distinct purposes behind the escape sequences. It's obviously necessary to be able to represent characters such as single quote and backslash unambiguously: that is one purpose. The second purpose applies to the following sequences which control the motions of a printing device when they are sent to it, as follows:

`\a`
: Ring the bell if there is one. Do not move.

`\b`
: Backspace.

`\f`
: Go to the first position on the ‘next page’, whatever that may mean for the output device.

`\n`
: Go to the start of the next line.

`\r`
: Go back to the start of the current line.

`\t`
: Go to the next horizontal tab position.

`\v`
: Go to the start of the line at the next vertical tab position.

For `\b`, `\t`, `\v`, if there is no such position, the behavior is unspecified. The Standard carefully avoids mentioning the physical directions of movement of the output device which are not necessarily the top to bottom, left to right movements common in Western cultural environments.

It is guaranteed that each escape sequence has a unique integral value which can be stored in a `char`.

Real constants

Real constants are formed by using the numbers 0 to 9 with decimal point. Some examples for real constants are as follows;

```
        1.0
```

```
2.
.1
2.634
.125
2.e5
2.e+5
.125e-3
2.5e5
3.1E-6
```

and so on. For readability, even if part of the number is zero, it is a good idea to show it:

```
1.0
0.1
```

The exponent part shows the number of powers of ten that the rest of the number should be raised to, so

```
3.0e3
```

is equivalent in value to the integer constant

```
3000
```

As you can see, the `e` can also be `E`. These constants all have type `double` unless they are suffixed with `f` or `F` to mean `float` or `l` or `L` to mean long double.

For completeness, here is the formal description of a real constant:

A real constant is one of:

- A *fractional constant* followed by an optional *exponent*.
- A *digit sequence* followed by an *exponent*.

In either case followed by an optional one of `f`, `l`, `F`, `L`, where:

- A *fractional constant* is one of:
  - o An optional *digit sequence* followed by a decimal point followed by a *digit sequence*.
  - o A *digit sequence* followed by a decimal point.
- An exponent is one of
  - o `e` or `E` followed by an optional `+` or `-` followed by a *digit sequence*.
- A *digit sequence* is an arbitrary combination of one or more digits.

Symbolic Constants

C also provides the facility to define symbolic constants for flexible writing of programs by user. For example, It's bad practice to bury ``magic numbers'' like 300 and

20 in a program; they convey little information to someone who might have to read the program later, and they are hard to change in a systematic way. One way to deal with magic numbers is to give them meaningful names. A #define line defines a *symbolic name or symbolic constant* to be a particular string of characters:

#define *name replacement list*

Thereafter, any occurrence of *name* (not in quotes and not part of another name) will be replaced by the corresponding *replacement tex*t. The *name* has the same form as a variable name: a sequence of letters and digits that begins with a letter. The *replacement text* can be any sequence of characters; it is not limited to numbers.

```
#include <stdio.h>
#define LOWER 0                  /* lower limit of table */
#define UPPER 300                /* upper limit */
#define STEP 20                   /* step size */
                              /* print Fahrenheit-Celsius table */

main()
{
int fahr;
for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

The quantities `LOWER`, `UPPER` and `STEP` are symbolic constants, not variables, so they do not appear in declarations. Symbolic constant names are conventionally written in upper case so they can be readily distinguished from lower case variable names. Notice that there is no semicolon at the end of a `#define` line.

Some more examples are

```
#define MAXLINE 1000

char line[MAXLINE+1];
```

or

```
#define LEAP 1 /* in leap years */

int days[31+28+LEAP+31+30+31+30+31+31+30+31+30+31];
```

2.4 Variables

A *variable* is a named data storage location in your computer's memory. By using a variable's name in your program, you are, in effect, referring to the data stored there.

Unlike constants, the value of variable can be changed during the execution of the program

Rules for using Variable Names

To use variables in your C programs, you must know how to create variable names. In C, variable names must adhere to the following rules:

- The first character of the name must be a letter.
- The underscore is also a legal first character, but its use is not recommended.
- The name can contain letters, digits, and the underscore character (_).No other special characters should be used.
- Case matters (that is, upper- and lowercase letters). Thus, the names count and Count refer to two different variables.
- C keywords can't be used as variable names. A keyword is a word that is part of the C language.
- ANSI standard recognize a length of 31 characters. However, the length should not be normally more than 8 characters.

The following list contains some examples of legal and illegal C variable names:

Variable Name	***Legality***
Avg	Legal
y2x5__fg7h	Legal
annual_pro	Legal
_1990_yr	Legal but not advised
savings#book	Illegal: Contains the illegal character #
double	Illegal: Is a C keyword
9winter	Illegal: First character is a digit

Because C is case-sensitive, the names percent, PERCENT, and Percent would be considered three different variables. C programmers commonly use only lowercase letters in variable names, although this isn't required. Using all-uppercase letters is usually reserved for the names of constants (which are covered later in this chapter). For many compilers, a C variable name can be up to 31 characters long. (It can actually be longer than that, but the compiler looks at only the first 31 characters of the name.) With this flexibility, you can create variable names that reflect the data being stored. For example, a program that calculates loan payments could store the value of the prime interest rate in a variable named interest_rate.

The variable name helps make its usage clear. You could also have created a variable named x or even johnny_carson; it doesn't matter to the C compiler. The use of the variable, however, wouldn't be nearly as clear to someone else looking at the source code. Although it might take a little more time to type descriptive variable names, the improvements in program clarity make it worthwhile.

Many naming conventions are used for variable names created from multiple words. You've seen one style: interest_rate. Using an underscore to separate words in a variable name makes it easy to interpret. The second style is called *camel notatio*n. Instead of using spaces, the first letter of each word is capitalized. Instead of interest_rate, the variable would be named InterestRate. Camel notation is gaining popularity, because it's easier to type a capital letter than an underscore. We use the underscore in this book because it's easier for most people to read. You should decide which style you want to adopt.

DO s and DON’T s

DO use variable names that are descriptive.
DO adopt and stick with a style for naming your variables.
DON'T start your variable names with an underscore unnecessarily.
DON'T name your variables with all capital letters unnecessarily.

Declaration of variable

Before you can use a variable in a C program, it must be declared. A variable declaration tells the compiler the name and type of a variable and optionally initializes the variable to a specific value. If your program attempts to use a variable that hasn't been declared, the compiler generates an error message. A variable declaration has the following form:

General Format *:*	Type name varname; or

Type name var1,var2,…var n;

Remember that, each statement in C should be terminated with semicolon (;).

typename specifies the variable type and must be any one of the keywords in C. The *varname* is the variable name, which must follow the rules mentioned earlier. You can declare multiple variables of the same type on one line by separating the variable names with commas:

```
int size, length, height; /* three integer variables */
```

```
float average, pay; /* two float variables */
```

A variable may also be initialized in its declaration. If the name is followed by an equals sign and an expression, the expression serves as an initializer, as in

```
char esc = '\\';
int i = 0;
int limit = MAXLINE+1;
float eps = 1.0e-5;
```

If the variable in question is not automatic, the initialization is done once only, conceptionally before the program starts executing, and the initializer must be a constant expression. An explicitly initialized automatic variable is initialized each time the function or block it is in is entered; the initializer may be any expression. External and static variables are initialized to zero by default. Automatic variables for which is no explicit initializer have undefined (i.e., garbage) values.

The qualifier const can be applied to the declaration of any variable to specify that its value will not be changed. For an array, the const qualifier says that the elements will not be altered.

```
const double e = 2.71828182845905;
const char msg[] = "warning: ";
```

The const declaration can also be used with array arguments, to indicate that the function does not change that array:

```
int strlen(const char[]);
```

The result is implementation-defined if an attempt is made to change a const.

2.5 Data Type

C language contains numerous *data types.* Storage representations and machine instructions to handle constants differ from machine to machine. The varieties of data types available allow the programmer to select the type appropriate to the needs of the application as well as the machine.

ANSIC supports four classes of data types:

1. Primary (or fundamental) data types
2. User-defined data types
3. Derived data types
4. Empty data set

The primary data types and their extensions are discussed in this section. The user-defined data types are defined in the next section while the derived data types such as arrays, functions, structures and pointers are discussed as and when they are encountered. The empty data set is discussed in the chapter on functions.

All C compilers support four fundamental data types, namely integer (int), character (char), floating point (float), and double-precision floating point (double). Many of them" also offer extended data types such as long int and long double. Various data types and the terminology used to describe them are given in the following figure.

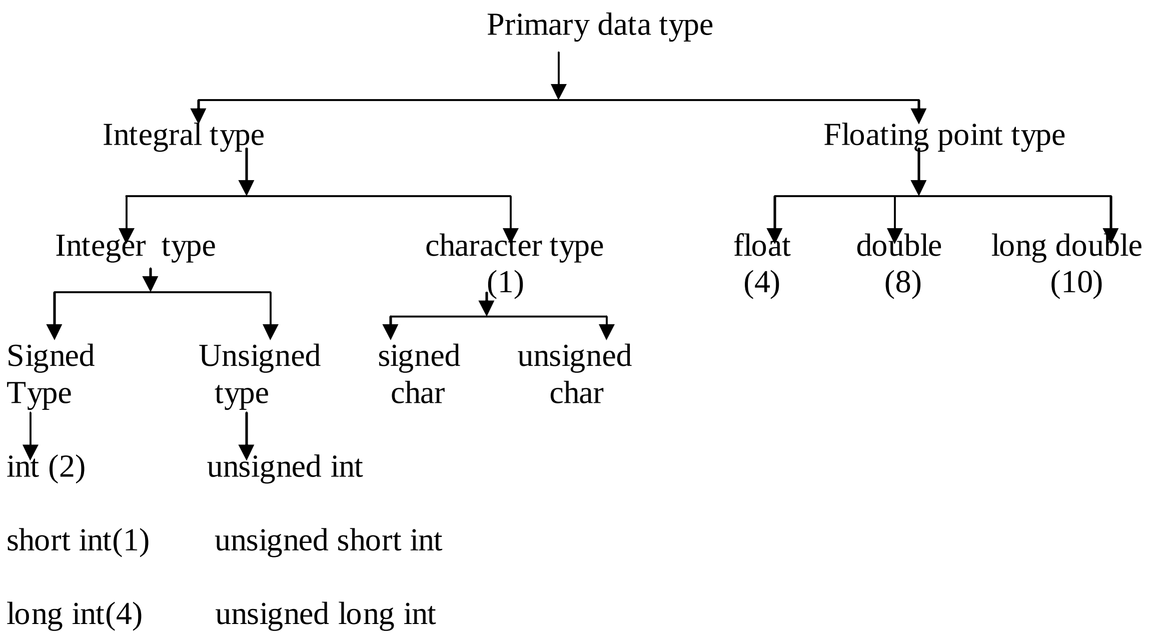


Figure 2.2 Classification of primary data type.

Numbers in parenthesis represent size of data type in bytes.

Even these many data types are available; we will generally use four basic data types. The range of the basic four types are given in the following table. We discuss briefly each one of them in this section.

Char	-128 to 127
Int	-32,768 to 32,767
float	3.4e--38 to 3.4e+38
double	1.7e-308 to 1.7e+308

Integer type

Integers are whole numbers with a range of values supported by a particular machine. Generally, integers occupy one word of storage, and since the word sizes of machines vary (typically, 16 or 31 bits) the size of an integer that can be stored depends on the computer. If we use a 16 bit word length, the size of the integer value is limited to the range -32768 to +32767 (that is, -215 to +215-1). A signed integer uses one bit for sign and 15 bits for the magnitude of the number. Similarly, a 32 bit word length can store an integer ranging from -2, 147,483,648 to 2,147,483,647.

In order to provide some control over the range of numbers and storage space, C has three classes of integer storage, namely short int, int, and long int, in both signed and unsigned forms. For example, short int represents fairly small integer values and requires half the amount of storage as a regular int number uses. Unlike signed integers, unsigned integers use all the bits for the magnitude of the number and are always positive. Therefore, for a 16 bit machine, the range of unsigned integer numbers will be from 0 to 65,535.

We declare long and unsigned integers to increase the range of values. The use of qualifier signed on integers is optional because the default declaration assumes a signed number. The following table shows all the allowed combinations of basic types and qualifiers and their size and range on a 16-bit macJ1ine.

Data Type	**Size**
char *or* signed char	8
unsigned char	8
int or signed int	16
unsigned int	16
short int or signed short int	8
unsigned short int	8
long int or signed long int	32
unsigned long int	32
float	32
double	64
long double	80

Floating point type

Floating point (or real) numbers are stored in 32 bits (on all 16 bit and 32 bit machines), with 6 digits of precision. Floating point numbers are defined in C by the keyword float. When the accuracy provided by a float number is not sufficient, the type double can be used to define the number. A double data type number uses 64 bits giving a precision of 14 digits. These are known as **double precision** numbers. Remember that double type represents the same data type that float represents, but with a greater precision. To extend the precision further, we may use long double which uses 80 bits.

Character type

A single character can be defined as a character(char) type data. Characters are usually stored in 8 bits (one byte) of internal storage. The qualifier signed or unsigned may be explicitly applied to char. While unsigned chars have values between 0 and 255, signed chars have values from-128 to 127.

User defined type

User defined data type in “C” helps the user to increase the readability of the program’s supports a feature known as "type definition" that allows users to represent an identifier that would represent an existing data type. The user-defined data type identifier can later be used to declare variables. It takes the general form: "

```
typedef  type identifier;
```

Where *type* refers to an existing data type and "identifier" refers to the "new” name given to the existing data type. The existing data type may belong to any class of type, including the user-defined ones. Remember that the new type is 'new' only in name, but not the data type. typedef cannot create a new type. Some examples of type definition are:

```
typedef  int number;
typedef float avg;
```

In the above example, number and avg are alias of data type int and float. Once alias names are created then we can declare the variables using alias names.

Example 2.5.

```
number   odd,even;
avg  fisrt, second;
```

Here, odd and even are declared as integer type and first and second are declared as floating point type. The main advantage of typedef is that we can create meaningful data type names for increasing the readability of the program.

Another user-defined data type provided by ANSI standard is enumerated data type, in which, user can create his own data type with predefined values. It is defined as follows:

```
enum identifier {value1, value2, ….., value n};
```

The "identifier" is a user-defined enumerated data type which can be used to declare variables that can have one of the values enclosed within the braces (known as *enumeration constants).* After this definition, we can declare variables to be of this 'new' type as below:

enum identifier v1, v2, ... vn;

The enumerated variables vI, v2, ... vn can only have one of the values *value1,value2, ... value n.* The assignments of the following types are valid:

```
V2 = value1;
V4 = value3;
```

For example

```
enum value {10,20,30,40,50};
```

enum value number1, number2;

for example 2.6, the following program is used to find a given number is prime or not.

Example 2.6

```
#include<stdio.h>
#include<conio.h>
main()
{
int I,c=0;
enum value {10,20,30,40,50};
enum value number1;
number1=20;
for (i=1;i<=number1;i++)
      {
            if (number1%i != 0)
                  c=1;
      }

if(c>=2)

        printf("number1 is prime");
else

          printf(number1 is not a prime");

}
```

The compiler automatically assigns integer digits beginning with 0 to all the enumeration constants. That is, the enumeration constant *value 1* is assigned 0, *value2* is assigned I, and so on. However, the automatic assignments can be overridden by assigning values explicitly to the enumeration constants. For example:

```
enum day {Sunday, Monday,…Saturday};
enum day {Monday = 1, Tuesday, ...};
```

Here, the constant Monday is assigned the value of 1. The remaining constants are assigned values that increase successively by 1.

Derived data type and empty data set

Arrays structures, pointers and functions will comes under derived data type, which we will discuss in the next chapters. Empty data set will come along with the chapter functions.

2.6 Type Conversion

Two basic classifications of conversion methods supported by "C" are,

(i). Implicit conversion or Automatic conversion
(ii). Explicit Conversion.

Implicit Conversion

When an operator has operands of different types, they are converted to a common type according to a small number of rules. In general, the only automatic conversions are those that convert a ``narrower'' operand into a ``wider'' one without losing information, such as converting an integer into floating point in an expression like `f + i`. Expressions that don't make sense, like using a `float` as a subscript, are disallowed. Expressions that might lose information, like assigning a longer integer type to a shorter, or a floating-point type to an integer, may draw a warning, but they are not illegal. A `char` is just a small integer, so `char`s may be freely used in arithmetic expressions. This permits considerable flexibility in certain kinds of character transformations. One is exemplified by this naive implementation of the function `atoi`, which converts a string of digits into its numeric equivalent.

Example 2.7

```
/* atoi: convert s to integer */
int atoi(char s[])
{
int i, n;
n = 0;
for (i = 0; s[i] >= '0' &s[i] <= '9'; ++i)
n = 10 * n + (s[i] - '0');
return n;
}
```

For example, the expression

```
s[i] - '0'
```

gives the numeric value of the character stored in `s[i]`, because the values of `'0'`, `'1'`, etc., form a contiguous increasing sequence.

Another example of `char` to `int` conversion is the function `lower`, which maps a single character to lower case *for the ASCII character se*t. If the character is not an upper case letter, `lower` returns it unchanged.

Example 2.8

```
/* lower: convert c to lower case; ASCII only */
int lower(int c)
{
if (c >= 'A' &c <= 'Z')
return c + 'a' - 'A';
else
return c;
}
```

This works for ASCII because corresponding upper case and lower case letters are a fixed distance apart as numeric values and each alphabet is contiguous -- there is nothing but letters between `A` and Z. This latter observation is not true of the EBCDIC character set, however, so this code would convert more than just letters in EBCDIC.

The standard header `<ctype.h>` defines a family of functions that provide tests and conversions that are independent of character set. For example, the function `tolower` is a portable replacement for the function `lower` shown above. Similarly, the test

```
c >= '0' &c <= '9'
```

can be replaced by,

```
isdigit(c)
```

We will use the `<ctype.h>` functions from now on.

There is one subtle point about the conversion of characters to integers. The language does not specify whether variables of type `char` are signed or unsigned quantities. When a `char` is converted to an `int`, can it ever produce a negative integer? The answer varies from machine to machine, reflecting differences in architecture. On some machines a `char` whose leftmost bit is 1 will be converted to a negative integer (``sign extension''). On others, a `char` is promoted to an int by adding zeros at the left end, and thus is always positive.

The definition of C guarantees that any character in the machine's standard printing character set will never be negative, so these characters will always be positive quantities in expressions. But arbitrary bit patterns stored in character variables may appear to be negative on some machines, yet positive on others. For portability, specify `signed` or `unsigned` if non-character data is to be stored in `char` variables. Relational expressions like `i > j` and logical expressions connected by `&&` and `||` are defined to have value 1 if true, and 0 if false. Thus the assignment

```
d = c >= '0' &c <= '9'
```

sets `d` to 1 if `c` is a digit, and 0 if not. However, functions like `isdigit` may return any non-zero value for true. In the test part of `if`, `while`, `for`, etc., ``true'' just means ``non-zero'', so this makes no difference. Implicit arithmetic conversions work much as expected. In general, if an operator like `+` or `*` that takes two operands (a binary operator) has operands of different types, the ``lower'' type is *promoted* to the ``higher'' type before the operation proceeds. The result is of the integer type. Given below is the sequence of rules that are applied while evaluating expressions. All short and char are automatically converted to int; then,

(1) if one of the operands is long double, the other will be converted to long double and the result will be long double;

(2) else, if one of the operands is double, the other will be converted to double and the result will be double;

(3) else, if one of the operands is float, the other will converted to float arid the result will be float;

(4) else, if one of the operands is unsigned long int, the other will be converted to unsigned long int and the result will be unsigned long int;

(5) else, if one of the operands is long int and the other is unsigned int, then:

  (a) if unsigned int can be converted to long int, the unsigned int operand will be converted as such and the result will be long int;
  (b) else, both operands will be converted to unsigned long int and the result will be unsigned long int;

(6) else, if one of the operands is long int, the other will be converted to long int and the result will be long int;

(7) else, if one of the operands is unsigned int, the other will be converted to unsigned int and the result will be unsigned int.

Notice that `float`s in an expression are not automatically converted to `double`; this is a change from the original definition. In general, mathematical functions like those in `<math.h>` will use double precision. The main reason for using `float` is to save storage in large arrays, or, less often, to save time on machines where double-precision arithmetic is particularly expensive.

Conversion rules are more complicated when `unsigned` operands are involved. The problem is that comparisons between signed and unsigned values are machine-dependent, because they depend on the sizes of the various integer types. For example, suppose that `int` is 16 bits and `long` is 32 bits. Then `-1L < 1U`, because `1U`, which is an `unsigned int`, is promoted to a `signed long`. But `-1L > 1UL` because `-1L` is promoted to `unsigned long` and thus appears to be a large positive number. Conversions take place across assignments; the value of the right side is converted to the type of the left, which is the type of the result. A character is converted to an integer, either by sign extension or not, as described above. Longer integers are converted to shorter ones or to `char`s by dropping the excess high-order bits. Thus in

```
int i;
char c;
i = c;
c = i;
```

the value of `c` is unchanged. This is true whether or not sign extension is involved. Reversing the order of assignments might lose information, however. If `x` is `float` and `i` is `int`, then `x = i` and `i = x` both cause conversions; `float` to `int` causes truncation of any fractional part. When a `double` is converted to `float`, whether the value is rounded or truncated is implementation dependent. Since an argument of a function call is an expression, type conversion also takes place when arguments are passed to functions. In the absence of a function prototype, `char` and `short` become int, and `float` becomes `double`. This is why we have declared function arguments to be `int` and `double` even when the function is called with `char` and `float`.

Casting a value

Explicit type conversions can also be possible in any expression, with a unary operator called a `cast`. In the construction

(type name) expression

the *expression* is converted to the named type by the conversion rules above. The precise meaning of a cast is as if the *expression* were assigned to a variable of the specified type, which is then used in place of the whole construction. For example, the library routine `sqrt` expects a `double` argument, and will produce nonsense if inadvertently handled something else. (`sqrt` is declared in `<math.h>`.) So if `n` is an integer, we can use `sqrt((double) n)` to convert the value of `n` to `double` before passing it to `sqrt`. Note that the cast produces the *value* of `n` in the proper type; `n` itself is not altered. The cast operator has the same high precedence as other unary operators, as summarized in the table at the end of this chapter. If arguments are declared by a function prototype, as the normally should be, the declaration causes automatic coercion of any arguments when the function is called. Thus, given a function prototype for `sqrt`:

```
double sqrt(double)
```

the call, `root2 = sqrt(2)`

coerces the integer 2 into the double value 2.0 without any need for a cast. The standard library includes a portable implementation of a pseudo-random number generator and a function for initializing the seed; the former illustrates a cast:

```
example 2.9
unsigned long int next = 1;

/* rand: return pseudo-random integer on 0..32767 */

int rand(void)
{

     next = next * 1103515245 + 12345;
     return (unsigned int)(next/65536) % 32768;

}

/* srand: set seed for rand() */

void srand(unsigned int seed)
{
    next = seed;
}
```

Examples of casting are given bellow.

x = (Int) 9.5	9.5 is converted to integer by truncation. Result is 9.
a = (Int) 7.11/ (Int) 5.5	Evaluated as 5/7 and the result will be 1 .
b = (double)total / subj	Division is done in floating point mode.
y = (int) (x+y)	The result of x+y is converted to integer.
z = (Int) x+y	x is converted to integer and then added to y.
p = cos ((double) a)	Converts 'a' to double before using it.

For example 2.10

```
/* program for type casting*/
#include<stdio.h>
main( )
{
float sum; int n;
sum = 0;
for( n = 1;n<=10;++n)
{
   sum = sum + 1/(float)n;

   printf ("% 2d %6.4f\n", n, sum);
}
}
```

Output

```
1 1.0000
2 1 .5000
3 1 .8333
4 2.0833
5 2.2833
6 2.4500
7 2.5929
8 2.7179
9 2.8290
10 2.9290
```

Casting can be used to round-off a given value. Consider the following statement:

```
x = (int) (y+O.5);
```

If y is 27.6, y+O.5 is 28.1 and on casting, the result becomes 28, the value that is assigned to x. Of course, the expression, being cast is not changed.

2.7 Let Us Sum-up

In this lesson, we discussed about

- Importance of ‘C’ programming and its applications
- The simplest way of implementing ‘C’ programs
- Basic elements used in ‘C’ like characterset, variables, declaration and Initialization, rules for using variables etc,.
- Various classification of constant and implementing symbolic constants
- Rich set of data types and type conversion.

2.8 Points for discussion

(i). Can we use keywords as variables?
(ii). What will be the maximum size of variable?
(iii). Why we need type casting?
(iv). Differentiate signed and unsigned type.

2.9 Check Your Progress

What is meant by character set in ‘C’?

A character set denotes any letter, digit or any other sign used to represent the language. Ex. Numbers(0-9), Letters (a to z) both upper case and lower case, Special characters..

Define Constants.

Constants are fixed quantities in “c” which can not be changed during the execution.

2.10. Lesson-end Activities

1. Can we use keywords as variable names?
2. Why some words are referred as key words in “c”?
3. Is it possible to treat integer data as float type?

2.11 References

1.Peter Aitken, Teach Yourself C in 21 Days, Fourth Edition, Sams Publisher.
2.Brian W. Kernighan and Dennis M. Ritchie, The C programming Language, Prentice-Hall in 1988
3.E.Balagursamy, Programming in Ansi C, TATA MCGraw Hill
4. http://publications.gbdirect.co.uk/c_book
5. http://www.cs.utk.edu

LESSON – 3 : Operators, Expressions and I/O statements

CONTENTS

3.0 Aims and Objectives

The main aim of this lesson is to make reader to understand about various operators supported by “C” programming language. This lesson will also motivate the programmer to implement programs using formatted and unformatted input and output statements.

3.1 Introduction

This lesson will clearly explain about the list of operators used in “C” programming and the way of reading and displaying the information. This lesson also covers formatted and unformatted input / output statement, so that the reader can increase readability of the program.

3.2 Operators and Expressions

An operator is a symbol which helps the user to command the computer to do a certain mathematical or logical manipulations. Operators are used in C language program to operate on data and variables. C supports rich set of operators. Variables and constants can be used in conjunction with C operators to create more complex expressions. The following table presents the set of C operators.

Operator	**Example**	**Description/Meaning**
()	`f()`	Function call
[]	`a[10]`	Array reference
->	`s->a`	Structure and union member selection
.	`s.a`	Structure and union member selection
+ [unary]	`+a`	Value of a
- [unary]	`-a`	Negative of a
* [unary]	`*a`	Reference to object at address a
& [unary]	`&a`	Address of a
~	`~a`	One's complement of a
++ [prefix]	`++a`	The value of a after increment
++ [postfix]	`a++`	The value of a before increment
- - [prefix]	`-a`	The value of a after decrement
- - [postfix]	`a-`	The value of a before decrement
`sizeof`	`sizeof (t1)`	Size in bytes of object with type t1
`sizeof`	`sizeof e`	Size in bytes of object having the type of ‘e’
+ [binary]	`a + b`	a plus b
- [binary]	`a – b`	a minus b
* [binary]	`a * b`	a times b
/	`a / b`	a divided by b %
%	`a % b`	Remainder of a/b
>>	`a >> b`	a, right-shifted b bits
<<	`a << b`	a, left-shifted b bits

<	a > b	1 if a < b; 0 otherwise
>	a < b	1 if a > b; 0 otherwise
<=	a<= b	1 if a <= b; 0 otherwise
>=	a>= b	1 if a > =b; 0 otherwise
==	a == b	1 if a == b; 0 otherwise
!=	a != b	1 if a != b; 0 otherwise

……………….

……………….

Generally the C operators are classified in to the following categories.

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Increment and decrement operators
6. Conditional operators
7. Bit wise operators
8. Other operators

3.2.1 Arithmetic Operators

C supports five basic arithmetic operators; +, -, *, / and %. All operators work the same way as they do in other languages. These can operate on any built-in data type allowed in C. The unary minus operator, in effect, multiplies its single operand by -1. Therefore, a number preceded by a minus sign changes its sign.

Operation	**Operator**	**Comment**	**Value of Sum before**	**Value of sum after**
Addition	+	sum = sum + 3;	4	7
Subtraction	-	sum = sum -2;	4	2
Multiply	*	sum = sum * 3;	4	12
Divide	/	sum = sum / 2;	4	2
Modulus	%	sum = sum % 3;	4	1

The following code fragment adds the variables *loop* and *count* together, leaving the result in the variable *sum*

```
sum = loop + count;
```

If the modulus **%** sign is needed to be displayed as part of a text string, use two, ie %%

Example 3.1

```
#include <stdio.h>

main()
```

```
{
        int sum = 50;
        float modulus;
        modulus = sum % 10;
        printf("The %% of %d by 10 is %f\n",sum, modulus);
}
```

```
        Sample Program Output

        The % of 50 by 10 is 0.000000
```

Example 3.2

```
#include //include header file stdio.h
main() //tell the compiler the start of the program
{
int numb1, num2, sum, sub, mul, div, mod; //declaration of variables
scanf (“%d %d”, &num1, &num2); //inputs the operands

sum = num1+num2; //addition of numbers and storing in sum.
printf(“\n Thu sum is = %d”, sum); //display the output

sub = num1-num2; //subtraction of numbers and storing in sub.
printf(“\n Thu difference is = %d”, sub); //display the output

mul = num1*num2; //multiplication of numbers and storing in mul.
printf(“\n Thu product is = %d”, mul); //display the output

div = num1/num2; //division of numbers and storing in div.
printf(“\n Thu division is = %d”, div); //display the output

mod = num1%num2; //modulus of numbers and storing in mod.
printf(“\n Thu modulus is = %d”, mod); //display the output
}
```

Integer Arithmetic

If both operands in a single arithmetic expression, such as x+y, are integers then the expression is called an *integer expression,* and the operation is called *integer arithmetic.* Integer arithmetic always yields an integer value. The largest integer value depends on the machine, as pointed out earlier. In the above examples, if a and b are integers, then for x = 13 and y = 2 we have the following results:

$x - y = 11$
$x + y = 15$
$x * y = 26$

x / y = 6 (decimal part truncated)
x % y = 1 (remainder of division)

During integer division, if both the operands are of the same sign, the result is truncated towards zero. If one of them is negative, the direction of truncation is implementation dependent. That is,

6 / 7 = 0 and – 6 / - 7 = 0

but -6 / 7 may be zero or -1. (Machine dependent)

Similarly, during modulo division, the sign of the result is always the sign of the first operand (the dividend.) That is,

-14 % 3 = -2
-14 % -3 = -2
14 % -3 = 2

The following program shows the use of integer arithmetic to convert a given number of days into months and days.

The variables months and days are declared as integers. Therefore, the statement

months = days/30;

truncates the decimal part and assigns the integer part to months. Similarly, the statement

days = days%30;

assigns the remainder part of the division to days. Thus the given number of days is converted into an equivalent number of months and days and the result is printed as shown in the output.

Example 3.3

```
#include <stdio.h>
main()
{
        int months, days;
        printf("Enter days\ n");
        scanf("%d", &days);
        months = days / 30i days = days % 30;
        printf("Months = %d Days = %d", months, days);
}
```

Output Enter days 265

```
Months = 8        Days = 25
Enter days 364
```

```
Months = 12        Days = 4
Enter days 45
Months = 1        Days = 15
```

Real arithmetic

If both operands in a single arithmetic expression, such as x + y, are integers then the expression is called an *real expression,* and the operation is called *real arithmetic.* A real operand may assume values either in decimal or exponential notation. Since floating point values are rounded to the number of significant digits permissible, the final value is an approximation of the correct result. If x, y, and z are floats, then we will have:

```
x   =   6.0 / 7.0         = 0.857143
Y   = 1.0 / 3.0           = 0.333333
z   = -2.0  / 3.0         = -0.666667
```

Mixed-mode Arithmetic

When one of the operands is real and the other is integer, the expression is called a *mixed mode arithmetic* expression. If either operand is of the real type, then only the real operation is performed and the result is always a real number. Thus

C = 5 / 2.0 gives 2.5 where as 5/2 gives 2 only.

3.2.2 Relational operators

Relational operators are used to compare two quantities. Relational operators have lower precedence than arithmetic operators, so an expression like `i < lim-1` is taken as `i < (lim-1)`, as would be expected. C supports following relational operators

Operator	**Meaning**	**Example**
>	greater than	8 > 4
>=	greater than or equal to	mark >= score
<	less than	height < 7
<=	less than or equal to	height <= input
==	equal to	score == mark
!=	not equal to	9 != 4

Here are important rules for using relational operators:

- Each of these six relational operators takes two operands. These two operands must both be *arithmetic* or both be strings. For arithmetic operands, if they are of different types (*i.e.*, one **INTEGER** and the other **REAL**), the **INTEGER operand will be converted to REAL.**

- **The outcome of a comparison is a LOGICAL value. For example, 5 /= 3 is .TRUE. and 7 + 3 >= 20 is .FALSE.**

- **All relational operators have equal priority and are lower than those of arithmetic operators.**

Type	*Operator*	*Associativity*
	**	right to left
Arithmetic	* /	left to right
operators	+ -	left to right
Relational	< <= > >= == /=	none

- This means that a relational operator can be evaluated only if its two operands have been evaluated. For example, in

```
a + b /= c*c + d*d
```

expressions a+b and c*c + d*d are evaluated before the relational operator /= is evaluated.

- If you are not comfortable in writing long relational expressions, use parenthesis. Thus,

```
3.0*SQRT(Total)/(Account + Sum) - Sum*Sum >= Total*GNP - b*b
```

is equivalent to the following:

```
(3.0*SQRT(Total)/(Account + Sum) - Sum*Sum) >= (Total*GNP - b*b)
```

- Although a < b < c is legal in mathematics, you cannot write comparisons this way in Fortran. The meaning of this expression is a < b and b < c.

For example

3**2 + 4**2 == 5**2 is .TRUE.

If the values of REAL variables a, b and c are 1.0, 2.0 and 4.0, respectively, then b*b - 4.0*a*c >= 0.0 is equivalent to 2.0*2.0 - 4.0*1.0*4.0 >= 0.0, which evaluates to -12.0 >= 0.0. Thus, the result is .FALSE.

If REAL variables x and y have values 3.0 and 7.0, and INTEGER variables p and q have values 6 and 2, what is the result of x*x - y*y + 2.0*x*y /= p*q + p**3 - q**3?

x*x - y*y + 2.0*x*y /= p*q + p**3 - q**3
--> 3.0*3.0 - 7.0*7.0 + 2.0*3.0*7.0 /= 6*2 + 6**3 - 2**3
--> [3.0*3.0] - 7.0*7.0 + 2.0*3.0*7.0 /= 6*2 + 6**3 - 2**3
--> 9.0 - 7.0*7.0 + 2.0*3.0*7.0 /= 6*2 + 6**3 - 2**3
--> 9.0 - [7.0*7.0] + 2.0*3.0*7.0 /= 6*2 + 6**3 - 2**3

```
--> 9.0 - 49.0 + 2.0*3.0*7.0 /= 6*2 + 6**3 - 2**3
--> [9.0 - 49.0] + 2.0*3.0*7.0 /= 6*2 + 6**3 - 2**3
--> -40.0 + 2.0*3.0*7.0 /= 6*2 + 6**3 - 2**3
--> -40.0 + [2.0*3.0]*7.0 /= 6*2 + 6**3 - 2**3
--> -40.0 + 6.0*7.0 /= 6*2 + 6**3 - 2**3
--> -40.0 + [6.0*7.0] /= 6*2 + 6**3 - 2**3
--> -40.0 + 42.0 /= 6*2 + 6**3 - 2**3
--> [-40.0 + 42.0] /= 6*2 + 6**3 - 2**3
--> 2.0 /= 6*2 + 6**3 - 2**3
--> 2.0 /= [6*2] + 6**3 - 2**3
--> 2.0 /= 12 + 6**3 - 2**3
--> 2.0 /= 12 + [6**3] - 2**3
--> 2.0 /= 12 + 216 - 2**3
--> 2.0 /= [12 + 216] - 2**3
--> 2.0 /= 228 - 2**3
--> 2.0 /= 228 - [2**3]
--> 2.0 /= 228 - 8
--> 2.0 /= [228 - 8]
--> 2.0 /= [220]
--> 2.0 /= 220.0
--> .TRUE.
```

In the above, please note the left-to-right evaluation order and the type conversion

Comparing CHARACTER Strings

Characters are encoded. Different standards (*e.g.* BCD, EBCDIC and ASCII) may have different encoding schemes. To write a program that can run on all different kind of computers and get the same comparison results, one can only assume the following ordering sequences:

A < B < C < D < E < F < G < H < I < J < K < L < M
< N < O < P < Q < R < S < T < U < V < W < X < Y < Z

a < b < c < d < e < f < g < h < i < j < k < l < m
< n < o < p < q < r < s < t < u < v < w < x < y < z

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

If you compare characters in *different* sequences such as 'A' < 'a' and '2' >= 'N', you are asking for trouble since different encoding methods may produce different answers. Moreover, do not assume there exists a specific order among upper and lower case letters, digits, and special symbols. Thus, '+' <= 'A', '*' >= '%', 'u' > '$', and '8' >= '?' make no sense. However, you can always compare if two characters are equal or not equal. Hence, '*' /= '9', 'a' == 'B' and '8' == 'b' are perfectly fine.

Here is the method for comparing two strings:

1) The comparison always starts at the first character and proceeds from left to right.

2) If the two corresponding characters are equal, then proceed to the next pair of characters.

3) Otherwise, the string containing the smaller character is considered to be the smaller one. And, the comparison halts.

4) During the process comparison,

    i) if both strings have consumed all of their characters, they are equal since all of their corresponding characters are equal.

    ii) otherwise, the shorter string is considered to be the smaller one.

For example

Compare "abcdef" and "abcefg"

```
 b c d e f
= = = <
a b c e f g
```

The first three characters of both strings are equal. Since 'd' of the first string is smaller than 'e' of the second, "abcdef" < "abcefg" holds.

Compare "01357" and "013579"

```
0 1 3 5 7
= = = = =
0 1 3 5 7 9
```

Since all compared characters are equal and the first string is shorter, "01357" < "013579" holds.

What is the result of "DOG" < "FOX"?

```
D O G
<
F O X
```

The first character (*i.e.*, 'D' < 'F') determines the outcome. That is, "DOG" < "FOX" yields .TRUE.

The priority of all six relational operators is lower than the string concatenation operator //. Therefore, if a relational expression involves //, then all string concatenations must be carried out before evaluating the comparison operator. Here is an example:

```
"abcde" // "xyz" < "abc" // ("dex" // "ijk")
    --> ["abcde" // "xyz"] < "abc" // ("dex" // "ijk")
    --> "abcdexyz" < "abc" // ("dex" // "ijk")
    --> "abcdexyz" < "abc" // (["dex" // "ijk"])
    --> "abcdexyz" < "abc" // ("dexijk")
    --> "abcdexyz" < "abc" // "dexijk"
    --> "abcdexyz" < ["abc" // "dexijk"]
    --> "abcdexyz" < "abcdexijk"
    --> .FALSE.
```

Remember that, if an expression is having relational operator then the expression is referred to as relational expression.

3.2.3 Logical operators

Logical operators are used to combine more then one relational expression. C supports the following logical operators.

Operator	**Meaning**	**Syntax**
&&	Logical AND.	exp1 && exp2
||	Logical OR.	exp1 || exp2
!	Logical NOT.	!exp1

The logical AND operator (&&) and the logical OR (||) operator evaluate the truth or falsehood of pairs of expressions. The AND operator evaluates to 1 if and only if both expressions are true. The OR operator evaluates to 1 if *either* expression is true. To test whether y is greater than x and less than z, you would write

```
(x < y) && (y < z)
```

The logical negation operator (!) takes only one operand. If the operand is true, the result is false; if the operand is false, the result is true. The operands to the logical operators may be integers or floating-point objects. The expression,

```
1 && -5
```

results in 1 because both operands are nonzero. The same is true of the expression

```
0.5 && -5
```

Logical operators (and the comma and conditional operators) are the only operators for which the order of evaluation of the operands is defined. The compiler must evaluate operands from left to right. Moreover, the compiler is guaranteed not to evaluate an operand if it is unnecessary. For example, in the expression

```
if ((a != 0) && (b/a == 6.0))
```

if a equals 0, the expression (b/a == 6) will not be evaluated. This rule can have unexpected consequences when one of the expressions contains side effects.

Truth Table for C's Logical Operators

In C, true is equivalent to any *nonzero* value, and false is equivalent to 0. The following table shows the logical tables for each operator, along with the numerical equivalent. All of the operators return 1 for true and 0 for false.

Truth table for AND (&&) operator

Input-1	**operator**	**Input -2**	**Output**
Zero	&&	zero	0
Nonzero	&&	zero	0
Zero	&&	nonzero	0
Nonzero	&&	nonzero	1

Truth table for OR (||) operator

Input-1	**operator**	**Input -2**	**Output**
Zero	||	zero	0
Nonzero	||	zero	1
Zero	||	nonzero	1
Nonzero	||	nonzero	1

Truth table for OR (||) operator

Input	**Output**
! zero	1
! one	0

Examples of Expressions Using the Logical Operators

The following table shows a number of examples that use relational and logical operators. The logical NOT operator has a higher precedence than the others. The AND operator has higher precedence than the OR operator. Both the logical AND and OR operators have lower precedence than the relational and arithmetic operators.

Given the following declarations:

```
int j = 0, m = 1, n = -1;
float x = 2.5, y = 0.0;
```

j && m	(j) && (m)	0
j < m && n < m	(j < m) && (n < m)	1
m + n || ! j	(m + n) || (!j)	1
x * 5 && 5 || m / n	((x * 5) && 5) || (m / n)	1
j <= 10 && x >= 1 && m	((j <= 10) && (x >= 1)) && m	1
!x || !n || m+n	((!x) || (!n)) || (m+n)	0
x * y < j + m || n	((x * y) < (j + m)) || n	1
(x > y) + !j || n++	((x > y) + (!j)) || (n++)	1
(j || m) + (x || ++n)	(j || m) + (x || (++n))	2

Side Effects in Logical Expressions

Logical operators (and the conditional and comma operators) are the only operators for which the order of evaluation of the operands is defined. For these operators, operands must be evaluated from left to right. However, the system evaluates only as much of a logical expression as it needs to determine the result. In many cases, this means that the system does not need to evaluate the entire expression. For instance, consider the following expression:

```
if ((a < b) && (c == d))
```

The system begins by evaluating (a < b). If a is not less than b, the system knows that the entire expression is false, so it will not evaluate (c == d). This can cause problems if some of the expressions contain side effects:

```
if ((a < b) && (c == d++))
```

In this case, d is only incremented when a is less than b. This may or may not be what the programmer intended. In general, you should avoid using side effect operators in logical expressions.

Example 3.4

```
/* Program name is "logical_ops_example". This program */
/* shows how logical operators are used. */

#include <stdio.h>
int main(void)
{
int won_lottery, enough_vacation, money_saved;
char answer;
won_lottery = enough_vacation = money_saved = 0;
printf("\nThis program determines whether you can ");
printf("take your next vacation in Europe.\n");
printf("Have you won the lottery? y or n: ");
fflush(stdin);
scanf("%c", &answer);
if (answer == 'y')
won_lottery = 1;

printf("Do you have enough vacation days saved? \y or n: ");
fflush(stdin);
scanf ("%c", &answer);
if (answer == 'y')
enough_vacation = 1;
printf("Have you saved enough money for the trip? \y or n: ");
fflush(stdin);
scanf("%c", &answer);
if (answer == 'y')
money_saved = 1;
printf("\n");
if (won_lottery)
{
printf("Why do you need a program to decide if you");
printf(" can afford a trip to Europe?\n");
} /* end if */
if (won_lottery || (enough_vacation &&money_saved))
printf("Look out Paris!\n");
else if (enough_vacation &&(!money_saved))
printf("You've got the time, but you haven't got \the dollars.\n");
else if (!enough_vacation || (!money_saved))
{
printf("Tough luck. Try saving your money and ");
printf("vacation days next year.\n");
} /* end else/if */
}
```

If you execute this program, you get the following output:

```
This program determines whether you can take your next vacation
in Europe.
Have you won the lottery? y or n: y
Do you have enough vacation days saved? y or n: n
Have you saved enough money for the trip? y or n: n

Why do you need a program to decide if you can afford a trip to
Europe?
Look out Paris!
```

Toi di lang thang lan trong bong toi buot gia, ve dau khi da mat em roi? Ve dau khi bao nhieu mo mong gio da vo tan... Ve dau toi biet di ve dau? http://gaigoibaucat.xlphp.net

3.2.4 Assignment operator

The Assignment Operator evaluates an expression on the right of the expression and substitutes it to the value or variable on the left of the expression.

for example

```
x = a + b
```

Here the value of `a + b` is evaluated and substituted to the variable `x.`

In addition, C has a set of shorthand assignment operators of the form.

```
var oper = exp;
```

Here `var` is a variable, `exp` is an expression and `oper` is a C binary arithmetic operator. The operator `oper =` is known as shorthand assignment operator

for example

```
x + = 1 is same as x = x + 1
```

The commonly used shorthand assignment operators are as follows

Shorthand assignment operators

Statement with simple assignment operator	**Statement with shorthand operator**
a = a + 1	a += 1
a = a – 1	a -= 1
a = a * (n+1)	a *= (n+1)
a = a / (n+1)	a /= (n+1)
a = a % b	a %= b

Example 3.5

```
#define N 100 //creates a variable N with constant value 100
#define A 2 //creates a variable A with constant value 2

main() //start of the program
{
int a; //variable a declaration
a = A; //assigns value 2 to a

while (a < N) //while value of a is less than N
{ //evaluate or do the following
printf("%d \n",a); //print the current value of a
a *= a; //shorthand form of a = a * a
} //end of the loop
} //end of the program
```

Output

```
2
4
16
```

The use of shorthand assignment operators has three advantages:

1. What appears on the left-hand side need not be repeated and therefore it becomes easier to write.

2. The statement is more concise and easier to read.

3. The statement is more efficient.

These advantages may be appreciated if we consider a slightly more involved statement, like

```
value(5*j-2) = value(5*j-2) + delta;
```

With the help .of the + = operator, this can be written as follows:

```
value(5*j-2) += delta;
```

It is easier to read and understand, and is more efficient because the expression 5 * j - 2 is evaluated only once.

3.2.5 Increment and decrement operator

The increment and decrement operators are one of the unary operators which are very useful in C language. They are extensively used in for and while loops. The syntax of the operators is given below

1. ++ variable name
2. variable name++
3. – –variable name
4. variable name– –

The increment operator ++ adds the value 1 to the current value of operand and the decrement operator – – subtracts the value 1 from the current value of operand. `++variable name` and `variable name++` mean the same thing when they form statements independently, they behave differently when they are used in expression on the right hand side of an assignment statement.

Consider the following

```
m = 5;
y = ++m; (prefix)
```

In this case the value of `y` and `m` would be 6

Suppose if we rewrite the above statement as

```
m = 5;
y = m++; (post fix)
```

Then the value of `y` will be 5 and that of `m` will be 6. A prefix operator first adds 1 to the operand and then the result is assigned to the variable on the left. On the other hand, a postfix operator first assigns the value to the variable on the left and then increments the operand.

3.2.6 Conditional or Ternary Operator

The conditional operator consists of 2 symbols the question mark `(?)` and the colon `(:)` The syntax for a ternary operator is as follows

```
exp1 ? exp2 : exp3
```

The ternary operator works as follows

exp1 is evaluated first. If the expression is true then exp2 is evaluated & its value becomes the value of the expression. If exp1 is false, exp3 is evaluated and its value becomes the value of the expression. Note that only one of the expression is evaluated.

For example 3.6

a = 10;
b = 15;
x = (a > b) ? a : b

Here x will be assigned to the value of b. The condition follows that the expression is false therefore b is assigned to x.

```
/* Example 3.7 : to find the maximum value using conditional operator)
#include
void main() //start of the program
{
int i,j,larger; //declaration of variables
printf (“Input 2 integers : ”); //ask the user to input 2 numbers
scanf(“%d %d”,&i, &j); //take the number from standard input and store it
larger = i > j ? i : j; //evaluation using ternary operator
printf(“The largest of two numbers is %d \n”, larger); // print the largest number
} // end of the program
```

Output

```
Input 2 integers : 34 45
The largest of two numbers is 45
```

3.2.7 Bitwise Operators

C has a distinction of supporting special operators known as bitwise operators for manipulation data at bit level. A bitwise operator operates on each bit of data. Those operators are used for testing, complementing or shifting bits to the right on left. Bitwise operators may not be applied to a float or double.

Meaning	**Operator**
Bitwise AND	&
Bitwise OR	|
Bitwise Exclusive	^
Shift left	<<
Shift right	>>

3.2.8 Special Operators

C supports some special operators of interest such as comma operator, size of operator, pointer operators `(& and *)` and member selection operators `(. and ->).` The size of and the comma operators are discussed here. The remaining operators are discussed in forth coming chapters.

The Comma Operator

The comma operator can be used to link related expressions together. A comma-linked list of expressions are evaluated left to right and value of right most expression is the value of the combined expression.

For example the statement

value = (x = 10, y = 5, x + y);

First assigns 10 to x and 5 to y and finally assigns 15 to value. Since comma has the lowest precedence in operators the parenthesis is necessary. Some examples of comma operator are
In for loops:

for (n=1, m=10, n <=m; n++,m++)

In while loops

While (c=getchar(), c != ‘10’)

Exchanging values

t = x, x = y, y = t;

The size of Operator

The operator size of gives the size of the data type or variable in terms of bytes occupied in the memory. The operand may be a variable, a constant or a data type qualifier.

Example

```
m = sizeof (sum);
n = sizeof (long int);
k = sizeof (235L);
```

The size of operator is normally used to determine the lengths of arrays and structures when their sizes are not known to the programmer. It is also used to allocate memory space dynamically to variables during the execution of the program.

Example3.8 Program that employs different kinds of operators. The results of their evaluation are also shown in comparison

```
#include<stdio.h>
main() //start of program
{
int a, b, c, d; //declaration of variables
a = 15; b = 10; c = ++a-b; //assign values to variables
printf (“a = %d, b = %d, c = %d\n”, a,b,c); //print the values
d=b++ + a;
printf (“a = %d, b = %d, d = %d\n, a,b,d);
printf (“a / b = %d\n, a / b);
printf (“a %% b = %d\n, a % b);
printf (“a *= b = %d\n, a *= b);
printf (“%d\n, (c > d) ? 1 : 0 );
printf (“%d\n, (c < d) ? 1 : 0 );
}
```

Notice the way the increment operator ++ works when used in an expression. In the statement c = ++a – b; new value a = 16 is used thus giving value 6 to C. That is ‘a’ is incremented by 1 before using in expression. However in the statement d = b++ + a; The old value b = 10 is used in the expression. Here b is incremented after it is used in the expression.

We can print the character % by placing it immediately after another % character in the control string. This is illustrated by the statement.

```
printf(“a %% b = %d\n”, a%b);
```

This program also illustrates that the expression
c > d ? 1 : 0
Assumes the value 0 when c is less than d and 1 when c is greater than d.

3.3 Formatted input output statements.

3.3.1 Input Statement – Scanf()

The formatted input refers to input data that has been arranged in a particular format. Input values are generally taken by using the scanf function. The scanf function has the general form.

```
Scanf (“control string”, arg1, arg2, arg3 ………….argn);
```

The format field is specified by the control string and the arguments arg1, arg2, …………….argn specifies the addrss of location where address is to be stored.

The control string specifies the field format which includes format specifications and optional number specifying field width and the conversion character % and also blanks, tabs and newlines.

The Blanks tabs and newlines are ignored by compiler. The conversion character % is followed by the type of data that is to be assigned to variable of the assignment. The field width specifier is optional.

The general format for reading a integer number is

```
% x d
```

Here percent sign (%) denotes that a specifier for conversion follows and x is an integer number which specifies the width of the field of the number that is being read. The data type character d indicates that the number should be read in integer mode.

for example

```
scanf (“%3d %4d”, &sum1, &sum2);
```

If the values input are 175 and 1342 here value 175 is assigned to sum1 and 1342 to sum 2. Suppose the input data was follows 1342 and 175.

The number 134 will be assigned to sum1 and sum2 has the value 2 because of %3d the number 1342 will be cut to 134 and the remaining part is assigned to second variable sum2. If floating point numbers are assigned then the decimal or fractional part is skipped by the computer. To read the long integer data type we can use conversion specifier % ld & % hd for short integer.

Input specifications for real number:

Field specifications are not to be use while representing a real number therefore real numbers are specified in a straight forward manner using % f specifier.

The general format of specifying a real number input is
Scanf (% f “, &variable);

for example
Scanf (“%f %f % f”, &a, &b, &c);

With the input data
321.76, 4.321, 678 The values
321.76 is assigned to a , 4.321 to b & 678 to C.

If the number input is a double data type then the format specifier should be % lf instead of %f.

Input specifications for a character.

Single character or strings can be input by using the character specifiers. The general format is

% xc or %xs

Where C and S represents character and string respectively and x represents the field width. The address operator need not be specified while we input strings.

for example

Scanf (“%C %15C”, &ch, nname):

Here suppose the input given is a, Robert then a is assigned to ch and name will be assigned to Robert.

Example 3.9

```
#include <stdio.h>
main()
{
float y;
int x;
puts( "Enter a float, then an int" );
scanf( "%f %d", &y, &x);
printf( "\nYou entered %f and %d ", y, x );
return 0;
}
```

You will want most of your programs to display information on-screen. The two most frequently used ways to do this are with C's library functions printf() and puts().

3.3.2 Output Statement - Printf() Function

The printf() function, part of the standard C library, is perhaps the most versatile way for a program to display data on-screen. You've already seen printf() used in many of the examples in this book. Now you will to see how printf() works.

Printing a text message on-screen is simple. Call the printf() function, passing the desired message enclosed in double quotation marks. For example, to display An error has occurred! on-screen, you write

```
printf("An error has occurred!");
```

In addition to text messages, however, you frequently need to display the value of program variables. This is a little more complicated than displaying only a message. For example, suppose you want to display the value of the numeric variable x on-screen, along with some identifying text. Furthermore, you want the information to start at the beginning of a new line.You could use the printf() function as follows:

```
printf("\nThe value of x is %d", x);
```

The resulting screen display, assuming that the value of x is 12, would be

```
The value of x is 12
```

In this example, two arguments are passed to printf(). The first argument is enclosed in double quotation marks and is called the *format strin*g. The second argument is the name of the variable (x) containing the value to be printed.

The printf() Format Strings

A printf() format string specifies how the output is formatted. Here are the three possible components of a format string:

Literal text is displayed exactly as entered in the format string. In the preceding example, the characters starting with the T (in The) and up to, but not including, the % comprise a literal string.

An *escape sequence* provides special formatting control. An escape sequence consists of a backslash (\) followed by a single character. In the preceding example, \n is an escape sequence. It is called the *newline character,* and it means "move to the start of the next line." Escape sequences are also used to print certain characters. Escape sequences are listed in Table 7.1.

A *conversion specifier* consists of the percent sign (%) followed by a single character. In the example, the conversion specifier is %d. A conversion specifier tells printf() how to interpret the variable(s) being printed. The %d tells printf() to interpret the variable x as a signed decimal integer.

Table 3.3.1. The most frequently used escape sequences.

Sequence	**Meaning**
\a	Bell (alert)
\b	Backspace
\n	Newline
\t	Horizontal tab
\\	Backslash

\?	Question mark
\'	Single quotation

The printf() Escape Sequences

Now let's look at the format string components in more detail. Escape sequences are used to control the location of output by moving the screen cursor. They are also used to print characters that would otherwise have a special meaning to printf(). For example, to print a single backslash character, include a double backslash (\\) in the format string. The first backslash tells printf() that the second backslash is to be interpreted as a literal character, not as the start of an escape sequence. In general, the backslash tells printf() to interpret the next character in a special manner. Here are some examples:

Sequence	*Meaning*
n	The character n
\n	Newline
\"	The double quotation character
"	The start or end of a string

Table 3.3.1 lists C's most commonly used escape sequences.

Using printf() escape sequences.

```
/* Demonstration of frequently used escape sequences */

#include <stdio.h>

#define QUIT 3

int get_menu_choice( void );
void print_report( void );

main()
{
int choice = 0;

while (choice != QUIT)
{
choice = get_menu_choice();

if (choice == 1)
printf("\nBeeping the computer\a\a\a" );
else
{
if (choice == 2)
print_report();
}
}
printf("You chose to quit!\n");

return 0;
}

int get_menu_choice( void )
{
int selection = 0;

do
{
printf( "\n" );
printf( "\n1 - Beep Computer" );
printf( "\n2 - Display Report");
printf( "\n3 - Quit");
printf( "\n" );
printf( "\nEnter a selection:" );

scanf( "%d", &selection );

}while ( selection < 1 || selection > 3 );

return selection;
}

void print_report( void )
{
printf( "\nSAMPLE REPORT" );
printf( "\n\nSequence\tMeaning" );
printf( "\n=========\t=======" );
printf( "\n\\a\t\tbell (alert)" );
printf( "\n\\b\t\tbackspace" );
printf( "\n...\t\t...");
}
```

```
1 - Beep Computer
2 - Display Report
3 - Quit
Enter a selection:1
Beeping the computer
1 - Beep Computer
2 - Display Report
3 - Quit
Enter a selection:2
SAMPLE REPORT
Sequence Meaning
========= =======
\a bell (alert)
\b backspace
... ...
1 - Beep Computer
2 - Display Report
3 - Quit
Enter a selection:3
You chose to quit!
```

In this program, it seems long compared with previous examples, but it offers some additions that are worth noting. The STDIO.H header was included in line 3 because printf() is used in this listing. In line 5, a constant named QUIT is defined. We know that #define makes using the constant QUIT equivalent to using the value 3. Lines 7 and 8 are function prototypes. This program has two functions: get_menu_choice() and print_report(). get_menu_choice() is defined in lines 31 through 49. Lines 37 and 41 contain calls to printf() that print the newline escape sequence. Lines 38, 39, 40, and 42 also use the newline escape character, and they print text. Line 37 could have been eliminated by changing line 38 to the following:

```
printf( "\n\n1 - Beep Computer" );
```

However, leaving line 37 makes the program easier to read.

Looking at the main() function, you see the start of a while loop on line 14. The while loop's statements will keep looping as long as choice is not equal to QUIT. Because QUIT is a constant, you could have replaced it with 3; however, the program wouldn't be as clear. Line 16 gets the variable choice, which is then analyzed in lines 18 through 25 in an if statement. If the user chooses 1, line 19 prints the newline character, a message, and then three beeps. If the user selects 2, line 23 calls the function print_report(). print_report() is defined on lines 51 through 59. This simple function shows the ease of using printf() and the escape sequences to print formatted information to the screen. You've already seen the newline character. Lines 54 through 58 also use the tab escape character, \t. It aligns the columns of the report vertically. Lines 56 and 57 might seem confusing at first, but if you start at the left and work to the right, they make sense. Line 56 prints a newline (\n), then a backslash (\), then the letter a, and then two tabs (\t\t). The line ends with some descriptive text, (bell (alert)). Line 57 follows the same format.

The printf() Conversion Specifiers

The format string must contain one conversion specifier for each printed variable. printf() then displays each variable as directed by its corresponding conversion specifier. You'll learn more about this process on Day 15. For now, be sure to use the conversion specifier that corresponds to the type of variable being printed. Exactly what does this mean? If you're printing a variable that is a signed decimal integer (types int and long), use the %d conversion specifier. For an unsigned decimal integer (types unsigned int and unsigned long), use %u. For a floating-point variable (types float and double), use the %f specifier. The conversion specifiers you need most often are listed in following Table

The most commonly needed conversion specifiers.

Specifier	Meaning	Types Converted
%c	Single character	char
%d	Signed decimal integer	int, short
%ld	Signed long decimal integer	long
%f	Decimal floating-point number	float, double
%s	Character string	char arrays
%u	Unsigned decimal integer	unsigned int, unsigned short
%lu	Unsigned long decimal integer	unsigned c

The literal text of a format specifier is anything that doesn't qualify as either an escape sequence or a conversion specifier. Literal text is simply printed as is, including all spaces. What about printing the values of more than one variable? A single printf() statement can print an unlimited number of variables, but the format string must contain one conversion specifier for each variable. The conversion specifiers are paired with variables in left-to-right order. If you write

```
printf("Rate = %f, amount = %d", rate, amount);
```

the variable rate is paired with the %f specifier, and the variable amount is paired with the %d specifier. The positions of the conversion specifiers in the format string determine the position of the output. If there are more variables passed to printf() than there are conversion specifiers, the unmatched variables aren't printed. If there are more specifiers than variables, the unmatched specifiers print "garbage." You aren't limited to printing the value of variables with printf(). The arguments can be any valid C expression. For example, to print the sum of x and y, you could write

```
z = x + y;
printf("%d", z);
```

You also could write

```
printf("%d", x + y);
```

Any program that uses printf() should include the header file STDIO.H.

The printf() Function

#include <stdio.h>
printf(*format-string*[,*argument*s,...]);

printf() is a function that accepts a series of *argument*s, each applying to a conversion specifier in the given format string. Printf() prints the formatted information to the standard output device, usually the display screen. When using printf(), you need to include the standard input/output header file, STDIO.H.

The *format-string* is required; however, arguments are optional. For each argument, there must be a conversion specifier.

The *format-string* can also contain escape sequences. The following are examples of calls to printf() and their output:

Example 3.10 Input

```
#include <stdio.h>
main()
{
printf("This is an example of something printed!");
return 0;
}
```

Example 3.10 Output

```
This is an example of something printed!
```

Example 3.11 Input

```
printf("This prints a character, %c\na number, %d\na floating point, %f", `z', 123,
456.789 );
```

Example 3.11 Outtput

```
This prints a character, z
a number, 123

a floating point, 456.789
```

printf formatting is controlled by 'format identifiers' which, are shown below in their simplest form.

%d %i	Decimal signed integer.
%o	Octal integer.
%x %X	Hex integer.
%u	Unsigned integer.
%c	Character.
%s	String. See below.
%f	double
%e %E	double.
%g %G	double.
%p	pointer.
%n	Number of characters written by this printf. No argument expected.
%%	%. No argument expected.

These identifiers actually have upto 6 parts as shown in the table below. They MUST be used in the order shown.

%	Flags	Minimum field width	Period	Precision. Maximum field width	Argument type
Required	Optional	Optional	Optional	Optional	Required

%
The % marks the start and therefore is minatory.
Flags
The format identifiers can be altered from their default function by applying the following **flags**:

-	Left justify.
0	Field is padded with 0's instead of blanks.
+	Sign of number always O/P.
blank	Positive values begin with a blank.
#	Various uses:
%#o (Octal)	0 prefix inserted.
%#x (Hex)	0x prefix added to non-zero values.
%#X (Hex)	0X prefix added to non-zero values.
%#e	Always show the decimal point.
%#E	Always show the decimal point.
%#f	Always show the decimal point.
%#g	Always show the decimal point trailing zeros not removed.
%#G	Always show the decimal point trailing zeros not removed.

The flags must follow the %. Where it makes sense, more than one flag can be used.
Here are a few more examples.

```
    printf(" %-10d \n", number);
    printf(" %010d \n", number);
    printf(" %-#10x \n", number);
    printf(" %#x \n", number);
```

Minimum field width.
By default the width of a field will be the minimum required to hold the data. If you want to increase the field width you can use the following syntax.

Example 3.12
```
main()
  {
    int number    =  5;
    char *pointer = "little";

    printf("Here is a number-%4d-and a-%10s-word.\n", number, pointer);
```

```
 }


 /********************************
  *
  *      Program result is:
  *
  *      Here is a number-   5-and a-    little-word.
  *
  ********************************/
```

As you can see, the data is right justified within the field. It can be left justified by using the - flag. A maximum string width can also be specified.
The width can also be given as a variable as shown below.

```
main()
  {
     int number=5;

     printf("---%*d----\n", 6, number);
  }

  /********************************
   *
   *   Program result is:
   *
   *   ----     5---
   *
   ********************************/
```

The * is replaced with the supplied **int** to provide the ability to dynamically specify the field width.

Period

If you wish to specify the precision of an argument, it MUST be prefixed with the period.

Precision

The Precision takes different meanings for the different format types.

Float Precision

    %8.2f

This says you require a total field of 8 characters, within the 8 characters the last 2 will hold the decimal part.

    %.2f

The example above requests the minimum field width and the last two characters are to hold the decimal part.

Character String Maximum field width

The precision within a string format specifies the maximum field width.

%4.8s

Specifies a minimum width of 4 and a maximum width of 8 characters. If the string is greater than 8 characters, it will be cropped down to size.

Precision

As with the 'width' above, the precision does not have to be hard coded, the * symbol can be used and an integer supplied to give its value.

Format Identifiers

The format identifier describes the expected data. The identifier is the character that ends Here is a list of the format identifers as used in 'printf' ,'sprintf' ,'fprintf' and 'scanf'. Except for '%' and 'n', all the identifiers expect to extract an argument from the **printf** parameter list.

All of the parameters should be the value to be inserted. EXCEPT %s, this expects a pointer to be passed.

Example 3.13.

```
main()
    {
      int number=5;
        char *pointer="little";

        printf("Here is a number %d and a %s word.\n", number, pointer);
      }
      /*********************************
       *
       *      Program result is:
       *
       *      Here is a number 5 and a little word.
       *
       *********************************/
```

3.3.3 Puts() and gets() functions

Displaying Messages with puts()

The puts() function can also be used to display text messages on-screen, but it can't display numeric variables. puts() takes a single string as its argument and displays it, automatically adding a newline at the end. For example, the statement

```
puts("Hello, world.");
```

performs the same action as

```
printf("Hello, world.\n");
```

You can include escape sequences (including \n) in a string passed to puts(). They have the same effect as when they are used with printf()

Any program that uses puts() should include the header file STDIO.H. Note that STDIO.H should be included only once in a program.

DO use the puts() function instead of the printf() function whenever you want to print text but don't need to
print any variables.

DON'T try to use conversion specifiers with the puts() statement.

The puts() Function

```
#include <stdio.h>
puts( string );
```

puts() is a function that copies a string to the standard output device, usually the display screen. When you use puts(), include the standard input/output header file (STDIO.H). puts() also appends a newline character to the end of the string that is printed. The format string can contain escape sequences. Table 7.1 lists the most frequently used escape sequences. The following are examples of calls to puts() and their output:

Example 1 Input

```
puts("This is printed with the puts() function!");
```

Example 1 Output

```
This is printed with the puts() function!
```

The gets() function

This function is used to read a line of text

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

```
gets( string );
```

if the input is

“Hello India”,

the gets function reads entire line of text and stored in to respective variable, where as, scanf will read only “Hello” even though the input is “Hello India”.

Simple I/O -- getchar, putchar

The getchar and putchar are the basic I/O library functions in C. getchar fetches one character from the standard input (usually the terminal) each time it is called, and returns that character as the value of the function. When it reaches the end of whatever file it is reading, thereafter it returns the character represented by `\0' (ascii NUL, which has value zero). We will see how to use this very shortly.

Putchar puts one character out on the standard output (usually the terminal) each time it is called. So the program above reads one character and writes it back out. By itself, this isn't very interesting, but observe that if we put a loop around this, and add a test for end of file, we have a complete program for copying one file to another.

```
main( )

{
      char c;
      c = getchar( );
      putchar(c);
}
```

3.4 Let us Sum Up

In this lesson, we discussed about various classification of operators supported by “C”. This will help the programmer to understand the difference between mathematical operators and equivalent operators used in “C”. This lesson also explained about various formatted and unformatted input and output statements available in “C”.

3.5 Points for discussion

Define Ternary operator.

Differentiate scanf() with gets().

How pre increment will differ from post increment?

Explain about mixed-mode expression.

3.6 Check your progress

Differentiate scanf() with gets().

Both functions are used to read the string. The scanf() will read the string until it finds a space (ie, it read single word) where as gets() will read entire line of text. You should also specify the general format for scanf() and gets() with example.

What %5.2f means?

This format code is used to print floating point data, in which, the total size of the data including decimal point is 5 and 2 represent the number of digits will appear after decimal point in the result.

3.7 Lesson-end Activities

1. How many categories of operators we are using in “C”?
2. What is the alternate name of conditional operator?
3. Where we are using Bit wise operators?

3.8 References

1. E.Balagursamy, Programming in Ansi C, TATA MCGraw Hill
2. http://imada.sdu.dk/~svalle/courses/
3. http://www.geocities.com/learnprogramming123/Clesson11
4. http://www.java2s.com/Code/C/

LESSON – 4: Decision Making – Branching statements

CONTENTS

4.0 Aims and objectives

The objective of this lesson is to make you to learn about C Programming - Decision Making, Branching, if Statement, The If else construct, Compound Relational tests, Nested if Statement, The ELSE If Ladder, The Switch Statement and The GOTO statement.

4.1 Introduction: Branching

The C language programs presented until now follows a sequential form of execution of statements. Many times it is required to alter the flow of the sequence of instructions. C language provides statements that can alter the flow of a sequence of instructions. These statements are called control statements. These statements help to jump from one part of the program to another. The control transfer may be conditional or unconditional.

4.2 Decision making with IF Statement

C support four different types of IF statements, as follows

1. Simple IF
2. IF Else
3. Else…IF ladder
4. Nested IF.

4.2.1 Simple IF

The simplest form of the control statement is the If statement. It is very frequently used in decision making and allowing the flow of program execution. The If structure has the following syntax

```
if(condition)
  statement;

if(condition or expression)
  {
        statement1;
        statement2;
        statement3;
        statement4;
        ……….
        ……….
}
```

The statement is any valid C’ language statement and the condition is any valid C’ language expression, frequently logical operators are used in the condition statement. The condition part should not end with a semicolon, since the condition and statement should be put together as a single statement. The command says if the condition is true then performs the following statement or If the condition is fake the computer skips the statement and moves on to the next instruction in the program.

Example 4.1

```
/* Calculate the absolute value of an integer */

# include < stdio.h >            //Include the stdio.h file
void main ( )                   // start of the program
{
int numbers;                    // Declare the variables
printf (“Type a number:”);      // message to the user
scanf (“%d”, & number);         // read the number from standard input
if (number < 0)                 // check whether the number is a negative
number
number = - number;              // If it is negative then convert it into
positive.
Printf (“The absolute value is % d \n”, number); // print the value
}
```

The above program checks the value of the input number to see if it is less than zero. If it is then the following program statement, which negates the value of the number, is executed. If the value of the number is not less than zero, we do not want to negate it then this statement is automatically skipped. The absolute number is then displayed by the program, and program execution ends.

4.2.2 If - Else

The if-else statement is used to express decisions. Formally the syntax is

```
    if (expression)
        statement1
    else
        statement2
```

where the else part is optional. The *expression* is evaluated; if it is true (that is, if *expression* has a non-zero value), *statement1* is executed. If it is false *(expression* is zero) and if there is an else part, *statement2* is executed instead. Since an if tests the numeric value of an expression, certain coding shortcuts are possible. The most obvious is writing

```
    if (expression)
```

instead of

```
    if (expression != 0)
```

Sometimes this is natural and clear; at other times it can be cryptic.

Because the else part of an if-else is optional, there is an ambiguity when an else if omitted from a nested if sequence. This is resolved by associating the else with the closest previous else-less if. For example, in

```
if (n > 0)
if (a > b)
z = a;
else
z = b;
```

the else goes to the inner if, as we have shown by indentation. If that isn't what you want, braces must be used to force the proper association:

```
if (n > 0) {
if (a > b)
z = a;
}
else
z = b;
```

The ambiguity is especially pernicious in situations like this:

```
if (n > 0)
for (i = 0; i < n; i++)
if (s[i] > 0) {
printf("...");
return i;
}
else /* WRONG */
printf("error -- n is negative\n");
```

The indentation shows unequivocally what you want, but the compiler doesn't get the message, and associates the else with the inner if. This kind of bug can be hard to find; it's a good idea to use braces when there are nested ifs. By the way, notice that there is a semicolon after z = a in

```
if (a > b)
z = a;
else
z = b;
```

This is because grammatically, a *statement* follows the if, and an expression statement like ``z = a;'' is always terminated by a semicolon.

Example 4.2

```
// Program find whether a number is negative or positive */

#include < stdio.h >                    //include the stdio.h header file in your program
void main ( )                           // Start of the main
{
int num;                                // declare variable num as integer
printf (“Enter the number”);            //message to the user
scanf (“%d”, &num);                     // read the input number from keyboard
if (num < 0)                            // check whether number is less than zero.
Printf (“The number is negative”)       // If it is less than zero then it is negative.
Else                                    // else statement.
Printf (“The number is positive”);       //If it is more than zero then the given
number is positive.
}
```

In the above program the If statement checks whether the given number is less than 0. If it is less than zero then it is negative therefore the condition becomes true then the statement The number is negative is executed. If the number is not less than zero the If else construct skips the first statement and prints the second statement declaring that the number is positive.

4.2.3 Else..if..ladder

This is another if structure available in “C” for making multiple decisions

```
If (Condition 1 or expression 1)
        Statement block 1;
else if (Condition 2 or expression 2)
                statement block 2;
else if…..
…………
…………
…………
else
        false block;
```

In this structure, condition 1 will be evaluated first. If condition 1 is true then respective statement block will be executed otherwise condition 2 will be evaluated next.

Binary search is an another example for using If.. Else and else…if structure.

Example 4.3

```
/* binsearch: find x in v[0] <= v[1] <= ... <= v[n-1] */
int binsearch(int x, int v[], int n)
{
int low, high, mid;
low = 0;
high = n - 1;
while (low <= high) {
mid = (low+high)/2;
if (x < v[mid])
high = mid + 1;
else if (x > v[mid])
low = mid + 1;
else /* found match */
return mid;
}
return -1; /* no match */
}
```

The fundamental decision is whether x is less than, greater than, or equal to the middle element v[mid] at each step; this is a natural for else-if.

Compound Relational tests:

C language provides the mechanisms necessary to perform compound relational tests. A compound relational test is simple one or more simple relational tests joined together by either the logical AND or the logical OR operators. These operators are represented by the character pairs && // respectively. The compound operators can be used to form Complex expressions in C.

Syntax

a> if (condition1 && condition2 && condition3)
b> if (condition1 // condition2 // condition3)

The syntax in the statement 'a' represents a complex if statement which combines different conditions using the and operator in this case if all the conditions are true only then the whole statement is considered to be true. Even if one condition is false the whole if statement is considered to be false. The statement 'b' uses the logical operator or (//) to group different expression to be checked. In this case if any one of the expression if found to be true the whole expression considered to be true, we can also uses the mixed expressions using logical operators && and || together.

4.2.4 Nested if Statement

The if statement may itself contain another if statement is known as nested if statement.

Syntax:

```
        if (condition1 or expressein1)
        if (condition2 or expression 2)
        statement-1;
        else
        statement-2;
        else
        statement-3;
```

The if statement may be nested as deeply as you need to nest it. One block of code will only be executed if two conditions are true. Condition 1 is tested first and then condition 2 is tested. The second if condition is nested in the first. The second if condition is tested only when the first condition is true else the program flow will skip to the corresponding Else statement.

Example 4.4 print the given numbers along with the largest number.

```
#include < stdio.h >                  //includes the stdio.h file to your program
main ( )                              //start of main function
{
int a,b,c,big;                        //declaration of variables
printf (“Enter three numbers”);        //message to the user
scanf (“%d %d %d”, &a, &b, &c);       //Read variables a,b,c,
if (a>b)                              // check whether a is greater than b if true then
if(a>c)                               // check whether a is greater than c
big = a ;                             // assign a to big
else big = c ;                        // assign c to big
else if (b>c)             // if the condition (a>b) fails check whether b is greater than c
big = b ;                             // assign b to big
else big = c ;                        // assign C to big
printf (“Largest of %d,%d&%d = %d”, a,b,c,big);
}
```

In the above program the statement if (a>c) is nested within the if (a>b). If the first If condition if (a>b). If (a>b) is true only then the second if statement if (a>b) is executed. If the first if condition is executed to be false then the program control shifts to the statement after corresponding else statement.

```
/* Example 4.5 program using compound if else construct */
/* This program determines if a year is a leap year */
# include < stdio.h > //Includes stdio.h file to your program

void main ( ) // start of the program
```

```
{
int year, rem_4, rem_10, rem_400;     // variable declaration
printf (“Enter the year to be tested”);  // message for user
scanf (“t.d”, & year);                 // Read the year from standard input.
rem_4 = year % 4;                     //find the remainder of year – by 4
rem_100 = year % 100;                 //find the remainder of year – by 100
rem_400 = year % 400;                 // find the remainder of year – by 400
if ((rem_4 = = 0 && rem_100! = 0)  //rem_400 = = 0)
                              //apply if condition 5 check whether remainder is zero
printf (“It is a leap year, \n”) ;      // print true condition
else
printf (“No. It is not a leap year. \n”); //print the false condition
}
```

The above program checks whether the given year is a leap year or not. The year is divided by 4100 and 400 respectively and its remainder is collected in the variables rem_4, rem_100 and rem_400. A if condition statements checks whether the remainders are zero. If remainder is zero then the year is a leap year. Here either the year – y 400 is to be zero or both the year – 4 and year – by 100 has to be zero, then the year is a leap year.

4.3 Switch Statement

The switch statement is a construct that is used when many conditions are being tested for. When there are many conditions, it becomes too difficult and complicated to use the if and else if constructs. Nested if/else statements arise when there are multiple alternative paths of execution based on some condition that is being tested for.

The general form of a switch statement is:

```
switch (variable)

{
case expression1:
        do something 1;
        break;
case expression2:
        do something 2;
        break;
....
default:
        do default processing;
}
```

Here's an example, this is a simple calculator that can be used to add, multiply, subtract, and divide. If this program was using if else statements than this program will work fine, but the if/else block is cumbersome. It would be easy, particularly if there were more choices and maybe sub choices involving more if/else's to end up with program that doesn't perform the actions intended. Here's the same program with a switch.

Example 4.6

```
#include <stdio.h>

int main(void)
{
   float numb1 = 0, numb2 = 0;          /* the two numbers to work on */
   int menu = 1;                        /* add or subtract or divide or multiply */
   float total = 0;                     /* the result of the calculation */
   char calType;                        /* what type of calculation */

   printf("Please enter in the first of the two numbers\n\t");
   scanf("%f", &numb1);                 /* READ first number */

   printf("\n\n Please enter the second of the two numbers\n\t");
   scanf("%f", &numb2);                 /* READ second number */

   printf("\n\n What would you like to do?\n\n");    /* WRITE instructions */
   printf("\t1 = add\n");
   printf("\t2 = subtract\n");
   printf("\t3 = multiply\n");
   printf("\t4 = divide\n");

   printf("\n\n Pleas make your selection now:\n\t");
   scanf("%d",&menu);                 /* READ calculation type */

   switch (menu)                /* select the type of calculation */
   {
   case 1: total = numb1 + numb2;
      calType = '+';           /* assign a char to symbolize calculation type */
      break;
   case 2: total = numb1 - numb2;
      calType = '-';
      break;
   case 3: total = numb1 * numb2;
      calType = '*';
      break;
   case 4: total = numb1 / numb2;
      calType = '/';
      break;
   default: printf("Invalid option selected\n");
   }
```

```
    if (menu == 3 && numb2 == 0)                /* cannot divide by 0 */
        printf("\n\n\tYou cannot divide by 0\n\n");

            /* display result to 2 decimal places */
    printf("\n\n************************");
    printf("\n\n\t%.3f %c %.3f = %.2f", numb1, calType, numb2, total);
    printf("\n\n************************\n\n");

    return 0;

}
```

The keyword *default* is executed when none of the conditions being tested for in the switch statement are met or executed. The *break* statement must be used after each condition because if it were not used than all the conditions from the one met will be executed. For example if case 2 was met, and there was no break statement at the end of the case, case 3 and case 4 and even default would all be executed.

Example 4.7

Program to count the occurrences of each digit, white space, and all other characters, using `switch` statement:

```
#include <stdio.h>
main() /* count digits, white space, others */
{
int c, i, nwhite, nother, ndigit[10];
nwhite = nother = 0;
for (i = 0; i < 10; i++)
ndigit[i] = 0;
while ((c = getchar()) != EOF) {
switch (c) {
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
ndigit[c-'0']++;
break;
case ' ':
case '\n':
case '\t':
nwhite++;
break;
default:
nother++;
break;
}
}
printf("digits =");
for (i = 0; i < 10; i++)
printf(" %d", ndigit[i]);
printf(", white space = %d, other = %d\n",
nwhite, nother);
return 0;
}
```

The `break` statement causes an immediate exit from the `switch`. Because cases serve just as labels, after the code for one case is done, execution *falls through* to the next unless you take explicit action to escape. `break` and `return` are the most common ways to leave a `switch`. A `break` statement can also be used to force an immediate exit from `while`, `for`, and `do` loops, as will be discussed later in this chapter.

Example 4.8

```
int number;
/* Estimate a number as none, one, two, several, many */
{       switch(number) {
        case 0 :
                printf("None\n");
                break;
        case 1 :
                printf("One\n");
                break;
        case 2 :
                printf("Two\n");
                break;
        case 3 :
        case 4 :
        case 5 :
                printf("Several\n");
                break;
        default :
                printf("Many\n");
                break;
        }
}
```

Each interesting case is listed with a corresponding action. The break statement prevents any further statements from being executed by leaving the switch. Since case 3 and case 4 have no following break, they continue on allowing the same action for several values of number.

Example 4.9

Consider the example shown below:

```
switch( Grade )
     {
        case 'A' : printf( "Excellent" );
        case 'B' : printf( "Good" );
        case 'C' : printf( "OK" );
        case 'D' : printf( "Mmmmm...." );
        case 'F' : printf( "You must do better than this" );
        default  : printf( "What is your grade anyway?" );
     }
```

Here, if the Grade is 'A' then the output will be

```
        Excellent
        Good
        OK
        Mmmmm....
        You must do better than this
        What is your grade anyway?
```

This is because, in the 'C' **switch** statement, execution continues on into the next case clause if it is not explicitly specified that the execution should exit the **switch** statement. The correct statement would be:

```
switch( Grade )
     {
        case 'A' : printf( "Excellent" );
                   break;

        case 'B' : printf( "Good" );
                   break;

        case 'C' : printf( "OK" );
                   break;

        case 'D' : printf( "Mmmmm...." );
                   break;

        case 'F' : printf( "You must do better than this" );
                   break;

        default  : printf( "What is your grade anyway?" );
                   break;
        }
```

Although the **break** in the **default** clause (or in general, after the last clause) is not necessary, it is good programming practice to put it in anyway.

Example 4.10

```
/***********************************************************************
****
 *
 * Purpose: Program to demonstrate the 'switch/case' structure.
 * Method:  Prog looks at the number of parameters passed to it and
 *          tells the user how many its got.
 * Author:  M J Leslie
 * Date:    09-Apr-94
 *

***********************************************************************
***/

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

  switch (argc)        /* Switch evaluates an expression (argc)  */
  {
                       /* If expression resolves to 1, jump here */
  case 1:
    puts("Only the command was entered.");
    break;             /* break - cases the execution to jump
                         out of the 'switch' block.             */

                       /* If expression resolves to 2, jump here */
  case 2:
    puts("Command plus one parm entered");
    break;

                       /* If expression resolves to 3, jump here */
  case 3:
    puts("Command plus two parm entered");
    break;

                       /* Any other value jumps here.            */
  default:
    printf("Command plus %d parms entered\n", argc-1);
    break;
  }
}
```

4.4 While Statement

The *while* loop is used to execute a block of code as long as some condition is true. If the condition is false from the start the block of code is not executed at al. The while loop tests the condition before it's executed so sometimes the loop may never be executed if initially the condition is not met. Its syntax is as follows.

```
  while (tested condition is satisfied)
{
   block of code
}
```

In all constructs, curly braces should only be used if the construct is to execute more than one line of code. The above program executes only one line of code so it not really necessary (same rules apply to if...else constructs) but you can use it to make the program seem more understandable or readable. Here is a simple example of the use of the while loop. This program counts from 1 to 100.

Example 4.11

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

   int count = 1;
   while (count <= 100)
   {
      printf("%d\n",count);
      count += 1;                  // Notice this statement
   }
   return 0;

}
```

Note that no semi-colons (;) are to be used after the while (condition) statement. These loops are very useful because the condition is tested before execution begins.

Example 4.12

```
/* Demonstrates a simple while statement */

#include <stdio.h>

int count;

int main()
{
/* Print the numbers 1 through 20 */

count = 1;

while (count <=10)
{
printf("%d\n", count);
count++;
}
return 0;
}
```

Output

```
1
```

```
2
3
4
5
6
7
8
9
10
```

The code fragment

```
x = 0;
while (x < 3)
{
  x++;
}
```

first checks whether x is larger than 3, which it is not, so it increments x by 1. It then checks the condition again, and executes again, repeating this process until the variable x has the value 3.

Example 4.13

```
int x = 0;
while (x < 10)
{
printf("\nThe value of x is %d", x );
x++;
}
```

Example 4.14

```
/* get numbers until you get one greater than 99 */
int nbr=0;
while (nbr <= 99)
scanf("%d", &nbr );
```

Example 4.15

```
/* Lets user enter up to 10 integer values */
/* Values are stored in an array named value. If 99 is */
/* entered, the loop stops */
int value[10];
int ctr = 0;
int nbr;
```

```
while (ctr < 10 && nbr != 99)
{
puts("Enter a number, 99 to quit ");
scanf("%d", &nbr);
value[ctr] = nbr;
ctr++;
}
```

Nesting while Statements

Like other statements, while statements can also be nested. If statement block of one While statement is having another While Statement then the structure is referred as nested structure.

```
While (i<=m)
{
  ----
  -----
    While (j<=n)

      {
              While(k<=t)

              {
                   ------------
                   -----------
              }
     }
 }
```

4.5 do ....while Structure

The do loop also executes a block of code as long as a condition is satisfied. The difference between a "do ...while" loop and a "while {} " loop is that the while loop tests its condition before execution of the contents of the loop begins; the "do" loop tests its condition after it's been executed at least once. As noted above, if the test condition is false as the while loop is entered the block of code is never executed. Since the condition is tested at the bottom of a do loop, its block of code is always executed at least once.

Some people don't like these loops because it is always executed at least once. When i ask them "so what?", they normally reply that the loop executes even if the data is incorrect. Basically because the loop is always executed, it will execute no matter what value or type of data is supposed to be required. The "do ....while" loops syntax is as follows

```
do
{
    block of code
} while (condition is satisfied);
```

Note that a semi-colon (;) must be used at the end of the do ...while loop. This semi-colon is needed because it instructs whether the while (condition) statement is the beginning of a while loop or the end of a do ...while loop. Here is an example of the use of a do loop.

Example 4.16

```
include <stdio.h>

int main(void)
{

int value, r_digit;
printf("Enter a number to be reversed.\n");
scanf("%d", &value);

do
{
   r_digit = value % 10;
   printf("%d", r_digit);
   value = value / 10;
} while (value != 0);

printf("\n");
return 0;

}
```

```
Example 4.17 /* program for converting upper to lower and lower to
upper characters in the input string */
```

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

int main(void)
{
  char ch;

  printf("Enter some text (type a period to quit).\n");
  do {
    ch = getchar();
    if(islower(ch))
      ch = toupper(ch);
    else
      ch = tolower(ch);

    putchar(ch);
  } while (ch != '.');

  return 0;
}
```

```
Input        : WELcome

Output       : welCOME
```

Example 4.18 /* Program for reversing a number */

```
#include <stdio.h>
void main() {
   int number = 123; /* The number to be reversed */
   int reversedNumber = 0; /* The reversed number       */
   int temp = 0;   /* Working storage           */

   temp = number; /* Copy to working storage  */

   /* Reverse the number stored in temp */
   do
   {
     reversedNumber = 10 * reversedNumber + temp % 10; /* Add the right
                                                        most digit    */
     temp = temp/10;                /* Remove the rightmost digit */
   } while (temp);                  /* Continue while temp>0      */

   printf ("\nThe number %d reversed is  %d rebmun ehT\n",
                                            number, reversedNumber);
}
Input        : 1234
Output       : 4321
```

```
Example 4.19 /* program for multiple selection */
#include <stdio.h>

int main(void) {
    char ch;

    do {
        printf("Load, Save, Edit, Quit?\n");
        do {
            printf("Enter your selection: ");
            ch = getchar();
        } while(ch!='L' && ch!='S' && ch!='E' && ch!='Q');

        if(ch == 'Q') {
            printf("Exit.");
        }

    } while(ch != 'Q');

    return 0;
}
```

```
Example 4.20/* to print equivalent value for a given character */
#include <stdio.h>

main()
{
  char ch;

  do
      ch = getche();
      printf ( "%d", ch );
  while ( ch++ < 'z' );

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

example 4.21 /* DO…While with Continue */

```
#include <stdio.h>

int main(void) {

  int total, i, j;

  total = 0;
  do {

    printf("Enter a number (0 to stop): ");
    scanf("%d", &i);

    printf("Enter the number again: ");
    scanf("%d", &j);

    if(i != j) {
      printf("Mismatch\n");
      continue;
    }
   total = total + i;
  } while(i);
  printf("Total is %d\n", total);
  return 0;
}
```

4.6 for statement

The third and last looping construct in C is the for loop. The for loop can execute a block of code for a fixed or given number of times. Its syntax is as follows.

```
for (initializations;test conditions;increment value)
{
   block of code
}
```

The simplest way to understand for loops is to study several examples.

First, here is a for loop that counts from 1 to 10.

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

The test conditions may be unrelated to the variables being initialized and updated. Here is a loop that counts until a user response terminates the loop.

```
for (count = 1; response != 'N'; count++)
{
   printf("%d\n",count);
   printf("Dam man, you still want to continue? (Y/N): \n");
   scanf("%c",&response);
}
```

More complicated test conditions are also allowed. Suppose the user of the last example never enters "N", but the loop should terminate when 100 is reached, regardless.

```
for (count = 1; (response != 'N') && (count <= 100); count++)
{
   printf("%d\n",count);
   printf("Dam man, you still want to continue? (Y/N): \n");
   scanf("%c",&response);
}
```

It is also possible to have multiple initializations and multiple actions. This loop starts one counter at 0 and another at 100, and finds a midpoint between them. (This is advanced material).

```
for (i = 0, j = 100; j != i; i++, j--)
{
   printf("i = %d, j = %d\n",i,j);
}

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

All of the constituent parts of the statement are optional. The initialization, condition, termination sections of the for loop can be blank. For instance, suppose we need to count from a user specified number to 100. The first semicolon is still required as a place keeper

```
printf("Enter a number to start the count: ");
scanf("%d",&count);

for ( ; count < 100 ; count++)
{
   printf("%d\n",count);
}
```

The actions are also optional. Here is a silly example that will repeatedly echo a single number until a user terminates the loop;

```
for (number = 5; response != 'Y';)
{
   printf("%d\n",number);
   printf("You still want to look at this? (Y/N) \n");
   scanf("%c",&response);

}
```

Example 4.22

```
main()
{
int row,column;
puts("\t\tMY Handy multiplication table");
for(row=1;tow<=10;row++)
{
for(column=1;column<=10;column++)
printf("%6d", row*column);
putchar('\n');
}
}
```

The output is a multiplication table of 10x10 size.

Example 4.23

/* This example explain implementation of backward for construct */

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

  long sum = 0L;

  int count = 10;  /* The number of integers to be summed */

  int i = 0;      /* The loop counter                   */


  /* Sum integers from count to 1 */
  for (i = count ; i >= 1 ; sum += i-- );
  printf("\nTotal of the first %d numbers is %ld\n", count, sum);
}
```

Example 4.24
/* every part in for structure is optional */

```
#include <stdio.h>
 main()
{
   long luckyNumber = 15;
   int yourInput = 0;
   int count = 3;    /* The maximum number of tries */
   for( ; count>0 ; --count) {

     printf("\nYou have %d tr%s left.", count, count == 1 ? "y" : "ies");
     printf("\nEnter: ");            /* Prompt for a guess  */
     scanf("%d", &yourInput);        /* Read in a guess     */

     /* Check for a correct guess */
     if (yourInput == luckyNumber)
     {
       printf("\nYou guessed it!\n");
       return;  /* End the program */
     }

     /* Check for an invalid guess */
     if(yourInput<1 || yourInput > 20)
```

```
        printf("I said between 1 and 20.\n ");
      else
        printf("Sorry. %d is wrong.\n", yourInput);
    }
    printf("\nYou have had three tries and failed. The number was %ld\n"
                                              , luckyNumber);
}
```

Example 4.25/* This example will draw a box */

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

{
    int count = 0;
    printf("\n$$$$$$$$$$$$$$$$$$$$$");     /* box top */

    for(count = 1 ; count <= 8 ; ++count)
      printf("\n$                    $");   /* box sides   */

    printf("\n$$$$$$$$$$$$$$$$$$$$$\n");   /* bottom of the box */
}
```

4.7 GO TO

C support unconditional control statement called as “goto”, which transfer the control from one location to another location with out checking the condition. The location where to transfer the control is specified by label name. For example,

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

{
  int i;
  i = 1;
  again:
  printf("%d ", i);
  i++;
  if(i<10)
      goto again;
  return 0;
 }
```

It transfer the control from ‘if’ statement to the location specified by “again” when the condition (i<10) becomes true.

4.8 Let us Sum-up

After read this lesson, we are clearly understand about various conditional and unconditional control structures used for implementing programs. The syntax of every loop structure and decision-making structures were discussed elaborately. Even though "C" supports unconditional control structure (ie., goto), its always suggestible to avoid "goto" statement in good programming practice.

4.9 Points for Discussion

Differentiate while and do…while structures.
Write about the syntax of for loop?
How the switch construct varies from else..if…ladder?

4.10 Check your Progress

Differentiate while and do…while structures.

If the question comes like differentiate two statements or structures then you should explain in the form of table, in which you should mention all the comparable information.

While	**do- while**
1. Condition will be executed first	statement block will be executed first
2. Statement block may or may no be executed.	statement block will be executed atleast one time
…………..	
…………..	

Define "break" statement

The break statement is used to terminate execution from current block.

4.11 Lesson-end Activities

1. Is there any restriction for using while loop?
2. Do we need Do..While loop?

3. What do you mean by branching statements?

4.12 References

Peter Aitken, Teach Yourself C in 21 Days, Fourth Edition, Sams Publisher.
Brian W. Kernighan and Dennis M. Ritchie, The C programming Language, Prentice-Hall in 1988
E.Balagursamy, Programming in Ansi C, TATA MCGraw Hill
http://publications.gbdirect.co.uk/c_book
http://www.cs.cf.ac.uk/Dave/C/
http://www.oreilly.com/catalog/pcp3/
http://www.cs.utah.edu/dept/
http://vergil.chemistry.gatech.edu/resources/programming/c-tutorial
http://sysprog.net
http://www.mycplus.com/
http://www.programmersheaven.com/download/
http://en.literateprograms.org/

UNIT – II Arrays

LESSON – 5

CONTENTS

5.0 Aims and objectives

In this lesson, you will learn about C Programming - Arrays - Declaration of arrays, Initialization of arrays, Multi dimensional Arrays, Elements of multi dimension arrays and Initialization of multidimensional arrays.

5.1 Array - Introduction

The C language provides a capability that enables the user to define a set of ordered data items known as an array. C uses arrays as a way of describing a collection of variables with identical properties. The group has a single name for all of the members, with the individual members being selected by an *index*.

5.2 Declaration of array

Like any other variable arrays must be declared before they are used. The general form of declaration is:

type variable-name[50];

The type specifies the type of the elements that will be contained in the array, such as int float or char and the size indicates the maximum number of elements that can be stored inside the array for ex:

float height[50];

declares the height to be an array containing 50 real elements. Any subscripts 0 to 49 are valid. In C the array elements index or subscript begins with number zero. So height [0] refers to the first element of the array. (For this reason, it is easier to think of it as referring to element number zero, rather than as referring to the first element).

As individual array element can be used anywhere that a normal variable with a statement such as

G = grade [50];

The statement assigns the value stored in the 50th index of the array to the variable ‘g’. More generally if ‘I’ is declared to be an integer variable, then the statement g=grades [I]; Will take the value contained in the element number I of the grades array to assign it to g. so if ‘I’ were equal to 7 when the above statement is executed, then the value of grades [7] would get assigned to g.

The name of the array is `ar` and its members are accessed as `ar[0]` through to `ar[99]` inclusive, as Figure 5.2.1 shows.

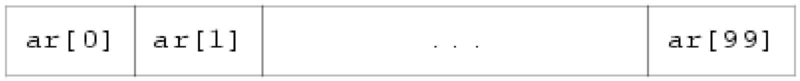
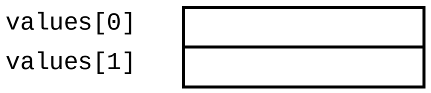


Figure 5.2.1. 100 element array

Each of the hundred members is a separate variable whose type is `double`. Without exception, all arrays in C are numbered from `0` up to one less than the bound given in the declaration

The declaration int values[10]; would reserve enough space for an array called values that could hold up to 10 integers. Refer to the figure 5.2.2 below given picture to conceptualize the reserved storage space.



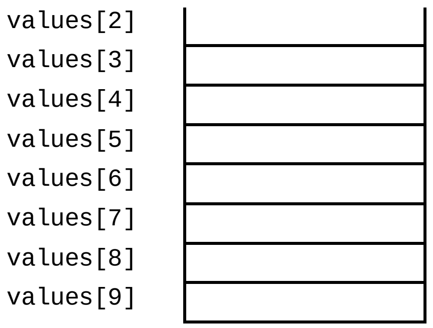


Figure 5.2.2 Internal organization of array in memory.

The array values stored in the memory.

Arrays are a data structure which hold multiple variables of the same data type. Consider the case where a programmer needs to keep track of a number of people within an organization. So far, our initial attempt will be to create a specific variable for each user. This might look like,

```
int name1 = 101;
int name2 = 232;
int name3 = 231;
```

It becomes increasingly more difficult to keep track of this as the number of variables increase. Arrays offer a solution to this problem.

An array is a multi-element box, a bit like a filing cabinet, and uses an indexing system to find each variable stored within it. In C, indexing starts at **zero**.

Arrays, like other variables in C, must be declared before they can be used.
The replacement of the above example using arrays looks like,

```
int names[4];
names[0] = 101;
names[1] = 232;
names[2] = 231;
names[3] = 0;
```

We created an array called *names*, which has space for four integer variables. You may also see that we stored 0 in the last space of the array. This is a common technique used by C programmers to signify the end of an array.

Arrays have the following syntax, using square brackets to access each indexed value (called an **element**).

```
x[i]
```

so that *x[5]* refers to the sixth element in an array called *x*. In C, array elements start with 0. Assigning values to array elements is done by,

```
x[10] = g;
```

and assigning array elements to a variable is done by,

```
g = x[10];
```

In the following example, a character based array named *word* is declared, and each element is assigned a character. The last element is filled with a zero value, to signify the end of the character string (in C, there is no string type, so character based arrays are used to hold strings). A printf statement is then used to print out all elements of the array.

Example 5.1
```
/*  Introducing array's, 2  */
#include <stdio.h>

main()
{
        char word[20];

        word[0] = 'H';
        word[1] = 'e';
        word[2] = 'l';
        word[3] = 'l';
        word[4] = 'o';
        word[5] = 0;
        printf("The contents of word[] is -->%s\n", word );
}
```

Sample Program Output

```
The contents of word[] is Hello
```

If we want an array of six integers , called "numbers", we write in C

```
int numbers[6];
```

For a character array called letters,

```
char letters[6]; and so on.
```

Classification of array

Arrays will be generally classified in to two category as follows

(i). Single dimension
(ii). Multiple dimension

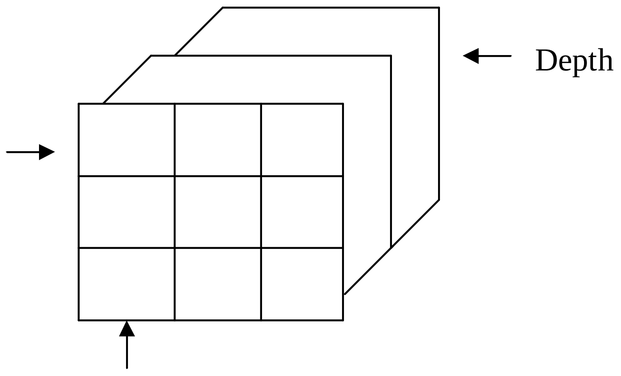
This classification is based on the number of subscript used to declare array variable. If an array is having only one subscript is referred as single dimension array. If an array declared by more than one subscript then it referred as multi dimension array. Consider an example of declaring two dimensional array,

int a[3][3];
declares 'a' as two dimensional array having 9 (ie, 3X3) elements. The first element starts with the index a[0][0] and last element will be at the location a[2][2] as shown in the figure 5.2.1.

a[0][0]	a[0][1]	a[0][1]
a[1][0]	a[1][1]	a[1][2]
a[2][0]	a[2][1]	a[2][2]

Figure 5.2.1 Two diamentional array

Similarly the multi dimensional array can be specified as follows,



Row

Column

Figure 5.2.2

5.3 Initialization of arrays:

We can initialize the elements in the array in the same way as the ordinary variables when they are declared. The general form of initialization off arrays is:

```
type array_name[size]={list of values};
```

The values in the list care separated by commas, for example the statement

```
int point[6]={0,0,1,0,0,0};
```

If we want to access a variable stored in an array, for example with the above declaration, the following code will store a 1 in the variable `x`

```
int x;
x = point[2];
```

Arrays in C are indexed starting at 0, as opposed to starting at 1. The first element of the array above is `point[0]`. The index to the last value in the array is the array size minus one. In the example above the subscripts run from 0 through 5. C does not guarantee bounds checking on array accesses. The compiler may not complain about the following (though the best compilers do):

```
char y;
int z = 9;
char point[6] = { 1, 2, 3, 4, 5, 6 };
//examples of accessing outside the array. A compile error is not always raised
y = point[15];
y = point[-4];
y = point[z];
```

During program execution, an out of bounds array access does not always cause a run time error. Your program may happily continue after retrieving a value from point[-1]. To alleviate indexing problems, the sizeof() expression is commonly used when coding loops that process arrays.

```
int ix;
short anArray[]= { 3, 6, 9, 12, 15 };
for (ix=0; ix< (sizeof(anArray)/sizeof(short)); ++ix)
 {
  DoSomethingWith( anArray[ix] );
}
```

Notice in the above example, the size of the array was not explicitly specified. The compiler knows to size it at 5 because of the five values in the initializer list. Adding an additional value to the list will cause it to be sized to six, and because of the sizeof expression in the `for` loop, the code automatically adjusts to this change. Good programming practice is declare a variable *size* and store the size of the array.

```
size = sizeof(anArray)/sizeof(short)
```

The initialization of arrays in c suffers two draw backs
1. There is no convenient way to initialize only selected elements.
2. There is no shortcut method to initialize large number of elements.

Example 5.2 /* Program to count the no of positive and negative numbers*/

```
#include< stdio.h >
void main( )
{
int a[50],n,count_neg=0,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(I=0;I < n;I++)
{
if(a[I] < 0)
count_neg++;
else
count_pos++;
}
printf(“There are %d negative numbers in the array\n”,count_neg);
printf(“There are %d positive numbers in the array\n”,count_pos);
}
```

One important point about array declarations is that they don't permit the use of varying subscripts. The numbers given must be constant expression which can be evaluated at compile time, not run time. For example, this function incorrectly tries to use its argument in the size of an array declaration:

```
f(int x)
```

```
{
    char var_sized_array[x];        /* FORBIDDEN */
}
```

It's forbidden because the value of x is unknown when the program is compiled; it's a run-time, not a compile-time, value.

To tell the truth, it would be easy to support arrays whose *first* dimension is variable, but neither Old C nor the Standard permits it, although we do know of one Very Old C compiler that used to do it.

```
/*
 Example 5.3  How to read elements into an array.
*/

#include <stdio.h>

main()
{
  /*  Declaration Statements  */
  char month[12][4];
  short units[12];
  double sales[12];
  short i;

  printf("C51.C -> How to read elements into an array \n");

  /*  Assignment Statements  */
  for (i = 1; i <= 12; i++) {
    printf("\nEnter the month (Jan,Feb,...)   : ");
    scanf("%s",month[i-1]);


   /*  Don't forget, in C the first element of an array is 0 */
    /*  and not 1.                                          */

    printf("\nEnter number of units sold    : ");
    scanf("%hd", &units[i - 1]);
    printf("\nEnter sales (in million $)    : ");
    scanf("%lg", &sales[i - 1]);
  }  /*  End of For{} loop  */

  return(0);
}
/*  End of Program C51  */
```

```
/*
 Example 5.4   This program demonstrates how to sum all the elements of an array.
*/

#include <stdio.h>

main()
{
  /*  Declaration Statements  */
  char month[12][4];
  short units[12];
  double sales[12];
  double ss;            /* ss stands for sum of sales */
  short i, j, su, k;    /* su stands for sum of units */

  clrscr();
  printf("C52.C -> How to sum all the elements of an array \n");

  /*  Assignment Statements  */

  for (i = 1; i <= 12; i++) {
    printf("\nMonth (Jan,Feb,...) : ");
    scanf("%s",month[i-1]);
    printf("\nUnits Sold          : ");
    scanf("%hd", &units[i - 1]);
    printf("\nSales (in million $): ");
    scanf("%lg", &sales[i - 1]);
  }

  i = 0;
  for (j = 1; j <= 4; j++) {
    su = 0;
    ss = 0.0;
    for (k = 1; k <= 3; k++) {
      i++;
      su += units[i - 1];
      ss += sales[i - 1];
    }  /*  End of inner for{} loop  */

    /*  Print result by quarter  */
    printf("\nquarter%2d%7d cars     $%5.1f m\n\n", j, su, ss);
  }  /*  End of outer for{} loop  */

  return(0);
}
/* End of Program C52 */
```

```
/*
Example 5.5    How to sum part of an array.
*/

#include <stdio.h>

main()
{
  /*  Declaration Statements  */
  char month[12][4];
  short units[12];
  short uq[4];      /*  uq stands for units per quarter  */
  double sales[12];
  double sq[4];     /*  dq stands for sales per quarter  */
  short i, j, k;
  char *TEMP;

  printf("C53.C -> Summing part of an array \n");

  /*  Assignment Statements  */
  for (i = 1; i <= 12; i++) {
    printf("Month (Jan,Feb,...) : ");
    scanf("%s",month[i-1]);
    printf("Units Sold          : ");
    scanf("%hd", &units[i - 1]);
    printf("Sales (in million $): ");
    scanf("%lg", &sales[i - 1]);
  }

  /*  Form 4 quarter totals  */

  i = 0;
  for (j = 1; j <= 4; j++) {
    uq[j - 1] = 0;
    sq[j - 1] = 0.0;
    for (k = 1; k <= 3; k++) {
      i++;
      uq[j - 1] += units[i - 1];
      sq[j - 1] += sales[i - 1];
    }  /*  End of inner for{} loop  */
  }  /*  End of outer for{} loop  */

  /*  Print results  */
  printf("\n Cars sold by quarter %5d%5d%5d%5d\n", uq[0], uq[1], uq[2],
         uq[3]);
  printf("\n Sales by quarter     % .1E% .1E% .1E% .1E\n", sq[0], sq[1],
         sq[2], sq[3]);

  return(0);
}
/* End of Program C53 */
```

Example 5.6

```
/*
  Several operations on an array (largest/smallest, average, etc)
*/

#include <stdio.h>

main()
{
  /*  Declaration Statements  */
  char month[12][4];
  short units[12];
  double sales[12];
  short uq[4];
  double sq[4];
  short i, cars, mins, maxs, j, k, iave;
  double totals, big, min, ave;

  printf("C54.C -> Operations an array \n");

  /*  Assignment Statements  */

  for (i = 1; i <= 12; i++)
  {                                  /* Read value from */
    printf("Month (Jan,Feb,...) : ");    /* keyboard entry. */
    scanf("%s",month[i-1]);
    printf("Units Sold          : ");
    scanf("%hd", &units[i - 1]);
    printf("Sales (in million $): ");
    scanf("%lg", &sales[i - 1]);
  }

  /*  Initializing variables  */
  cars = units[0];
  totals = sales[0];
  big = totals;
```

```
min = totals;
mins = 1;
maxs = 1;

for (i = 2; i <= 12; i++)
{
  cars += units[i - 1];
  totals += sales[i - 1];
  if (sales[i - 1] > big)     /* It selects the biggest sales  */
  {                           /* and keep track of the index   */
    big = sales[i - 1];       /* to know when it occurs.       */
    maxs = i;
  }  /*  End of if statement  */
  else
  {
     if (sales[i - 1] < min)  /* It selects the smallest sales */
     {                        /* and keep track of the index   */
     min = sales[i - 1];      /* to know when it occurs.       */
     mins = i;
     }
  }  /*  End of else statement  */
}  /*  End of for{} loop  */

i = 0;
for (j = 1; j <= 4; j++)
{
  uq[j - 1] = 0;
  sq[j - 1] = 0.0;
  for (k = 1; k <= 3; k++)
  {
    i++;
    uq[j - 1] += units[i - 1];
    sq[j - 1] += sales[i - 1];
  }  /*  End of inner for{} loop  */
}  /*  End of outer for{} loop  */

/*  Output original data and results  */

printf("\n");
printf("Month unit  sales m$\n");
for (k = 1; k <= 12; k++)
  printf("%3c%s%6d%6.1f\n",' ', month[k-1], units[k-1], sales[k-1]);
printf("\nTotals%5d%6.1f\n", cars, totals);

iave = cars / 12;      /*  Compute the units sold average */
ave = totals / 12.0;   /*  and sales average by month.    */
```

```
  printf("\Nave.%7d%6.1f\n\n", iave, ave);
  printf("Best/worst-%5.1f%5.1f\n\n", big, min);
  printf("Occurred in%2c%s%2c%s\n\n",
        ' ', month[maxs - 1], ' ', month[mins - 1]);
  printf("Cars sold by quarter%5d%5d%5d%5d\n", uq[0], uq[1], uq[2],
        uq[3]);
  printf("Sales by quarter    %5.1f%5.1f%5.1f%5.1f\n",
        sq[0], sq[1], sq[2], sq[3]);

  return(0);
}
/*  End of Program C54  */
```

5.4 Character array

The most common type of array in C is the array of characters. To illustrate the use of character arrays and functions to manipulate them, let's write a program that reads a set of text lines and prints the longest. The outline is simple enough:

```
while (there's another line)
if (it's longer than the previous longest)
(save it)
(save its length)
print longest line
```

This outline makes it clear that the program divides naturally into pieces. One piece gets a new line, another saves it, and the rest controls the process.

Since things divide so nicely, it would be well to write them that way too. Accordingly, let us first write a separate function getline to fetch the next line of input. We will try to make the function useful in other contexts. At the minimum, getline has to return a signal about possible end of file; a more useful design would be to return the length of the line, or zero if end of file is encountered. Zero is an acceptable end-of-file return because it is never a valid line length. Every text line has at least one character; even a line containing only a newline has length 1. Then we find a line that is longer than the previous longest line, it must be saved somewhere. This suggests a second function, copy, to copy the new line to a safe place. Finally, we need a main program to control getline and copy. Here is the result.

Example 5.7

```
#include <stdio.h>
#define MAXLINE 1000 /* maximum input line length */
int getline(char line[], int maxline);
void copy(char to[], char from[]);
```

```
/* print the longest input line */
main()
{
int len; /* current line length */
int max; /* maximum length seen so far */
char line[MAXLINE]; /* current input line */
char longest[MAXLINE]; /* longest line saved here */
max = 0;
while ((len = getline(line, MAXLINE)) > 0)
if (len > max) {
max = len;
copy(longest, line);
}
if (max > 0) /* there was a line */
printf("%s", longest);
return 0;
}
/* getline: read a line into s, return length */
int getline(char s[],int lim)
{
int c, i;
for (i=0; i < lim-1 &(c=getchar())!=EOF &c!='\n'; ++i)
s[i] = c;
if (c == '\n') {
s[i] = c;
++i;
}
s[i] = '\0';
return i;
}
/* copy: copy 'from' into 'to'; assume to is big enough */
void copy(char to[], char from[])
{
int i;
i = 0;
while ((to[i] = from[i]) != '\0')
++i;
}
```

The functions getline and copy are declared at the beginning of the program, which we assume is contained in one file. main and getline communicate through a pair of arguments and a returned value. In getline, the arguments are declared by the line

```
int getline(char s[], int lim);
```

which specifies that the first argument, s, is an array, and the second, lim, is an integer. The purpose of supplying the size of an array in a declaration is to set aside storage. The length of an array s is not necessary in getline since its size is set in main. getline uses return to send a value back to the caller, just as the function power did. This line also declares that getline returns an int; since int is the default return type, it could be omitted.

Some functions return a useful value; others, like copy, are used only for their effect and return no value. The return type of copy is void, which states explicitly that no value is returned.

getline puts the character '\0' (the *null characte*r, whose value is zero) at the end of the array it is creating, to mark the end of the string of characters. This conversion is also used by the C language: when a string constant like

"hello\n"

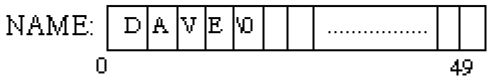
appears in a C program, it is stored as an array of characters containing the characters in the string and terminated with a '\0' to mark the end.

The %s format specification in printf expects the corresponding argument to be a string represented in this form. copy also relies on the fact that its input argument is terminated with a '\0', and copies this character into the output.

It is worth mentioning in passing that even a program as small as this one presents some sticky design problems. For example, what should main do if it encounters a line which is bigger than its limit? Getline works safely, in that it stops collecting when the array is full, even if no newline has been seen. By testing the length and the last character returned, main can determine whether the line was too long, and then cope as it wishes. In the interests of brevity, we have ignored this issue. There is no way for a user of getline to know in advance how long an input line might be, so getline checks for overflow. On the other hand, the user of copy already knows (or can find out) how big the strings are, so we have chosen not to add error checking to it.

5.5 String Handling functions

Recall from our discussion of arrays that strings are defined as an array of characters or a pointer to a portion of memory containing ASCII characters. A `string` in C is a sequence of zero or more characters followed by a `NULL` (\0)character:



It is important to preserve the `NULL` terminating character as it is how C defines and manages variable length strings. **All** the C standard library functions require this for successful operation.

In general, apart from some *length-restricted functions* (`strncat(), strncmp,()` and `strncpy()`), unless you create strings by hand you should not encounter any such problems, . You should use the many useful string handling functions and not really need to *get your hands dirty* dismantling and assembling strings.

All the string handling functions are prototyped in:

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

The common functions are described below:

char *stpcpy (const char *dest,const char *src) -- Copy one string into another.

int strcmp(const char *string1,const char *string2) - Compare string1 and string2 to determine alphabetic order.

char *strcpy(const char *string1,const char *string2) -- Copy string2 to stringl.

char *strerror(int errnum) -- Get error message corresponding to specified error number.

int strlen(const char *string) -- Determine the length of a string.

char *strncat(const char *string1, char *string2, size_t n) -- Append n characters from string2 to stringl.

int strncmp(const char *string1, char *string2, size_t n) -- Compare first n characters of two strings.

char *strncpy(const char *string1,const char *string2, size_t n) -- Copy first n characters of string2 to stringl .

int strcasecmp(const char *s1, const char *s2) -- case insensitive version of strcmp().
int strncasecmp(const char *s1, const char *s2, int n) -- case insensitive version of strncmp().

The use of most of the functions is straightforward, for example:

```
char *str1 = "HELLO";
char *str2;
int length;

length = strlen("HELLO"); /* length = 5 */
```

(void) strcpy(str2,str1);

Note that both strcat() and strcopy() both return a copy of their first argument which is the destination array. Note the order of the arguments is *destination array* followed by *source array* which is sometimes easy to get the wrong around when programming.
The strcmp() function *lexically* compares the two input strings and returns:

Less than zero
-- if string1 is lexically less than string2
Zero
-- if string1 and string2 are lexically equal
Greater than zero
-- if string1 is lexically greater than string2

This can also confuse beginners and experience programmers forget this too. The strncat(), strncmp,() and strncpy() copy functions are string restricted version of their more general counterparts. They perform a similar task but only up to the first n characters. Note the the NULL terminated requirement may get violated when using these functions, for example:

char *str1 = "HELLO";
char *str2;
int length = 2;

(void) strcpy(str2,str1, length); /* str2 = "HE" */
str2 is not null terminated

5.5.1 String Searching Functions

The library also provides several string searching functions:

char *strchr(const char *string, int c) -- Find first occurrence of character c in string.

char *strrchr(const char *string, int c) -- Find last occurrence of character c in string.

char *strstr(const char *s1, const char *s2) -- locates the first occurrence of the string `s2` in string `s1`.

`char *strpbrk(const char *s1, const char *s2)` -- returns a pointer to the first occurrence in string s1 of any character from string `s2`, or a null pointer if no character from `s2` exists in `s1`

`size_t strspn(const char *s1, const char *s2)` -- returns the number of characters at the begining of `s1` that match `s2`.

`size_t strcspn(const char *s1, const char *s2)` -- returns the number of characters at the begining of `s1` that *do not* match `s2`.

char *strtok(char *s1, const char *s2) -- break the string pointed to by s1 into a sequence of tokens, each of which is delimited by one or more characters from the string pointed to by s2.

char *strtok_r(char *s1, const char *s2, char **lasts) -- has the same functionality as strtok() except that a pointer to a string placeholder lasts must be supplied by the caller. strchr() and strrchr() are the simplest to use, for example:

```
char *str1 = "Hello";
char *ans;

ans = strchr(str1,'l');
```

After this execution, ans points to the location str1 + 2

strpbrk() is a more general function that searches for the first occurrence of any of a group of characters, for example:

```
char *str1 = "Hello";
char *ans;

ans = strpbrk(str1,'aeiou');
```

Here, ans points to the location str1 + 1, the location of the first e.

strstr() returns a pointer to the specified search string or a null pointer if the string is not found. If s2 points to a string with zero length (that is, the string ""), the function returns s1. For example,

```
char *str1 = "Hello";
char *ans;

ans = strstr(str1,'lo');
```
will yield ans = str + 3.
strtok() is a little more complicated in operation. If the first argument is not NULL then the function finds the position of any of the second argument characters. However, the position is remembered and any subsequent calls to strtok() will start from this position if on these subsequent calls the first argument is NULL. For example, If we wish to break up the string str1 at each space and print each token on a new line we could do:

```
char *str1 = "Hello Big Boy";
char *t1;
```

```
for ( t1 = strtok(str1," ");
    t1 != NULL;
    t1 = strtok(NULL, " ") )
printf("%s\n",t1);
```

Here we use the for loop in a non-standard counting fashion:

- The initialization calls strtok() loads the function with the string str1
- We terminate when t1 is NULL
- We keep assigning tokens of str1 to t1 until termination by calling strtok() with a NULL first argument

5.5.2 Character testing:

The header file <ctype.h> which contains many useful functions to convert and test *single* characters. The common functions are prototypes as follows:

int isalnum(int c) -- True if c is alphanumeric.
int isalpha(int c) -- True if c is a letter.
int isascii(int c) -- True if c is ASCII .
int iscntrl(int c) -- True if c is a control character.
int isdigit(int c) -- True if c is a decimal digit
int isgraph(int c) -- True if c is a graphical character.
int islower(int c) -- True if c is a lowercase letter
int isprint(int c) -- True if c is a printable character
int ispunct (int c) -- True if c is a punctuation character.
int isspace(int c) -- True if c is a space character.
int isupper(int c) -- True if c is an uppercase letter.
int isxdigit(int c) -- True if c is a hexadecimal digit

5.5.3 Character Conversion:

int toascii(int c) -- Convert c to ASCII .
tolower(int c) -- Convert c to lowercase.
int toupper(int c) -- Convert c to uppercase.

The use of these functions is straightforward and we do not give examples here.

5.6 Let us sum-up

In this lesson, we discussed briefly about Arrays and its declaration, Initialization of arrays, single and Multi dimensional Array, Elements of multi dimension arrays and Initialization of multidimensional arrays. We also discussed about character array and its implementation in programming.

5.7 Points for discussion

1. What will be the first value of index by default?
2. Give an example for multidimensional array.
3. How to initialize array elements?
4. What are the advantages and disadvantages of array?

5.8 Check your progress

How to initialize array elements?

Array can be initialize in two different ways. First, at the time of declaring an array, we can assign the values to array members. Second, the array members can be initializing individually.

Advantages and disadvantages

Advantage

- ➢ We can group more number of similar data under a single label name.
- ➢ The elements of the array will be stored in consecutive memory locations.

Disadvantage

We cant group different data under the same name.

5.9 Lesson-end Activities

1. How array concept will helps you to increase the efficiency of your program?
2. Do you feel array is flexible compare with individual variables?
3. Is there any limitations for an index used in array?

5.10 Suggested Readings/References/Sources

1.Peter Aitken, Teach Yourself C in 21 Days, Fourth Edition, Sams Publisher.
2.Brian W. Kernighan and Dennis M. Ritchie, The C programming Language, Prentice-Hall in 1988
3.E.Balagursamy, Programming in Ansi C, TATA MCGraw Hill
http://sysprog.net/
http://www.mycplus.com/cplus
http://www.programmersheaven.com/download/

LESSON – 6 Functions

CONTENTS

6.0 Aims and Objectives

The objective of this lesson is to make the reader to understand and develop programming style by using function. The effective programming includes two major factors, such as, size of the program and time need to execute a program. Both factors can be achieved by using functions.

6.1 Introduction

A function in C can perform a particular task, and supports the concept of modular programming design techniques. We have already been exposed to functions. The main body of a C program, identified by the keyword *main*, and enclosed by the left and right braces is a function. It is called by the operating system when the program is loaded, and when terminated, returns to the operating system.

6.2 Classification of function

Generally, the function can be classified in to three categories as follows.

(1). Function with no argument and no return value

(2). Function with argument and no return value

(3). Function with argument and return value.

Functions have a basic structure. Their format is:

```
return_ data_type function name (arguments, arguments)
 data _type _declarations of arguments;
          {
          function body

         }
```

It is worth noting that a return data type is assumed to be of type *int* unless otherwise specified, thus programs we have seen so far imply that *main()* returns an integer to the operating system. ANSI C varies slightly in the way that functions are declared. Its format is:

```
return_data_type function name (data type variable name, data type variable name,….)
{

   function                                                                      body

}
```

This permits type checking by utilizing function prototypes to inform the compiler of the type and number of parameters a function accepts. When calling a function, this information is used to perform the type and parameter checking.

ANSI C also requires that the return_ data_type for a function which does not return data must my of type *void*. The default return_ data_type is assumed to be an integer unless otherwise specified, but must match that which the function declaration specifies. A simple function is:

```
int print_message ()
{
   printf(“This is a module called print_message.\n”);

}
```

Note the function name is print_message. No arguments are accepted by the function, this is indicated by the keyword void in accepted parameter section of function declaration. The return_data_type is void, this data is not returned by the function. An ANSI C function prototype for print_message() is:

```
int print_message ();
```

Function prototypes are listed at the beginning of the source file. Often, they might be placed in a users .h (header) file.Now lets incorporate this function into a program.

```
/*  Example 6.1 Program illustrating a simple function call */

#include <stdio.h>

void print_message (void);      /* ANSI C function prototype */

void print_message (void)      /* the function code */
{

   printf(“This is a module called print_message.\n”);
}

int main(void)
{

   print_message();

   return 0;

}
```

Sample program output

```
This is a module called print_message.
```

To call a function, it is only necessary to write its name. The code associated with the function is executed at this point in the program. When the function terminates, execution begins with the statement which follows the function name.

In the above program, execution begins at *main()*. The only statement inside main the main body of the program is a call to the code of function *print_message()*. This code is executed, and when finished returns control back to *main()*.

As there is no further example, the function accepts a single data variable, but does not return any information.

```
/*Example 6.2  Program to calculate a specific factorial number */

#iinclude <stdio.h>

void calc_factorial (int); // ANSI function prototype

void calc_factorial (int i)
{

   int I, factorial_number = 1;

   for (i=1; I <=n; ++i)
       factorial_number *= I;

   printf("The factorial of %d is %d\n", n, factorial_number);

}
int main(void)
{

   int number = 0;

   printf("Enter a number\n");

   scanf("%d", &number);
   calc_factorial (number);

   return 0;

}
```

Sample program output

```
Enter a number
3
The factorial of 3 is 6
```

Lets look at the function *calc_factorial()*. The declaration of the function

```
   void calc_factorial (int n)
```

Indicates there is no return data type and a single integer is accepted, known inside the body of the function as n. Next comes the declaration of the **local variables**.

```
   int i, factorial_ number = 0;
```

It is more correct in C to use:

```
   auto int i, factorial_number = 0;
```

As the keyword ***auto*** designates to the compiler that the variable is local. The program works by accepting a variable from the keyboard which is then passed to the function. In other words, the variable number inside the main body is then copied to the variable n in the function, which then calculates the correct answer.

Returning function results

This is done by the use of the keyword return, followed by a data_variable or constant value, the data type os which must match that of the declared return_data_type for the function.

```
float add_numbers (float n1, float n2)
{

   return n1 + n2;          // legal
   return 6;                // illegal, not the same type

   return 6.0;              // legal

}
```

It is possible for a function to have multiple return statements.

```
int validate_input (char command)
{

   switch (command)
   {
      case ‘+’ :
      case ‘-‘ : return 1;
      case ‘*’ :
      case ‘/’ : return 2;
      default : return 0;
   }
}
```

Here is another example 6.3

```
/* Simple multiply program using argument passing */

#include <stdio.h>

int calc_result (int, int) // ANSI function prototype
{

   auto int result;
   result = numb1 * numb2;
   return result;

}

int main(void)
{

   int digit1 = 10, digit2 = 30, answer = 0;

   answer = calc_result(digit1, digit2);

   printf("%d multiplied by %d is %d\n", digit1, digit2, answer);

   return 0;

}
```

Sample program output

```
10 multiplied by 30 is 300
```

NOTE that the value which is returned from the function (ie result) must be declared in the function.

NOTE: The formal declaration of the function name is preceded by the data type which is returned,

```
int calc_result (numb1, numb2)
```

6.3 Scope of variables

Local and global variables

6.3.1 Local variables:

These variables only exist inside the specific function that creates them. They are unknown to other functions and to the main program. As such, they are normally implemented using a stack. Local variables cease to exist once the function that created them is completed. They are recreated each time a function is executed or called.

6.3.2 Global variables:

These variables can be accessed (ie known) by any function comprising the program. They are implemented by associating memory locations with variable names. They do not get recreated if the function is recalled. To declare a global variable, declare it outside of all the functions. There is no general rule for where outside the functions these should be declared, but declaring them on top of the code is normally recommended for reasons of scope, as explained below. If a variable of the same name is declared both within a function and outside of it, the function will use the variable that was declared within it and ignore the global one.

I recommend to use as few global variables as possible.

Defining global variables:

```
/* Example 6.4Demonstrating global variables */

#include <stdio.h>
int add_numbers( void); // ANSI function prototype
/* These are global variables and can be accessed by functions from this point on */
int value1, value2, value3;
int add_numbers (void)
{

   auto int result;
   result = value1 + value2 + value3;

   return result;

}

int main(void)
{

   auto int result;
   value1 = 10;
   value2 = 20;
   value3 = 30;
   result = add_numbers();
   printf(“The sum of %d + %d + %d is %d\n”, value1, value2, value3, final_result);

   return 0;

}
```

Sample program output

The scope of a global variable can be restricted by carefully placing the declaration. They are visible from the declaration until the end of the current source file.

Example 6.5

```
#include <stdio.h>

int no_access (void);     // ANSI function prototype
int all_access(void);

static int n2; // n2 is known from this point onwards
int no_access (void)
{
   n1 = 10;              // illegal, n1 not yet known
   n2 = 5;               // valid
}

int all_access(void)
{

   n1 = 10;              // valid
   n2 = 3;               // valid

}
```

Example 6.6 A program that uses a function to calculate the cube of a number.

```
/* Demonstrates a simple function */
#include <stdio.h>

long cube(long x);

long input, answer;

```

```
main()
{
printf("Enter an integer value: ");
scanf("%d", &input);
answer = cube(input);
/* Note: %ld is the conversion specifier for */
/* a long integer */
printf("\nThe cube of %ld is %ld.\n", input, answer);

return 0;
}

/* Function: cube() - Calculates the cubed value of a variable */
long cube(long x)
{
long x_cubed;

x_cubed = x * x * x;
return x_cubed;
}
```

```
Enter an integer value: 100
The cube of 100 is 1000000.
Enter an integer value: 9
The cube of 9 is 729.
Enter an integer value: 3

The cube of 3 is 27.
```

Example 6.7 Using multiple return statements in a function.

```
/* Demonstrates using multiple return statements in a function. */

#include <stdio.h>

int x, y, z;

int larger_of( int , int );

main()
{
puts("Enter two different integer values: ");
scanf("%d%d", &x, &y);

z = larger_of(x,y);

printf("\nThe larger value is %d.", z);

return 0;
}

int larger_of( int a, int b)
{
if (a > b)
return a;
else
return b;
}
```

```
Enter two different integer values:
200 300
The larger value is 300.
Enter two different integer values:
300
200
The larger value is 300.
```

6.4 Recursion

The term *recursion* refers to a situation in which a function calls itself either directly or indirectly. *Indirect recursion* occurs when one function calls another function that then calls the first function. C allows recursive functions, and they can be useful in some situations. For example, recursion can be used to calculate the factorial of a number. The factorial of a number x is written x! and is calculated as follows:

x! = x * (x-1) * (x-2) * (x-3) * ... * (2) * 1

However, you can also calculate x! like this:

x! = x * (x-1)!

Going one step further, you can calculate (x-1)! using the same procedure:

(x-1)! = (x-1) * (x-2)!

You can continue calculating recursively until you're down to a value of 1, in which case you're finished. The program in Listing 5.5 uses a recursive function to calculate factorials. Because the program uses unsigned integers, it's limited to an input value of 8; the factorial of 9 and larger values are outside the allowed range for integers.

Example 6.8 Using a recursive function to calculate factorials.

```
/* Demonstrates function recursion. Calculates the */
```

```
2: /* factorial of a number. */
3:
4: #include <stdio.h>
5:
6: unsigned int f, x;
7: unsigned int factorial(unsigned int a);
8:
9: main()
10: {
11: puts("Enter an integer value between 1 and 8: ");
12: scanf("%d", &x);
13:
14: if( x > 8 || x < 1)
15: {
16: printf("Only values from 1 to 8 are acceptable!");
17: }
18: else
19: {
20: f = factorial(x);
21: printf("%u factorial equals %u\n", x, f);
22: }
23:
24: return 0;
25: }
26:
27: unsigned int factorial(unsigned int a)
28: {
29: if (a == 1)
30: return 1;
31: else
32: {
33: a *= factorial(a-1);
34: return a;
35: }
36: }
```

```
Enter an integer value between 1 and 8:
6
6 factorial equals 720
```

The first half of this program is like many of the other programs you have worked with so far. It starts with comments on lines 1 and 2. On line 4, the appropriate header file is included for the input/output routines. Line 6 declares a couple of unsigned integer values. Line 7 is a function prototype for the factorial function. Notice that it takes an unsigned int as its parameter and returns an unsigned int. Lines 9 through 25 are the main() function. Lines 11 and 12 print a message asking for a value from 1 to 8 and then accept an entered value.

Lines 14 through 22 show an interesting if statement. Because a value greater than 8 causes a problem, this if statement checks the value. If it's greater than 8, an error message is printed; otherwise, the program figures the factorial on line 20 and prints the result on line 21. When you know there could be a problem, such as a limit on the size of a number, add code to detect the problem and prevent it. Our recursive function, factorial(), is located on lines 27 through 36. The value passed is assigned to a. On line 29, the value of a is checked. If it's 1, the program returns the value of 1. If the value isn't 1, a is set equal to itself times the value of factorial(a-1). The program calls the factorial function again, but this time the value of a is (a-1). If (a-1) isn't equal to 1, factorial() is called again with ((a-1)-1), which is the same as (a-2). This process continues until the if statement on line 29 is true. If the value of the factorial is 3, the factorial is evaluated to the following:

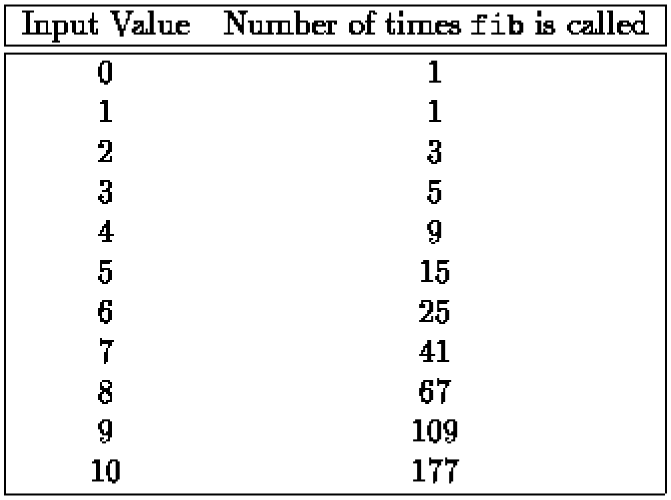
```
3 * (3-1) * ((3-1)-1)
```

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);
}
```



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

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.

Example 6.9 /* program to calculate average of numbers */

```
#include <stdio.h>
#include <stdarg.h>
double average(double v1 , double v2,...);
int main()
{

  printf("\n Average = %lf", average(3.5, 4.5, 0.0));
  printf("\n Average = %lf", average(1.0, 2.0));
  printf("\n Average = %lf\n", average(0.0,1.2,1.5));
}

double average( double v1, double v2,...)
{
  va_list parg;
  double sum = v1+v2;
  double value = 0;
  int count = 2;
  va_start(parg,v2);
  while((value = va_arg(parg, double)) != 0.0)
  {
    sum += value;
    printf("\n in averge = %.2lf", value);
    count++;
  }
  va_end(parg);              /* End variable argument process      */
  return sum/count;
}
```

output

```
Average = 5.250000
Average = 1.900000
Average = 9.100000
```

Example 6.10 * Compute area of triangles *\

```
#include <stdio.h>

float triangle(float width, float height)
{
    float area;

    area = width * height / 2.0;
    return (area);
}

int main()
{
    printf("Triangle #1 %f\n", triangle(1.3, 8.3));
    printf("Triangle #2 %f\n", triangle(4.8, 9.8));
    printf("Triangle #3 %f\n", triangle(1.2, 2.0));
    return (0);
}
```

6.5 Passing arrays as function argument.

Arrays can also be passed to the function. If an array as function argument, we should specify name of the array and size of the array. If we miss to specify the size of the array in formal declaration section, the compiler automatically assign the size of the array during execution based on actual argument.

Example 6.10

```
#include <stdio.h>

void print_onedim(int a[]);
void print_twodim(int a[][4]);
void print_threedim(int a[][3][4]);
```

```
main() {
   int cnt=0;
   int a[2][3][4];
   int i;
   int j;
   int k;

   for(i = 0;i < 2; i++){
      for(j = 0;j < 3; j++){
         for(k = 0;k < 4; k++) {
            a[i][j][k] = cnt;
            cnt++;
         }
      }
   }
   print_onedim(a[1][1]);
   print_twodim(a[1]);
   print_threedim(a);
}

void print_onedim(int a[]) {
   int i;

   for(i = 0; i < 4 ; i++)
      printf("%d ", a[i]);
}

void print_twodim(int a[][4]) {
   int j;
   for(j = 0;j < 3; j++)
      print_onedim(a[j]);

   printf("\n");
}

void print_threedim(int a[][3][4]) {
   int j;

   printf("Each two dimension matrix\n");

   for(j = 0; j < 2 ; j++)
      print_twodim( a [ j ] );

}
```

Example 6.11

```
#include <stdio.h>

void printarr(int a[]) {
   int i;
   for(i = 0;i<5;i++) {
      printf(" %d\n",a[i]);
   }
}

main() {
   int a[5];
   int i;

   for(i = 0;i<5;i++) {
      a[i]=i;
   }
   printarr(a);
}
```

6.6 Let us sum-up

In this lesson, make the reader to understand and develop programming style by using function. The effective programming includes two major factors, such as, size of the program and time need to execute a program. Both factors can be achieved by using functions.

6.7 Points for discussion

1. Why we need functions?

2. Write about two advantages of an array?

3. Define recursion

4. Define the term call by value.

6.8 Check your progress

1. Recursion - A function called itself. This may be either direct or indirect recursion. Give example
2. Explain about string handling function.

You can specify all string handling functions, especially four basic functions, such as, strlen(), strcpy(), strcat(), strcmp(). Give some example also.

6.9 Lesson-end Activities

1. What are the advantages of using functions?

2. What is the purpose of using return statement?

6.10 Suggested Readings/references /sourcing

1.Peter Aitken, Teach Yourself C in 21 Days, Fourth Edition, Sams Publisher.
2.Brian W. Kernighan and Dennis M. Ritchie, The C programming Language, Prentice-Hall in 1988
3.E.Balagursamy, Programming in Ansi C, TATA MCGraw Hill
http://publications.gbdirect.co.uk/c_book
http://www.cs.cf.ac.uk/Dave/C/
http://www.oreilly.com/catalog/pcp3/
http://www.cs.utah.edu/dept/old/texinfo/cpp
http://vergil.chemistry.gatech.edu/resources/programming/c-tutorial
http://sysprog.net/
http://www.mycplus.com

LESSON – 7 Pointers

CONTENTS

7.0 Aims and Objectives

The aim of this lesion is to motivated people to understand about the importance and usage of pointers and its applications.

7.1 Introduction

Pointers are a fundamental part of C. If you cannot use pointers properly then you have basically lost all the power and flexibility that C allows. The secret to C is in its use of pointers.

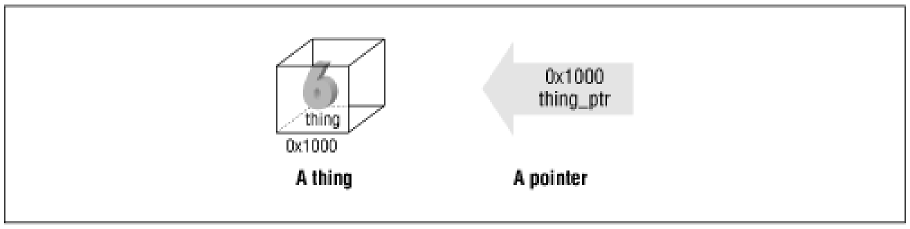
7.2 Fundamentals of pointers

C uses *pointers* a lot. **Why?**:

- It is the only way to express some computations.
- It produces compact and efficient code.
- It provides a very powerful tool.

There are things and pointers to things. Knowing the difference between the two is very important. This concept is illustrated in following Figure 7.2.1.

Figure 7.2.1 A thing and a pointer to a thing



In this book, we use a box to represent a thing. The name of the variable is written on the bottom of the box. In this case, our variable is named `thing`. The value of the variable is 6.

The address of `thing` is `0x1000`. Addresses are automatically assigned by the C compiler to every variable. Normally, you don't have to worry about the addresses of variables, but you should understand that they're there.

Our pointer (`thing_ptr`) points to the variable `thing`. Pointers are also called *address variables* because they contain the addresses of other variables. In this case, our pointer contains the address `0x1000`. Because this is the address of `thing`, we say that `thing_ptr` points to `thing`.

Variables and pointers are much like street addresses and houses. For example, your address might be "214 Green Hill Lane." Houses come in many different shapes and sizes. Addresses are approximately the same size (street, city, state, and zip). So, while "1600 Pennsylvania Ave." might point to a very big white house and "8347 Undersea Street" might be a one-room shack, both addresses are the same size.

The same is true in C. While things may be big and small, pointers come in one size (relatively small).

Many novice programmers get pointers and their contents confused. In order to limit this problem, all pointer variables in this book end with the extension `_ptr`. You might want to follow this convention in your own programs. Although this notation is not as common as it should be, it is extremely useful.

Many different address variables can point to the same thing. This concept is true for street addresses as well. The following Table 7.2.2 lists the location of important services in a small town.

Table 7.2.2 : Directory of Ed's Town USA

Service (variable name)	**Address (address value)**	**Building (thing)**
Fire Department	1 Main Street	City Hall
Police Station	1 Main Street	City Hall
Planning office	1 Main Street	City Hall
Gas Station	2 Main Street	Ed's Gas Station

In this case, we have a government building that serves many functions. Although it has one address, three different pointers point to it.

As we will see in this chapter, pointers can be used as a quick and simple way to access arrays. In later chapters, we will discover how pointers can be used to create new variables and complex data structures such as linked lists and trees. As you go through the rest of the book, you will be able to understand these data structures as well as create your own.

A pointer is declared by putting an asterisk (`*`) in front of the variable name in the declaration statement:

```
int thing;         /* define a thing */
int *thing_ptr;  /* define a pointer to a thing */
```

7.3 Operators with pointers

The following Table7.3.1 lists the operators used in conjunction with pointers.

Table 7.3.1 : Pointer Operators

Operator	Meaning
*	*Dereference* (given a pointer, get the thing referenced)
&	*Address of* (given a thing, point to it)

The operator ampersand (`&`) returns the address of a thing which is a pointer. The operator asterisk (`*`) returns the object to which a pointer points. These operators can easily cause confusion. The following Table 7.3.2 shows the syntax for the various pointer operators.

Table 7.3.2 : Pointer Operator Syntax

C Code	Description
`thing`	Simple thing (variable)
`&thing`	Pointer to variable `thing`
`thing_ptr`	Pointer to an integer (may or may not be specific integer `thing`)
`*thing_ptr`	Integer

Let's look at some typical uses of the various pointer operators:

```
int thing; /* Declare an integer (a thing) */
thing = 4;
```

The variable `thing` is a thing. The declaration `int thing` does *not* contain an `*`, so `thing` is not a pointer:

```
int *thing_ptr;      /* Declare a pointer to a thing */
```

The variable `thing_ptr` is a pointer. The `*` in the declaration indicates this is a pointer. Also, we have put the extension `_ptr` onto the name:

```
thing_ptr = &thing;  /* Point to the thing */
```

The expression `&thing` is a pointer to a thing. The variable `thing` is an object. The `&` (address of operator) gets the address of an object (a pointer), so `&thing` is a pointer. We then assign this to `thing_ptr`, also of type pointer:

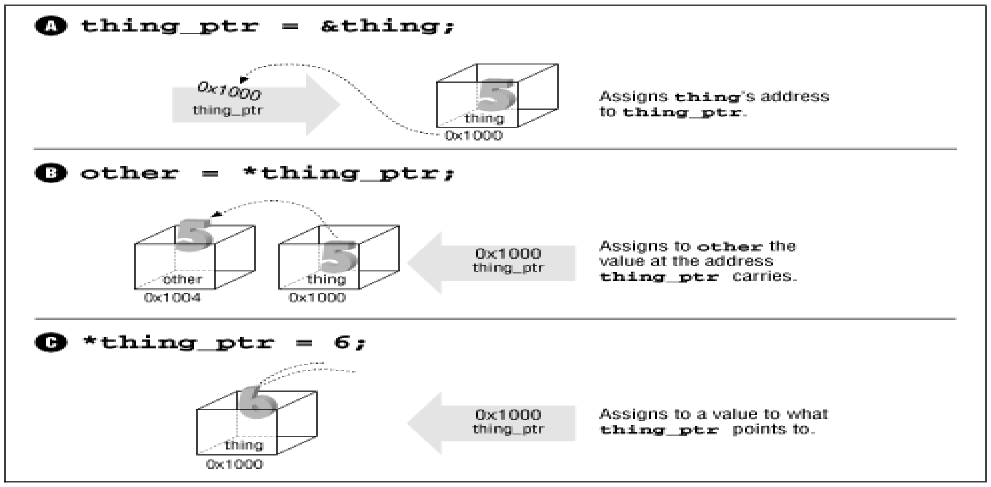
```
*thing_ptr = 5;      /* Set "thing" to 5 */
                      /* We may or may not be pointing */
                      /* to the specific integer "thing" */
```

The expression `*thing_ptr` indicates a thing. The variable `thing_ptr` is a pointer. The `*` (dereference operator) tells C to look at the data pointed to, not the pointer itself. Note that this points to any integer. It may or may not point to the specific variable `thing`.

7.4 Pointer Operations

These pointer operations are summarized in the following Figure 7.4.1.

Figure 7.4.1. Pointer operations



The following examples show how to misuse the pointer operations:

`*thing`

is illegal. It asks C to get the object pointed to by the variable `thing`. Because `thing` is not a pointer, this operation is invalid.

`&thing_ptr`

is legal, but strange. `thing_ptr` is a pointer. The `&` (address of operator) gets a pointer to the object (in this case `thing_ptr`). The result is a pointer to a pointer.

Example illustrates a simple use of pointers. It declares one object, one `thing`, and a pointer, `thing_ptr`. `thing` is set explicitly by the line:

```
thing = 2;
```

The line:

```
thing_ptr = &thing;
```

causes C to set `thing_ptr` to the address of `thing`. From this point on, `thing` and `*thing_ptr` are the same.

Example 7.1 : thing/thing.c

```
#include <stdio.h>
int main()
{
    int   thing_var;  /* define a variable for thing */
    int  *thing_ptr;  /* define a pointer to thing */

    thing_var = 2;      /* assigning a value to thing */
    printf("Thing %d\n", thing_var);

    thing_ptr = &thing_var; /* make the pointer point to thing */
    *thing_ptr = 3;     /* thing_ptr points to thing_var so */
                        /* thing_var changes to 3 */
    printf("Thing %d\n", thing_var);

    /* another way of doing the printf */
    printf("Thing %d\n", *thing_ptr);
    return (0);
}
```

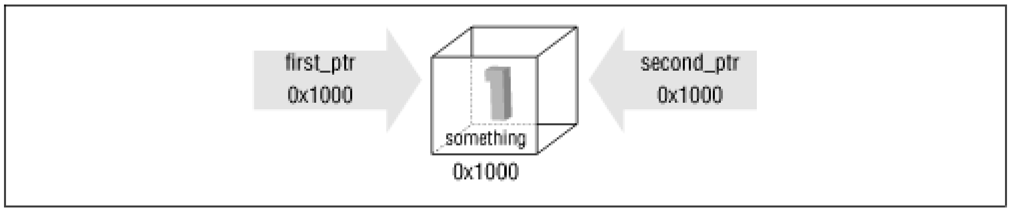
Several pointers can point to the same thing:

```
1:      int something;
2:
3:      int      *first_ptr;     /* one pointer */
4:      int      *second_ptr;    /* another pointer */
5:
6:      something = 1;          /* give the thing a value */
7:
8:      first_ptr = &something;
9:      second_ptr = first_ptr;
```

In line 8, we use the `&` operator to change `something`, a thing, into a pointer that can be assigned to `first_ptr`. Because `first_ptr` and `second_ptr` are both pointers, we can do a direct assignment in line 9.

After executing this program fragment, we have the situation shown in the following Figure 7.4.2.

Figure 7.4.2. Two pointers and a thing



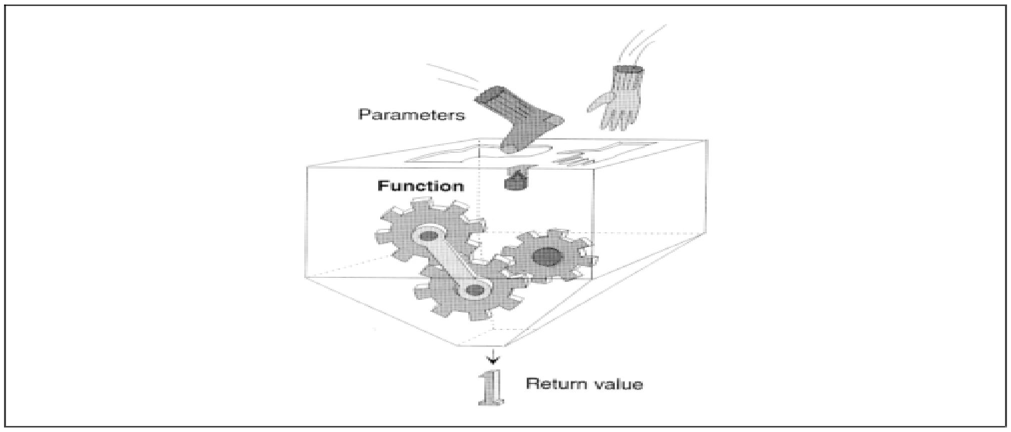
You should note that while we have three variables, there is only one integer (`something`). The following are all equivalent:

```
something = 1;
*first_ptr = 1;
*second_ptr = 1;
```

7.5 Pointers as Function Arguments

C passes parameters using "call by value." That is, the parameters go only one way into the function. The only result of a function is a single return value. This concept is illustrated in the following Figure 7.5.1.

Figure 7.5.1 . Function call



However, pointers can be used to get around this restriction. Imagine that there are two people, Sam and Joe, and whenever they meet, Sam can only talk and Joe can only listen. How is Sam ever going to get any information from Joe? Simple: all Sam has to do is tell Joe, "I want you to leave the answer in the mailbox at 335 West 5th Street."

C uses a similar trick to pass information from a function to its caller. In this Example , `main` wants the function `inc_count` to increment the variable `count`.

Passing it directly would not work, so a pointer is passed instead ("Here's the address of the variable I want you to increment"). Note that the prototype for `inc_count` contains an `int *`. This format indicates that the single parameter given to this function is a pointer to an integer, not the integer itself.

7.5.1 Call by reference

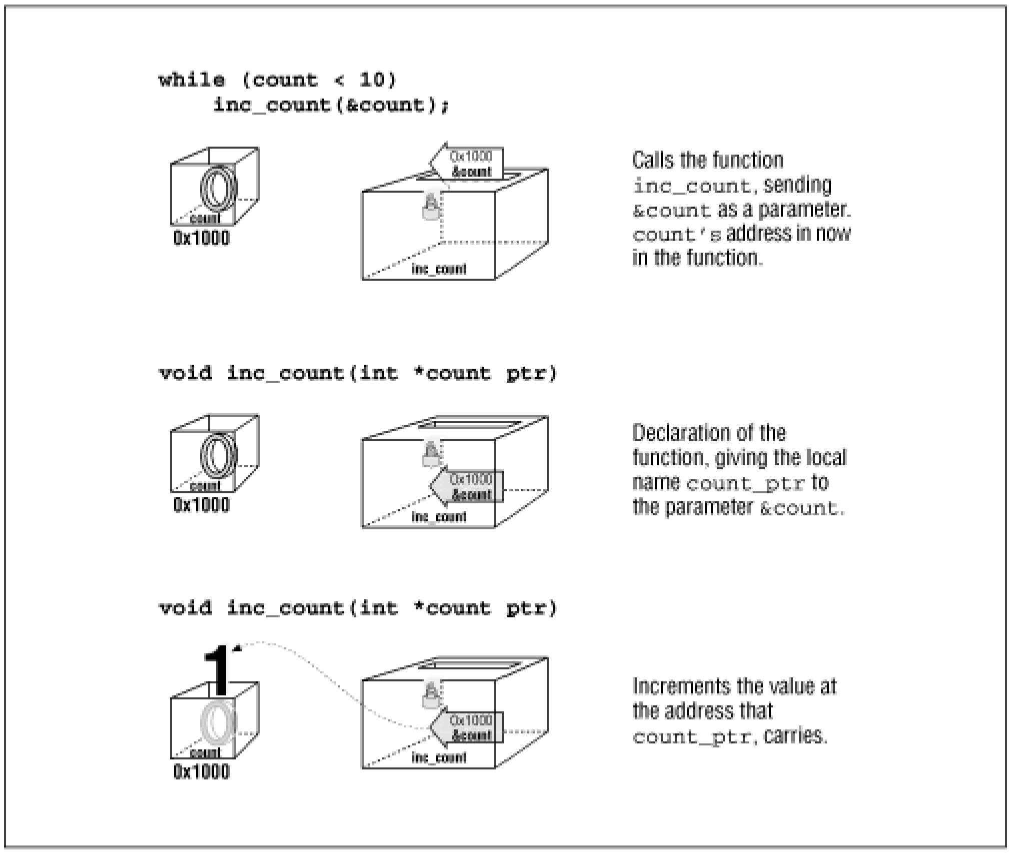
If the arguments are reference to the value then it referred as call by reference otherwise the function is referred as call by value.

Example 7.2 : call/call.c / **Call by Reference** */*

```
#include <stdio.h>
void inc_count(int *count_ptr)
{
    (*count_ptr)++;
}

int main()
{
    int  count = 0;     /* number of times through */

    while (count < 10)
        inc_count(&count);

    return (0);
}
```

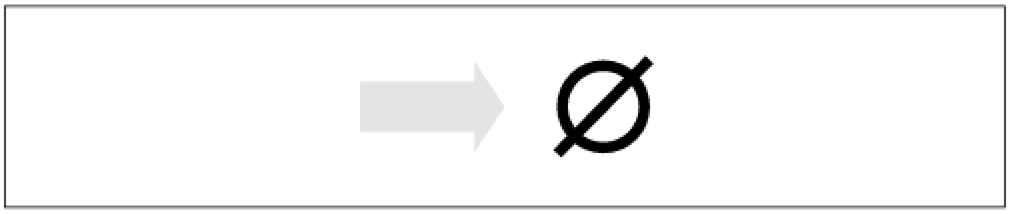
This code is represented graphically in the following Figure 7.5.2 Note that the parameter is not changed, but what it points to is changed.

Figure 7.5.2 . Call of inc_count


while (count < 10)
inc_count(&count);
0x1000
&count
count
0x1000
inc_count
Calls the function inc_count, sending &count as a parameter. count's address in now in the function.
void inc_count(int *count ptr)
Declaration of the function, giving the local name count_ptr to the parameter &count.
void inc_count(int *count ptr)
1
Increments the value at the address that count_ptr, carries.

Finally, there is a special pointer called `NULL`. It points to nothing. (The actual numeric value is 0.) The standard include file, *locale.h*, defines the constant `NULL`. (This file is usually not directly included, but is usually brought in by the include files *stdio.h* or *stdlib.h*.) The `NULL` pointer is represented graphically in Figure 7.5.2 .

Figure 7.5.2. NULL



7.6 const Pointers

Declaring constant pointers is a little tricky. For example, the declaration:

```
const int result = 5;
```

tells C that `result` is a constant so that:

```
result = 10;      /* Illegal */
```

is illegal. The declaration:

```
const char *answer_ptr = "Forty-Two";
```

does *not* tell C that the variable `answer_ptr` is a constant. Instead, it tells C that the data pointed to by `answer_ptr` is a constant. The data cannot be changed, but the pointer can. Again we need to make sure we know the difference between "things" and "pointers to things."

What's `answer_ptr`? A pointer. Can it be changed? Yes, it's just a pointer. What does it point to? A `const char` array. Can the data pointed to by `answer_ptr` be changed? No, it's constant.

In C this is:

```
answer_ptr = "Fifty-One";    /* Legal (answer_ptr is a variable) */
*answer_ptr = 'X';           /* Illegal (*answer_ptr is a constant) */
```

If we put the **const** after the * we tell C that the pointer is constant.

For example:

```
char *const name_ptr = "Test";
```

What's `name_ptr`? It is a constant pointer. Can it be changed? No. What does it point to? A character. Can the data we pointed to by `name_ptr` be changed? Yes.

```
name_ptr = "New";           /* Illegal (name_ptr is constant) */
*name_ptr = 'B';            /* Legal (*name_ptr is a char) */
```

Finally, we can put **const** in both places, creating a pointer that cannot be changed to a data item that cannot be changed:

```
const char *const title_ptr = "Title";
```

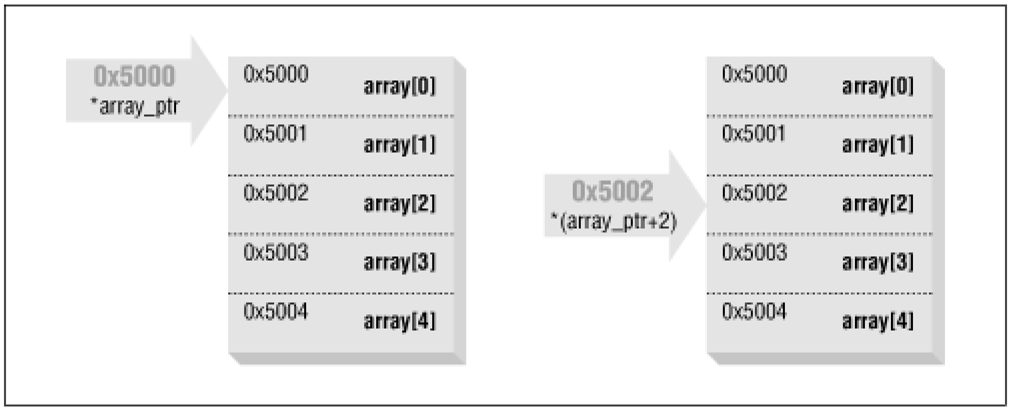
7.7 Pointers and Arrays

C allows pointer arithmetic (addition and subtraction). Suppose we have:

```
char array[5];
char *array_ptr = &array[0];
```

In this example, `*array_ptr` is the same as `array[0]`, `*(array_ptr+1)` is the same as `array[1]`, `*(array_ptr+2)` is the same as `array[2]`, and so on. Note the use of parentheses. Pointer arithmetic is represented graphically in Figure 7.7.1.

Figure 7.7.1. Pointers into an array



However, `(*array_ptr)+1` is *not* the same as `array[1]`. The `+1` is outside the parentheses, so it is added after the dereference. So `(*array_ptr)+1` is the same as `array[0]+1`.

At first glance, this method may seem like a complex way of representing simple array indices. We are starting with simple pointer arithmetic. In later chapters we will use more complex pointers to handle more difficult functions efficiently.

The elements of an array are assigned to consecutive addresses. For example, `array[0]` may be placed at address `0xff000024`. Then `array[1]` would be placed at address `0xff000025`, and so on. This structure means that a pointer can be used to find each element of the array. In the example, it prints out the elements and addresses of a simple character array.

Example7.3 : array-p/array-p.c

```
#include <stdio.h>
 #define ARRAY_SIZE 10   /* Number of characters in array */
/* Array to print */
char array[ARRAY_SIZE] = "0123456789";

int main()
{
    int index;  /* Index into the array */

    for (index = 0; index < ARRAY_SIZE; ++index) {
        printf("&array[index]=0x%p (array+index)=0x%p
array[index]=0x%x\n",
            &array[index], (array+index), array[index]);
    }
    return (0);
}
```

NOTE: When printing pointers, the special conversion `%p` should be used.

When run, this program prints:

```
&array[index] (array+index) array[index]
0x40b0         0x40b0        0x30
0x40b1         0x40b1        0x31
0x40b2         0x40b2        0x32
0x40b3         0x40b3        0x33
0x40b4         0x40b4        0x34
0x40b5         0x40b5        0x35
0x40b6         0x40b6        0x36
0x40b7         0x40b7        0x37
0x40b8         0x40b8        0x38
0x40b9         0x40b9        0x39
```

Characters use one byte, so the elements in a character array will be assigned consecutive addresses. A `short int` font uses two bytes, so in an array of `short int`, the addresses increase by two. Does this mean that `array+1` will not work for anything other than characters? No. C automatically scales pointer arithmetic so that it works correctly. In this case, `array+1` will point to element number 1.

C provides a shorthand for dealing with arrays. Rather than writing:

```
array_ptr = &array[0];
```

we can write:

```
array_ptr = array;
```

C blurs the distinction between pointers and arrays by treating them in the same manner in many cases. Here we use the variable `array` as a pointer, and C automatically does the necessary conversion.

Example counts the number of elements that are nonzero and stops when a zero is found. No limit check is provided, so there must be at least one zero in the array.

Example 7.4 : ptr2.c

```
#include <stdio.h>

int array[] = {4, 5, 8, 9, 8, 1, 0, 1, 9, 3};
int index;

int main()
{
    index = 0;
    while (array[index] != 0)
        ++index;
     printf("Number of elements before zero %d\n",
                 index);
    return (0);
}
```

Next Example is a version of previous Example that uses pointers.

Example7.5 : ptr3/ptr3.c

```
#include <stdio.h>

int array[] = {4, 5, 8, 9, 8, 1, 0, 1, 9, 3};
int *array_ptr;

int main()
{
    array_ptr = array;

    while ((*array_ptr) != 0)
        ++array_ptr;

    printf("Number of elements before zero %d\n",
                  array_ptr - array);
    return (0);
}
```

Notice that when we wish to examine the data in the array, we use the dereference operator (`*`). This operator is used in the statement:

```
while ((*array_ptr) != 0)
```

When we wish to change the pointer itself, no other operator is used. For example, the line:

```
++array_ptr;
```

increments the pointer, not the data.

In the above example uses the expression `(array[index] != 0)`. This expression requires the compiler to generate an index operation, which takes longer than a simple pointer dereference, `((*array_ptr) != 0)`.

The expression at the end of this program, `array_ptr - array`, computes how far `array_ptr` is into the array.

When passing an array to a procedure, C will automatically change the array into a pointer. In fact, if you put `&` before the array, C will issue a warning. The following example illustrates the various ways in which an array can be passed to a subroutine.

Example 7.6 : init-a/init-a.c (continued)

```
#define MAX 10  /* Size of the array */
/********************************************************
 * init_array_1 -- Zeroes out an array.                 *
 *                                                      *
 * Parameters                                           *
 *      data -- The array to zero out.                  *
 ********************************************************/
void init_array_1(int data[])
{
    int  index;

    for (index = 0; index < MAX; ++index)
        data[index] = 0;
}

/********************************************************
 * init_array_2 -- Zeroes out an array.                 *
 *                                                      *
 * Parameters                                           *
 *      data_ptr -- Pointer to array to zero.           *
 ********************************************************/
void init_array_2(int *data_ptr)
{
    int index;

    for (index = 0; index < MAX; ++index)
        *(data_ptr + index) = 0;
}
int main()
{
    int  array[MAX];

    void init_array_1();
    void init_array_2();

    /* one way of initializing the array */
    init_array_1(array);

    /* another way of initializing the array */
    init_array_1(&array[0]);

    /* works, but the compiler generates a warning */
    init_array_1(&array);

    /* Similar to the first method but  */
    /*    function is different */
    init_array_2(array);

    return (0);
}
```

7.8 How Not to Use Pointers

The major goal of this book is to teach you how to create clear, readable, maintainable code. Unfortunately, not everyone has read this book and some people still believe that you should make your code as compact as possible. This belief can result in programmers using the ++ and -- operators inside other statements.

Example shows several examples in which pointers and the increment operator are used together.

Example7.7 : Bad Pointer Usage

```
/* This program shows programming practices that should **NOT** be used
*/
/* Unfortunately, too many programmers use them */
int array[10];    /* An array for our data */
int main()
{
    int *data_ptr;   /* Pointer to the data */
    int value;       /* A data value */

    data_ptr = &array[0];/* Point to the first element */
    value = *data_ptr++;  /* Get element #0, data_ptr points to element
                                       #1 */
    value = *++data_ptr;  /* Get element #2, data_ptr points to element
                                       #2 */
    value = ++*data_ptr; /* Increment element #2, return its value */
                         /* Leave data_ptr alone */
```

To understand each of these statements, you must carefully dissect each expression to discover its hidden meaning. When I do maintenance programming, I don't want to have to worry about hidden meanings, so please don't code like this, and shoot anyone who does.

This example is a little extreme, but it illustrates how side effects can easily become confusing.

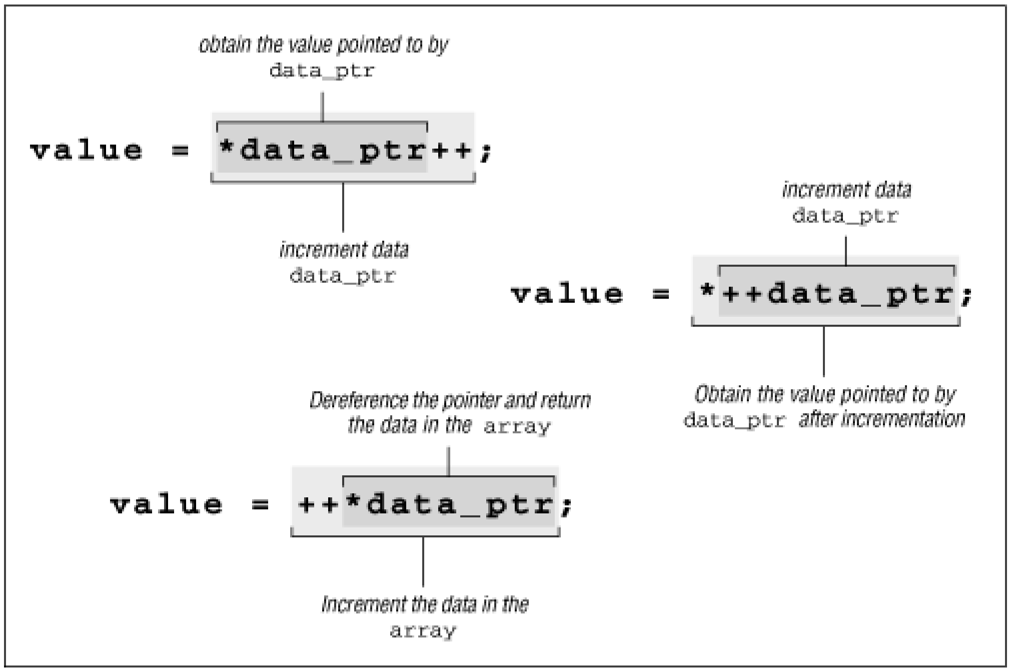
Example is an example of the code you're more likely to run into. The program copies a string from the source (p) to the destination (q).

```
void copy_string(char *p, char *q)
{
    while (*p++ = *q++);
}
```

Given time, a good programmer will decode this. However, understanding the program is much easier when we are a bit more verbose, as in Example .

These statements are dissected in Figure 7.8.1 .

Figure 7.8.1. Pointer operations dissected


obtain the value pointed to by data_ptr
value = *data_ptr++;
increment data data_ptr
increment data data_ptr
value = *++data_ptr;
Obtain the value pointed to by data_ptr after incrementation
Dereference the pointer and return the data in the array
value = ++*data_ptr;
Increment the data in the array

Example7.8 : Readable Use of Pointers

```
/********************************************************
 * copy_string -- Copies one string to another.         *
 *                                                      *
 * Parameters                                           *
 *      dest -- Where to put the string                 *
 *      source -- Where to get it                       *
 ********************************************************/
void copy_string(char *dest, char *source)
{
    while (1) {
        *dest = *source;

        /* Exit if we copied the end of string */
        if (*dest == '\0')
            return;

        ++dest;
        ++source;
    }
}
```

7.9 Using Pointers to Split a String

Suppose we are given a string of the form "Last/First." We want to split this into two strings, one containing the first name and one containing the last name.

We need a function to find the slash in the name. The standard function `strchr` performs this job for us. In this program, we have chosen to duplicate this function to show you how it works.

This function takes a pointer to a string (`string_ptr`) and a character to find (`find`) as its arguments. It starts with a **while** loop that will continue until we find the character we are looking for (or we are stopped by some other code below).

```
while (*string_ptr != find) {
```

Next we test to see if we've run out of string. In this case, our pointer (`string_ptr`) points to the end-of-string character. If we have reached the end of string before finding the character, we return `NULL`:

```
if (*string_ptr == '\0')
    return (NULL);
```

If we get this far, we have not found what we are looking for, and are not at the end of the string. So we move the pointer to the next character, and return to the top of the loop to try again:

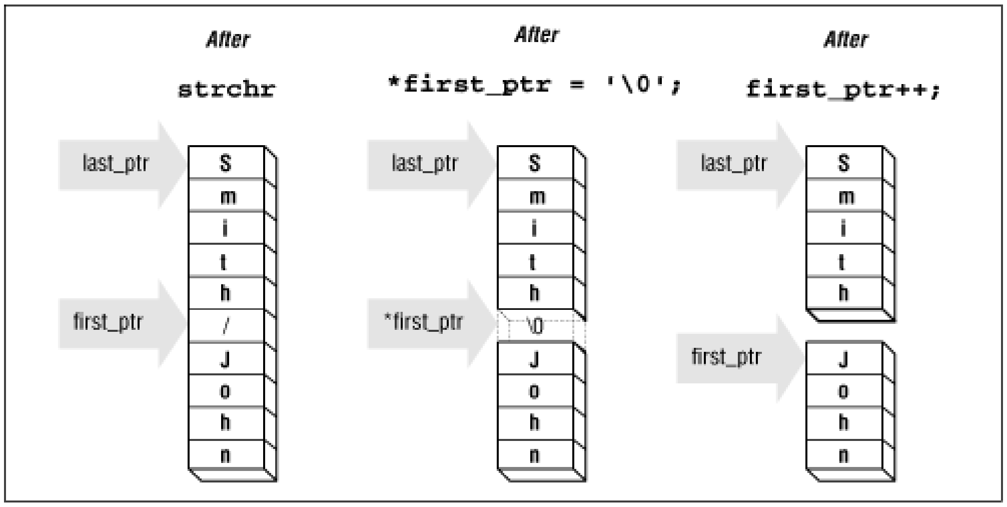
```
++string_ptr;
}
```

Our main program reads in a single line, stripping the newline character from it. The function `my_strchr` is called to find the location of the slash (/).

At this point, `last_ptr` points to the first character of the last name and `first_ptr` points to slash. We then split the string by replacing the slash (/) with an end of string (NUL or `\0`). Now `last_ptr` points to just the last name and `first_ptr` points to a null string. Moving `first_ptr` to the next character makes it point to the beginning of the first name.

The sequence of steps in splitting the string is illustrated in Figure 7.9.1.

Figure 7.9.1. Splitting a string



Following example contains the full program, which demonstrates how pointers and character arrays can be used for simple string processing.

Example 7.8: split/split.c (continued)

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

/*********************************************************
 * my_strchr -- Finds a character in a string.           *
 *      Duplicate of a standard library function,        *
 *      put here for illustrative purposes               *
 *                                                       *
 * Parameters                                            *
 *      string_ptr -- String to look through.            *
 *      find -- Character to find.                       *
 *                                                       *
 * Returns                                               *
 *      pointer to 1st occurrence of character           *
 *      in string or NULL for error.                     *
 *********************************************************/
char *my_strchr(char * string_ptr, char find)
{
    while (*string_ptr != find) {

       /* Check for end */

       if (*string_ptr == '\0')
           return (NULL);       /* not found */

        ++string_ptr;
    }
    return (string_ptr);        /* Found */
}

int main()
{
    char line[80];      /* The input line */
    char *first_ptr;    /* pointer to the first name */
    char *last_ptr;     /* pointer to the last name */

    fgets(line, sizeof(line), stdin);

    /* Get rid of trailing newline */
    line[strlen(line)-1] = '\0';

    last_ptr = line;    /* last name is at beginning of line */

    first_ptr = my_strchr(line, '/');      /* Find slash */

    /* Check for an error */
    if (first_ptr == NULL) {
        fprintf(stderr,
            "Error: Unable to find slash in %s\n", line);
        exit (8);
    }

    *first_ptr = '\0';  /* Zero out the slash */

    ++first_ptr;        /* Move to first character of name */

    printf("First:%s Last:%s\n", first_ptr, last_ptr);
    return (0);
}
```

Question : Example is supposed to print out:

```
Name: tmp1 but instead, we get:
Name: !_@$#ds80
(Your results may vary.) Why?
```

Example 7.9: tmp-name/tmp-name.c

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

/********************************************************
 * tmp_name -- Return a temporary filename.             *
 *                                                      *
 * Each time this function is called, a new name will   *
 * be returned.                                         *
 *                                                      *
 * Returns                                              *
 *      Pointer to the new filename.                    *
 *******************************************************/
char *tmp_name(void)
{
    char name[30];      /* The name we are generating */
    static int sequence = 0;    /* Sequence number for last digit */

    ++sequence; /* Move to the next filename */

    strcpy(name, "tmp");

    /* But in the sequence digit */
    name[3] = sequence + '0';

    /* End the string */
    name[4] = '\0';

    return(name);
}

int main()
{
    char *tmp_name(void);       /* Get name of temporary file */

    printf("Name: %s\n", tmp_name());
    return(0);
}
```

7.10 Pointers and Structures

Consider the example, we can defined a structure for a mailing list as follows

```
struct mailing {
    char name[60];    /* last name, first name */
    char address1[60];/* two lines of street address */
    char address2[60];
    char city[40];
    char state[2];    /* two-character abbreviation */
    long int zip;     /* numeric zip code */
} list[MAX_ENTRIES];
```

Mailing lists must frequently be sorted by name and zip code. We could sort the entries themselves, but each entry is 226 bytes long. That's a lot of data to move around. One way around this problem is to declare an array of pointers, and then sort the pointers:

```
/* Pointer to the data */
struct mailing *list_ptrs[MAX_ENTRIES];
int current;    /* current mailing list entry */

    for (current = 0; current = number_of_entries; ++current)
        list_ptrs[current] = &list[current];
    /* Sort list_ptrs by zip code */
```

Now, instead of having to move a 226-byte structure around, we are moving 4-byte pointers. Our sorting is much faster. Imagine that you had a warehouse full of big heavy boxes and you needed to locate any box quickly. You could put them in alphabetical order, but that would require a lot of moving. Instead, you assign each location a number, write down the name and number on index cards, and sort the cards by name.

7.10.1 Command-Line Arguments

The procedure `main` actually takes two arguments. They are called `argc` and `argv`[2]:

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

(If you realize that the arguments are in alphabetical order, you can easily remember which one comes first.)

The parameter `argc` is the number of arguments on the command line (including the program name). The array `argv` contains the actual arguments. For example, if the program `args` were run with the command line:

```
args this is a test
```

then:

```
   argc =     5
argv[0] =    "args"
argv[1] =    "this"
argv[2] =    "is"
argv[3] =    "a"
argv[4] =    "test"
argv[5] =    NULL
```

NOTE: The UNIX shell expands wildcard characters like *, ?, and [ ] before sending the command line to the program. See your `sh` or `csh` manual for details.

Turbo C++ and Borland C++ expand wildcard characters if the file *WILDARG.OBJ* is linked with your program. See the manual for details.

Almost all UNIX commands use a standard command-line format. This standard has carried over into other environments. A standard UNIX command has the form:

```
command options file1 file1 file3 ...
```

Options are preceded by a dash (-) and are usually a single letter. For example, the option `-v` might turn on verbose mode for a particular command. If the option takes a parameter, it follows the letter. For example, the option `-m1024` sets the maximum number of symbols to 1024 and `-ooutfile` sets the output filename to *outfile*.

Let's look at writing a program that can read the command-line arguments and act accordingly. This program formats and prints files. Part of the documentation for the program is given here:

```
print_file [-v] [-llength] [-oname] [file1] [file2] ...
```

where:

-v

specifies verbose options; turns on a lot of progress information messages

-llength

sets the page size to length lines (default = 66)

-oname

sets the output file to name (default = print.out)

file1, file2, ...

is a list of files to print. If no files are specified, the file print.in is printed.

We can use a **while** loop to cycle through the command-line options. The actual loop is:

```
while ((argc > 1) && (argv[1][0] == '-')) {
```

One argument always exists: the program name. The expression `(argc > 1)` checks for additional arguments. The first one is numbered 1. The first character of the first argument is `argv[1][0]`. If this is a dash, we have an option.

At the end of the loop is the code:

```
    --argc;
    ++argv;
}
```

This consumes an argument. The number of arguments is decremented to indicate one less option, and the pointer to the first option is incremented, shifting the list to the left one place. (Note: after the first increment, `argv[0]` no longer points to the program name.)

The **switch** statement is used to decode the options. Character 0 of the argument is the dash (-). Character 1 is the option character, so we use the expression:

```
switch (argv[1][1]) {
```

to decode the option.

The option `-v` has no arguments; it just causes a flag to be set.

The option `-o` takes a filename. Rather than copy the whole string, we set the character pointer `out_file` to point to the name part of the string. By this time we know the following:

```
argv[1][0]   ='-'
argv[1][1]   ='o'
argv[1][2]   = first character of the filename
```

We set `out_file` to point to the string with the statement:

```
out_file = &argv[1][2];
```

The address of operator (`&`) is used to get the address of the first character in the output filename. This process is appropriate because we are assigning the address to a character pointer named `out_file`.

The `-l` option takes an integer argument. The library function `atoi` is used to convert the string into an integer. From the previous example, we know that `argv[1][2]` is the first character of the string containing the number. This string is passed to `atoi`.

Finally, all the options are parsed and we fall through to the processing loop. This merely executes the function `do_file` for each file argument. The following example print.c contains the `print` program.

This is one way of parsing the argument list. The use of the **while** loop and **switch** statement is simple and easy to understand. This method does have a limitation. The argument must immediately follow the options. For example, `-odata.out` will work, but `"-o data.out"` will not. An improved parser would make the program more friendly, but the techniques described here work for simple programs.

Example7.10 : print/print.c (continued)

```
[File: print/print.c]
/********************************************************
 * Program: Print                                       *
 *                                                      *
 * Purpose:                                             *
 *      Formats files for printing.                     *
 *                                                      *
 * Usage:                                               *
 *      print [options] file(s)                         *
 *                                                      *
 * Options:                                             *
 *      -v              Produces verbose messages.      *
 *      -o<file>        Sends output to a file          *
 *                      (default=print.out).            *
 *      -l<lines>       Sets the number of lines/page   *
 *                      (default=66).                   *
 ********************************************************/
#include <stdio.h>
#include <stdlib.h>

int verbose = 0;         /* verbose mode (default = false) */
char *out_file = "print.out";   /* output filename */
char *program_name;      /* name of the program (for errors) */
int line_max = 66;       /* number of lines per page */

/********************************************************
 * do_file -- Dummy routine to handle a file.           *
 *                                                      *
 * Parameter                                            *
 *      name -- Name of the file to print.              *
 ********************************************************/
void do_file(char *name)
{
    printf("Verbose %d Lines %d Input %s Output %s\n",
        verbose, line_max, name, out_file);
}
/*********************************************************
 * usage -- Tells the user how to use this program and   *
 *              exit.                                    *
 ********************************************************/
void usage(void)
{
    fprintf(stderr,"Usage is %s [options] [file-list]\n",
                                program_name);
    fprintf(stderr,"Options\n");
    fprintf(stderr,"  -v          verbose\n");
    fprintf(stderr,"  -l<number>  Number of lines\n");
    fprintf(stderr,"  -o<name>    Set output filename\n");
    exit (8);
}
int main(int argc, char *argv[])
{
    /* save the program name for future use */
    program_name = argv[0];

    /*
     * loop for each option
     *   Stop if we run out of arguments
     *   or we get an argument without a dash
     */
    while ((argc > 1) && (argv[1][0] == '-')) {
        /*
         * argv[1][1] is the actual option character
         */
        switch (argv[1][1]) {
            /*
             * -v verbose
             */
            case 'v':
                verbose = 1;
                break;
            /*
             * -o<name>  output file
             *    [0] is the dash
             *    [1] is the "o"
             *    [2] starts the name
             */
            case 'o':
                out_file = &argv[1][2];
                break;
            /*
             * -l<number> set max number of lines
             */
            case 'l':
                line_max = atoi(&argv[1][2]);
                break;
            default:
                fprintf(stderr,"Bad option %s\n", argv[1]);
                usage();
        }
        /*
         * move the argument list up one
         * move the count down one
         */
        ++argv;
        --argc;
    }

    /*
     * At this point, all the options have been processed.
     * Check to see if we have no files in the list.
     * If no files exist, we need to process just standard input
stream.
     */
    if (argc == 1) {
        do_file("print.in");
    } else {
        while (argc > 1) {
          do_file(argv[1]);
          ++argv;
          --argc;
        }
    }
    return (0);
}
```

7.11 Let us Sum up.

In this lesson we discussed about :

Pointers as Function Arguments
Usage of const Pointers
Pointers and Arrays
How Not to Use Pointers
Using Pointers to Split a String
Pointers and Structures

7.12 Points for discussion

1. What are all the advantages of using pointers?
2. Define call by reference and call by value
3. How to set pointer to function?
4. Explan rules for using pointers

7.13 Check your progress

Define call by reference and call by value

If the arguments are reference to the value then it referred as call by reference otherwise the function is referred as call by value. Give appropriate example.

Explain any twoPurposse of using pointers.

1. Pointers will increase the spead of execution.
2. It saves memory also
3. We can access the variable, even though it may declare outside the function.

7.14 Lesson-end Activities

1. What do you mean by address of and value of operators?
2. How pointer will increase your program efficiency?

7.15 References

1.Peter Aitken, Teach Yourself C in 21 Days, Fourth Edition, Sams Publisher.
2.Brian W. Kernighan and Dennis M. Ritchie, The C programming Language, Prentice-Hall in 1988
3.E.Balagursamy, Programming in Ansi C, TATA McGraw Hill
http://sysprog.net/
http://www.mycplus.com/cplus

LESSON - 8 : PREPROCESSORS

CONTENTS

8.0 Aims and Objectives

The aim of this lesson is to make the readers to understand about a facility called C-Preprocessors and its applications.

8.1 Introduction

The C preprocessor is a **macro processor** that is used automatically by the C compiler to transform your program before actual compilation. It is called a macro processor because it allows you to define **macros**, which are brief abbreviations for longer constructs.

The C preprocessor provides four separate facilities that you can use as you see fit:

- Inclusion of header files. These are files of declarations that can be substituted into your program.
- Macro expansion. You can define **macros**, which are abbreviations for arbitrary fragments of C code, and then the C preprocessor will replace the macros with their definitions throughout the program.
- Conditional compilation. Using special preprocessing directives, you can include or exclude parts of the program according to various conditions.
- Line control. If you use a program to combine or rearrange source files into an intermediate file which is then compiled, you can use line control to inform the compiler of where each source line originally came from.

C preprocessors vary in some details. This manual discusses the GNU C preprocessor, the C Compatible Compiler Preprocessor. The GNU C preprocessor provides a superset of the features of ANSI Standard C.

ANSI Standard C requires the rejection of many harmless constructs commonly used by today's C programs. Such incompatibility would be inconvenient for users, so the GNU C preprocessor is configured to accept these constructs by default. Strictly speaking, to get ANSI Standard C, you must use the options `` `-trigraphs' ``, `` `-undef' `` and `` `-pedantic' ``, but in practice the consequences of having strict ANSI Standard C make it undesirable to do this.

The C preprocessor is designed for C-like languages; you may run into problems if you apply it to other kinds of languages, because it assumes that it is dealing with C. For example, the C preprocessor sometimes outputs extra white space to avoid inadvertent C token concatenation, and this may cause problems with other languages.

8.2 Preprocessing Directives

Most preprocessor features are active only if you use preprocessing directives to request their use. Preprocessing directives are lines in your program that start with `` `#' ``. The `` `#' `` is followed by an identifier that is the **directive name**. For example, `` `#define' `` is the directive that defines a macro. White space is also allowed before and after the `` `#' ``.

The set of valid directive names is fixed. Programs cannot define new preprocessing directives. Some directive names require arguments; these make up the rest of the directive line and must be separated from the directive name by whitespace. For example, `` `#define' `` must be followed by a macro name and the intended expansion of the macro

A preprocessing directive cannot be more than one line in normal circumstances. It may be split cosmetically with Backslash-Newline, but that has no effect on its meaning. Comments containing Newlines can also divide the directive into multiple lines, but the comments are changed to Spaces before the directive is interpreted. The only way a significant Newline can occur in a preprocessing directive is within a string constant or character constant. Note that most C compilers that might be applied to the output from the preprocessor do not accept string or character constants containing Newlines.

The `` `#' `` and the directive name cannot come from a macro expansion. For example, if `` `foo' `` is defined as a macro expanding to `` `define' ``, that does not make `` `#foo' `` a valid preprocessing directive.

8.2.1 Header Files

A header file is a file containing C declarations and macro definitions to be shared between several source files. You request the use of a header file in your program with the C preprocessing directive `` `#include' ``.

8.2.2 Uses of Header Files

Header files serve two kinds of purposes.

- System header files declare the interfaces to parts of the operating system. You include them in your program to supply the definitions and declarations you need to invoke system calls and libraries.
- Your own header files contain declarations for interfaces between the source files of your program. Each time you have a group of related declarations and macro

definitions all or most of which are needed in several different source files, it is a good idea to create a header file for them.

Including a header file produces the same results in C compilation as copying the header file into each source file that needs it. But such copying would be time-consuming and error-prone. With a header file, the related declarations appear in only one place. If they need to be changed, they can be changed in one place, and programs that include the header file will automatically use the new version when next recompiled. The header file eliminates the labor of finding and changing all the copies as well as the risk that a failure to find one copy will result in inconsistencies within a program.

The usual convention is to give header files names that end with `` `.h' ``. Avoid unusual characters in header file names, as they reduce portability.

8.2.3 The `#include' Directive

Both user and system header files are included using the preprocessing directive `` `#include' ``. It has three variants:

`#include <`*`file`*`>`

This variant is used for system header files. It searches for a file named *file* in a list of directories specified by you, then in a standard list of system directories. You specify directories to search for header files with the command option `` `-I' ``. The option `` `-nostdinc' `` inhibits searching the standard system directories; in this case only the directories you specify are searched. The parsing of this form of `` `#include' `` is slightly special because comments are not recognized within the `` `<...>' ``. Thus, in `` `#include <x/*y>' `` the `` `/*' `` does not start a comment and the directive specifies inclusion of a system header file named `` `x/*y' ``. Of course, a header file with such a name is unlikely to exist on Unix, where shell wildcard features would make it hard to manipulate. The argument *file* may not contain a `` `>' `` character. It may, however, contain a `` `<' `` character.

`#include "`*`file`*`"`

This variant is used for header files of your own program. It searches for a file named *file* first in the current directory, then in the same directories used for system header files. The current directory is the directory of the current input file. It is tried first because it is presumed to be the location of the files that the current input file refers to. (If the `` `-I-' `` option is used, the special treatment of the current directory is inhibited.) The argument *file* may not contain `` `"' `` characters. If backslashes occur within *file*, they are considered ordinary text characters, not escape characters. None of the character escape sequences appropriate to string constants in C are processed. Thus, `` `#include "x\n\\y"' `` specifies a filename containing three backslashes. It is not clear why this behavior is ever useful, but the ANSI standard specifies it.

`#include `*`anything else`*

This variant is called a **computed #include**. Any `` `#include' `` directive whose argument does not fit the above two forms is a computed include. The text *anything else* is checked for macro calls, which are expanded . When this is done, the result must fit one of the above two variants--in particular, the expanded text must in the end be surrounded by either quotes or angle braces. This feature allows you to define a macro which controls the file name to be used at a later point in the program. One application of this is to allow a site-specific configuration file for your program to specify the names of the system include files to be used. This can help in porting the program to various operating systems in which the necessary system header files are found in different places.

How `#include' Works

The `` `#include' `` directive works by directing the C preprocessor to scan the specified file as input before continuing with the rest of the current file. The output from the preprocessor contains the output already generated, followed by the output resulting from the included file, followed by the output that comes from the text after the `` `#include' `` directive. For example, given a header file `` `header.h' `` as follows,

```
char *test ();
```

and a main program called `` `program.c' `` that uses the header file, like this,

```
int x;
#include "header.h"

main ()
{
  printf (test ());
}
```

the output generated by the C preprocessor for `` `program.c' `` as input would be

```
int x;
char *test ();

main ()
{
  printf (test ());
}
```

Included files are not limited to declarations and macro definitions; those are merely the typical uses. Any fragment of a C program can be included from another file. The include file could even contain the beginning of a statement that is concluded in the containing file, or the end of a statement that was started in the including file. However, a comment or a string or character constant may not start in the included file and finish in the including file. An unterminated comment, string constant or character constant in an included file is considered to end (with an error message) at the end of the file.

It is possible for a header file to begin or end a syntactic unit such as a function definition, but that would be very confusing, so don't do it.

The line following the `` `#include' `` directive is always treated as a separate line by the C preprocessor even if the included file lacks a final newline.

Once-Only Include Files

Very often, one header file includes another. It can easily result that a certain header file is included more than once. This may lead to errors, if the header file defines structure types or typedefs, and is certainly wasteful. Therefore, we often wish to prevent multiple inclusion of a header file.

The standard way to do this is to enclose the entire real contents of the file in a conditional, like this:

```
#ifndef FILE_FOO_SEEN
#define FILE_FOO_SEEN

the entire file

#endif /* FILE_FOO_SEEN */
```

The macro `FILE_FOO_SEEN` indicates that the file has been included once already. In a user header file, the macro name should not begin with `` `_' ``. In a system header file, this name should begin with `` `__' `` to avoid conflicts with user programs. In any kind of header file, the macro name should contain the name of the file and some additional text, to avoid conflicts with other header files.

The GNU C preprocessor is programmed to notice when a header file uses this particular construct and handle it efficiently. If a header file is contained entirely in a `` `#ifndef' `` conditional, then it records that fact. If a subsequent `` `#include' `` specifies the same file, and the macro in the `` `#ifndef' `` is already defined, then the file is entirely skipped, without even reading it.

There is also an explicit directive to tell the preprocessor that it need not include a file more than once. This is called `` `#pragma once' ``, and was used *in addition to* the `` `#ifndef' `` conditional around the contents of the header file. `` `#pragma once' `` is now obsolete and should not be used at all.

In the Objective C language, there is a variant of `` `#include' `` called `` `#import' `` which includes a file, but does so at most once. If you use `` `#import' `` *instead of* `` `#include' ``, then you don't need the conditionals inside the header file to prevent multiple execution of the contents.

`` `#import' `` is obsolete because it is not a well designed feature. It requires the users of a header file--the applications programmers--to know that a certain header file should only be included once. It is much better for the header file's implementor to write the file so that users don't need to know this. Using `` `#ifndef' `` accomplishes this goal.

8.3 Macros

A macro is a sort of abbreviation which you can define once and then use later. There are many complicated features associated with macros in the C preprocessor.

8.3.1 Simple Macros

A **simple macro** is a kind of abbreviation. It is a name which stands for a fragment of code. Some people refer to these as **manifest constants**.

Before you can use a macro, you must **define** it explicitly with the `` `#define' `` directive. `` `#define' `` is followed by the name of the macro and then the code it should be an abbreviation for. For example,

```
#define BUFFER_SIZE 1020
```

defines a macro named `` `BUFFER_SIZE' `` as an abbreviation for the text `` `1020' ``. If somewhere after this `` `#define' `` directive there comes a C statement of the form

```
foo = (char *) xmalloc (BUFFER_SIZE);
```

then the C preprocessor will recognize and **expand** the macro `` `BUFFER_SIZE' ``, resulting in

```
foo = (char *) xmalloc (1020);
```

The use of all upper case for macro names is a standard convention. Programs are easier to read when it is possible to tell at a glance which names are macros.

Normally, a macro definition must be a single line, like all C preprocessing directives. (You can split a long macro definition cosmetically with Backslash-Newline.) There is one exception: Newlines can be included in the macro definition if within a string or character constant. This is because it is not possible for a macro definition to contain an unbalanced quote character; the definition automatically extends to include the matching quote character that ends the string or character constant. Comments within a macro definition may contain Newlines, which make no difference since the comments are entirely replaced with Spaces regardless of their contents.

Aside from the above, there is no restriction on what can go in a macro body. Parentheses need not balance. The body need not resemble valid C code. (But if it does not, you may get error messages from the C compiler when you use the macro.)

The C preprocessor scans your program sequentially, so macro definitions take effect at the place you write them. Therefore, the following input to the C preprocessor

```
foo = X;
#define X 4
```

```
bar = X;
```

produces as output

```
foo = X;

bar = 4;
```

After the preprocessor expands a macro name, the macro's definition body is appended to the front of the remaining input, and the check for macro calls continues. Therefore, the macro body can contain calls to other macros. For example, after

```
#define BUFSIZE 1020
#define TABLESIZE BUFSIZE
```

the name `` `TABLESIZE' `` when used in the program would go through two stages of expansion, resulting ultimately in `` `1020' ``.

This is not at all the same as defining `` `TABLESIZE' `` to be `` `1020' ``. The `` `#define' `` for `` `TABLESIZE' `` uses exactly the body you specify--in this case, `` `BUFSIZE' ``---and does not check to see whether it too is the name of a macro. It's only when you *use* `` `TABLESIZE' `` that the result of its expansion is checked for more macro names.

8.3.2 Macros with Arguments

A simple macro always stands for exactly the same text, each time it is used. Macros can be more flexible when they accept **arguments**. Arguments are fragments of code that you supply each time the macro is used. These fragments are included in the expansion of the macro according to the directions in the macro definition. A macro that accepts arguments is called a **function-like macro** because the syntax for using it looks like a function call.

To define a macro that uses arguments, you write a `` `#define' `` directive with a list of **argument names** in parentheses after the name of the macro. The argument names may be any valid C identifiers, separated by commas and optionally whitespace. The open-parenthesis must follow the macro name immediately, with no space in between.

For example, here is a macro that computes the minimum of two numeric values, as it is defined in many C programs:

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
```

(This is not the best way to define a "minimum" macro in GNU C.

To use a macro that expects arguments, you write the name of the macro followed by a list of **actual arguments** in parentheses, separated by commas. The number of actual arguments you give must match the number of arguments the macro expects. Examples of use of the macro `` `min' `` include `` `min (1, 2)' `` and `` `min (x + 28, *p)' ``.

The expansion text of the macro depends on the arguments you use. Each of the argument names of the macro is replaced, throughout the macro definition, with the corresponding actual argument. Using the same macro `` `min' `` defined above, `` `min (1, 2)' `` expands into

```
((1) < (2) ? (1) : (2))
```

where `` `1' `` has been substituted for `` `X' `` and `` `2' `` for `` `Y' ``.

Likewise, `` `min (x + 28, *p)' `` expands into

```
((x + 28) < (*p) ? (x + 28) : (*p))
```

Parentheses in the actual arguments must balance; a comma within parentheses does not end an argument. However, there is no requirement for brackets or braces to balance, and they do not prevent a comma from separating arguments. Thus,

```
macro (array[x = y, x + 1])
```

passes two arguments to `macro`: `` `array[x = y' `` and `` `x + 1]' ``. If you want to supply `` `array[x = y, x + 1]' `` as an argument, you must write it as `` `array[(x = y, x + 1)]' ``, which is equivalent C code.

After the actual arguments are substituted into the macro body, the entire result is appended to the front of the remaining input, and the check for macro calls continues. Therefore, the actual arguments can contain calls to other macros, either with or without arguments, or even to the same macro. The macro body can also contain calls to other macros. For example, `` `min (min (a, b), c)' `` expands into this text:

```
((((a) < (b) ? (a) : (b))) < (c)
 ? (((a) < (b) ? (a) : (b)))
 : (c))
```

(Line breaks shown here for clarity would not actually be generated.)

If a macro `foo` takes one argument, and you want to supply an empty argument, you must write at least some whitespace between the parentheses, like this: `` `foo ( )' ``. Just `` `foo ()' `` is providing no arguments, which is an error if `foo` expects an argument. But `` `foo0 ()' `` is the correct way to call a macro defined to take zero arguments, like this:

```
#define foo0() ...
```

If you use the macro name followed by something other than an open-parenthesis (after ignoring any spaces, tabs and comments that follow), it is not a call to the macro, and the preprocessor does not change what you have written. Therefore, it is possible for the same name to be a variable or function in your program as well as a macro, and you can choose in each instance whether to refer to the macro (if an actual argument list follows) or the variable or function (if an argument list does not follow).

Such dual use of one name could be confusing and should be avoided except when the two meanings are effectively synonymous: that is, when the name is both a macro and a function and the two have similar effects. You can think of the name simply as a function; use of the name for purposes other than calling it (such as, to take the address) will refer to the function, while calls will expand the macro and generate better but equivalent code. For example, you can use a function named `` `min' `` in the same source file that defines the macro. If you write `` `&min' `` with no argument list, you refer to the function. If you write `` `min (x, bb)' ``, with an argument list, the macro is expanded. If you write `` `(min) (a, bb)' ``, where the name `` `min' `` is not followed by an open-parenthesis, the macro is not expanded, so you wind up with a call to the function `` `min' ``.

You may not define the same name as both a simple macro and a macro with arguments.

In the definition of a macro with arguments, the list of argument names must follow the macro name immediately with no space in between. If there is a space after the macro name, the macro is defined as taking no arguments, and all the rest of the line is taken to be the expansion. The reason for this is that it is often useful to define a macro that takes no arguments and whose definition begins with an identifier in parentheses. This rule about spaces makes it possible for you to do either this:

```
#define FOO(x) - 1 / (x)
```

(which defines `` `FOO' `` to take an argument and expand into minus the reciprocal of that argument) or this:

```
#define BAR (x) - 1 / (x)
```

(which defines `` `BAR' `` to take no argument and always expand into `` `(x) - 1 / (x)' ``).

Note that the *uses* of a macro with arguments can have spaces before the left parenthesis; it's the *definition* where it matters whether there is a space.

8.3.3 Predefined Macros

Several simple macros are predefined. You can use them without giving definitions for them. They fall into two classes: standard macros and system-specific macros.

Standard Predefined Macros

The standard predefined macros are available with the same meanings regardless of the machine or operating system on which you are using GNU C. Their names all start and end with double underscores. Those preceding `__GNUC__` in this table are standardized by ANSI C; the rest are GNU C extensions.

`__FILE__`

This macro expands to the name of the current input file, in the form of a C string constant. The precise name returned is the one that was specified in `` `#include' `` or as the input file name argument.

`__LINE__`

This macro expands to the current input line number, in the form of a decimal integer constant. While we call it a predefined macro, it's a pretty strange macro, since its "definition" changes with each new line of source code.

This and `` `__FILE__' `` are useful in generating an error message to report an inconsistency detected by the program; the message can state the source line at which the inconsistency was detected. For example,

```
fprintf (stderr, "Internal error: "
          "negative string length "
                "%d at %s, line %d.",
         length, __FILE__, __LINE__);
```

A `` `#include' `` command changes the expansions of `` `__FILE__' `` and `` `__LINE__' `` to correspond to the included file. At the end of that file, when processing resumes on the input file that contained the `` `#include' `` command, the expansions of `` `__FILE__' `` and `` `__LINE__' `` revert to the values they had before the `` `#include' `` (but `` `__LINE__' `` is then incremented by one as processing moves to the line after the `` `#include' ``).

The expansions of both `` `__FILE__' `` and `` `__LINE__' `` are altered if a `` `#line' `` command is used.

`__INCLUDE_LEVEL__`

This macro expands to a decimal integer constant that represents the depth of nesting in include files. The value of this macro is incremented on every `` `#include' `` command and decremented at every end of file.

`__DATE__`

This macro expands to a string constant that describes the date on which the preprocessor is being run. The string constant contains eleven characters and looks like `` `"Jan 29 1987"' `` or `` `"Apr 1 1905"' ``.

`__TIME__`

This macro expands to a string constant that describes the time at which the preprocessor is being run. The string constant contains eight characters and looks like `` `"23:59:01"' ``.

`__STDC__`

This macro expands to the constant 1, to signify that this is ANSI Standard C. (Whether that is actually true depends on what C compiler will operate on the output from the preprocessor.)

`__GNUC__`

This macro is defined if and only if this is GNU C. This macro is defined only when the entire GNU C compiler is in use; if you invoke the preprocessor directly, `` `__GNUC__' `` is undefined.

`__GNUG__`

The GNU C compiler defines this when the compilation language is C++; use `` `__GNUG__' `` to distinguish between GNU C and GNU C++.

`__cplusplus`

The draft ANSI standard for C++ used to require predefining this variable. Though it is no longer required, GNU C++ continues to define it, as do other popular C++ compilers. You can use `` `__cplusplus' `` to test whether a header is compiled by a C compiler or a C++ compiler.

`__STRICT_ANSI__`

This macro is defined if and only if the `` `-ansi' `` switch was specified when GNU C was invoked. Its definition is the null string. This macro exists primarily to direct certain GNU header files not to define certain traditional Unix constructs which are incompatible with ANSI C.

`__BASE_FILE__`

This macro expands to the name of the main input file, in the form of a C string constant. This is the source file that was specified as an argument when the C compiler was invoked.

`__VERSION__`

This macro expands to a string which describes the version number of GNU C. The string is normally a sequence of decimal numbers separated by periods, such as `` `"1.18"' ``. The only reasonable use of this macro is to incorporate it into a string constant.

`__OPTIMIZE__`

This macro is defined in optimizing compilations. It causes certain GNU header files to define alternative macro definitions for some system library functions. It is unwise to refer to or test the definition of this macro unless you make very sure that programs will execute with the same effect regardless.

`__CHAR_UNSIGNED__`

This macro is defined if and only if the data type `char` is unsigned on the target machine. It exists to cause the standard header file `` `limit.h' `` to work correctly. It is bad practice to refer to this macro yourself; instead, refer to the standard macros defined in `` `limit.h' ``. The preprocessor uses this macro to determine whether or not to sign-extend large character constants written in octal; see section The `` `#if' `` Command.

8.3.4 Nonstandard Predefined Macros

The C preprocessor normally has several predefined macros that vary between machines because their purpose is to indicate what type of system and machine is in use. This manual, being for all systems and machines, cannot tell you exactly what their names are; instead, we offer a list of some typical ones. You can use `` `cpp -dM' `` to see the values of predefined macros;

Some nonstandard predefined macros describe the operating system in use, with more or less specificity. For example,

`unix`
: `` `unix' `` is normally predefined on all Unix systems.

`BSD`
: `` `BSD' `` is predefined on recent versions of Berkeley Unix (perhaps only in version 4.3).

Other nonstandard predefined macros describe the kind of CPU, with more or less specificity. For example,

`vax`
: `` `vax' `` is predefined on Vax computers.

`mc68000`
: `` `mc68000' `` is predefined on most computers whose CPU is a Motorola 68000, 68010 or 68020.

`m68k`
: `` `m68k' `` is also predefined on most computers whose CPU is a 68000, 68010 or 68020; however, some makers use `` `mc68000' `` and some use `` `m68k' ``. Some predefine both names. What happens in GNU C depends on the system you are using it on.

`M68020`
: `` `M68020' `` has been observed to be predefined on some systems that use 68020 CPUs--in addition to `` `mc68000' `` and `` `m68k' ``, which are less specific.

`_AM29K`
`_AM29000`
: Both `` `_AM29K' `` and `` `_AM29000' `` are predefined for the AMD 29000 CPU family.

`ns32000`
: `` `ns32000' `` is predefined on computers which use the National Semiconductor 32000 series CPU.

Yet other nonstandard predefined macros describe the manufacturer of the system. For example,

`sun`
: `` `sun' `` is predefined on all models of Sun computers.

`pyr`
: `` `pyr' `` is predefined on all models of Pyramid computers.

`sequent`
: `` `sequent' `` is predefined on all models of Sequent computers.

These predefined symbols are not only nonstandard, they are contrary to the ANSI standard because their names do not start with underscores. Therefore, the option `-ansi' inhibits the definition of these symbols.

This tends to make `-ansi' useless, since many programs depend on the customary nonstandard predefined symbols. Even system header files check them and will generate incorrect declarations if they do not find the names that are expected. You might think that the header files supplied for the Uglix computer would not need to test what machine they are running on, because they can simply assume it is the Uglix; but often they do, and they do so using the customary names. As a result, very few C programs will compile with `-ansi'. We intend to avoid such problems on the GNU system.

What, then, should you do in an ANSI C program to test the type of machine it will run on?

GNU C offers a parallel series of symbols for this purpose, whose names are made from the customary ones by adding `__' at the beginning and end. Thus, the symbol __vax__ would be available on a Vax, and so on.

The set of nonstandard predefined names in the GNU C preprocessor is controlled (when cpp is itself compiled) by the macro `CPP_PREDEFINES', which should be a string containing `-D' options, separated by spaces. For example, on the Sun 3, we use the following definition:

```
#define CPP_PREDEFINES "-Dmc68000 -Dsun -Dunix -Dm68k"
```

This macro is usually specified in `tm.h'.

8.3.5 Stringification

Stringification means turning a code fragment into a string constant whose contents are the text for the code fragment. For example, stringifying `foo (z)' results in `"foo (z)"'. In the C preprocessor, stringification is an option available when macro arguments are substituted into the macro definition. In the body of the definition, when an argument name appears, the character `#' before the name specifies stringification of the corresponding actual argument when it is substituted at that point in the definition. The same argument may be substituted in other places in the definition without stringification if the argument name appears in those places with no `#'.

Here is an example of a macro definition that uses stringification:

```
#define WARN_IF(EXP) \
do { if (EXP) \
        fprintf (stderr, "Warning: " #EXP "\n"); } \
while (0)
```

Here the actual argument for `` `EXP' `` is substituted once as given, into the `` `if' `` statement, and once as stringified, into the argument to `` `fprintf' ``. The `` `do' `` and `` `while (0)' `` are a kludge to make it possible to write `` `WARN_IF (*arg*);' ``, which the resemblance of `` `WARN_IF' `` to a function would make C programmers want to do;

The stringification feature is limited to transforming one macro argument into one string constant: there is no way to combine the argument with other text and then stringify it all together. But the example above shows how an equivalent result can be obtained in ANSI Standard C using the feature that adjacent string constants are concatenated as one string constant. The preprocessor stringifies the actual value of `` `EXP' `` into a separate string constant, resulting in text like

```
do { if (x == 0) \
       fprintf (stderr, "Warning: " "x == 0" "\n"); } \
while (0)
```

but the C compiler then sees three consecutive string constants and concatenates them into one, producing effectively

```
do { if (x == 0) \
       fprintf (stderr, "Warning: x == 0\n"); } \
while (0)
```

Stringification in C involves more than putting doublequote characters around the fragment; it is necessary to put backslashes in front of all doublequote characters, and all backslashes in string and character constants, in order to get a valid C string constant with the proper contents. Thus, stringifying `` `p = "foo\n";' `` results in `` `"p = \"foo\\n\";"' ``. However, backslashes that are not inside of string or character constants are not duplicated: `` `\n' `` by itself stringifies to `` `"\n"' ``.

Whitespace (including comments) in the text being stringified is handled according to precise rules. All leading and trailing whitespace is ignored. Any sequence of whitespace in the middle of the text is converted to a single space in the stringified result.

8.3.6 Concatenation

Concatenation means joining two strings into one. In the context of macro expansion, concatenation refers to joining two lexical units into one longer one. Specifically, an actual argument to the macro can be concatenated with another actual argument or with fixed text to produce a longer name. The longer name might be the name of a function, variable or type, or a C keyword; it might even be the name of another macro, in which case it will be expanded.

When you define a macro, you request concatenation with the special operator `` `##' `` in the macro body. When the macro is called, after actual arguments are substituted, all `` `##' `` operators are deleted, and so is any whitespace next to them (including whitespace that was part of an actual argument). The result is to concatenate the syntactic tokens on either side of the `` `##' ``.

Consider a C program that interprets named commands. There probably needs to be a table of commands, perhaps an array of structures declared as follows:

```
struct command
{
  char *name;
  void (*function) ();
};

struct command commands[] =
{
  { "quit", quit_command},
  { "help", help_command},
  ...
};
```

It would be cleaner not to have to give each command name twice, once in the string constant and once in the function name. A macro which takes the name of a command as an argument can make this unnecessary. The string constant can be created with stringification, and the function name by concatenating the argument with `` `_command' ``. Here is how it is done:

```
#define COMMAND(NAME)  { #NAME, NAME ## _command }

struct command commands[] =
{
  COMMAND (quit),
  COMMAND (help),
  ...
};
```

The usual case of concatenation is concatenating two names (or a name and a number) into a longer name. But this isn't the only valid case. It is also possible to concatenate two numbers (or a number and a name, such as `` `1.5' `` and `` `e3' ``) into a number. Also, multi-character operators such as `` `+=' `` can be formed by concatenation. In some cases it is even possible to piece together a string constant. However, two pieces of text that don't together form a valid lexical unit cannot be concatenated. For example, concatenation with `` `x' `` on one side and `` `+' `` on the other is not meaningful because those two characters can't fit together in any lexical unit of C. The ANSI standard says that such attempts at concatenation are undefined, but in the GNU C preprocessor it is well defined: it puts the `` `x' `` and `` `+' `` side by side with no particular special results.

Keep in mind that the C preprocessor converts comments to whitespace before macros are even considered. Therefore, you cannot create a comment by concatenating `` `/' `` and `` `*' ``: the `` `/*' `` sequence that starts a comment is not a lexical unit, but rather the beginning of a "long" space character. Also, you can freely use comments next to a `` `##' ``

in a macro definition, or in actual arguments that will be concatenated, because the comments will be converted to spaces at first sight, and concatenation will later discard the spaces.

8.3.7 Undefining Macros

To **undefine** a macro means to cancel its definition. This is done with the `` `#undef' `` directive. `` `#undef' `` is followed by the macro name to be undefined.

Like definition, undefinition occurs at a specific point in the source file, and it applies starting from that point. The name ceases to be a macro name, and from that point on it is treated by the preprocessor as if it had never been a macro name.

For example,

```
#define FOO 4
x = FOO;
#undef FOO
x = FOO;
```

expands into

```
x = 4;

x = FOO;
```

In this example, `` `FOO' `` had better be a variable or function as well as (temporarily) a macro, in order for the result of the expansion to be valid C code.

The same form of `` `#undef' `` directive will cancel definitions with arguments or definitions that don't expect arguments. The `` `#undef' `` directive has no effect when used on a name not currently defined as a macro.

8.3.8 Redefining Macros

Redefining a macro means defining (with `` `#define' ``) a name that is already defined as a macro. A redefinition is trivial if the new definition is transparently identical to the old one. You probably wouldn't deliberately write a trivial redefinition, but they can happen automatically when a header file is included more than once, so they are accepted silently and without effect.

Nontrivial redefinition is considered likely to be an error, so it provokes a warning message from the preprocessor. However, sometimes it is useful to change the definition of a macro in mid-compilation. You can inhibit the warning by undefining the macro with `` `#undef' `` before the second definition.

In order for a redefinition to be trivial, the new definition must exactly match the one already in effect, with two possible exceptions:

- Whitespace may be added or deleted at the beginning or the end.
- Whitespace may be changed in the middle (but not inside strings). However, it may not be eliminated entirely, and it may not be added where there was no whitespace at all.

Recall that a comment counts as whitespace.

8.4 Let us Sum up

This lesson explains importance of pre processor derectives and its applications. We were also discussed about the usage og macros, creating macros, defining and undefining macros, usage of header file, the procedure for making our own header file etc.,

8.5 Points for discussion

1. Why we need preprocessor directives?

2. Write any one of the drawback of preprocessor?

3. What is the purpose of using header files?

4. How to create our own header files?

8.6 Check your progress

1. Write any one of the drawback of preprocessor?

    1. It is difficult to identify errors.
    2. Difficult to understand also.

2. What is meant by Undefining Macros?

The word '**undefine** a macro' means to cancel its definition. This is done with the `` `#undef' `` directive. `` `#undef' `` is followed by the macro name to be undefined. Like definition, undefinition occurs at a specific point in the source file, and it applies starting from that point. The name ceases to be a macro name, and from that point on it is treated by the preprocessor as if it had never been a macro name.

8.7 Lesson-end Activities

1. Can you implement all applications using preprocessors?

2. Who will be responsible for taking care of macros?

8.8 References

Peter Aitken, Teach Yourself C in 21 Days, Fourth Edition, Sams Publisher.
Brian W. Kernighan and Dennis M. Ritchie, The C programming Language, Prentice-Hall in 1988
E.Balagursamy, Programming in Ansi C, TATA MCGraw Hill
http://www.java2s.com/Code/C/
http://publications.gbdirect.co.uk/c_book
http://www.cs.cf.ac.uk/Dave/C/
http://www.oreilly.com/catalog/pcp3/
http://www.cs.utah.edu/dept/old/texinfo/cpp
http://vergil.chemistry.gatech.edu/resources/programming/c-tutorial
http://sysprog.net/

UNIT - III

LESSON - 9 : STRUCTURES

CONTENTS

9.0 Aims and Objectives

The major aim of this lesson is to provide detailed information about structures and its implementations. This helps the programmers to make effective programs with the concept of structures.This lesson also explain about fails used in “C”.

9.1 Introduction

A *structure* is a collection of one or more variables grouped under a single name for easy manipulation. The variables in a structure, unlike those in an array, can be of different variable types. A structure can contain any of C's data types, including arrays and other structures. Each variable within a structure is called a *member* of the structure. The next section shows a simple example. You should start with simple structures. Note that the C language makes no distinction between simple and complex structures, but it's easier to explain structures in this way.

9.2 Defining and Declaring Structures

If you're writing a graphics program, your code needs to deal with the coordinates of points on the screen. Screen coordinates are written as an x value, giving the horizontal position, and a y value, giving the vertical position. You can define a structure named coord that contains both the x and y values of a screen location as follows:

```
struct coord {
int x;
int y;
};
```

The struct keyword, which identifies the beginning of a structure definition, must be followed immediately by the structure name, or *tag* (which follows the same rules as other C variable names). Within the braces following the structure name is a list of the structure's member variables. You must give a variable type and name for each member.

The preceding statements define a structure type named coord that contains two integer variables, x and y. They do not, however, actually create any instances of the structure coord. In other words, they don't *declare* (set aside storage for) any structures. There are two ways to declare structures. One is to follow the structure definition with a list of one or more variable names, as is done here:

```
struct coord {
int x;
int y;
} first, second;
```

These statements define the structure type coord and declare two structures, first and second, of type coord. First and second are each *instances* of type coord; first contains two integer members named x and y, and so does second. This method of declaring structures combines the declaration with the definition. The second method is to declare structure variables at a different location in your source code from the definition. The following statements also declare two instances of type coord:

```
struct coord {
int x;
int y;
};
/* Additional code may go here */
struct coord first, second;
```

9.3 Accessing Structure Members

Individual structure members can be used like other variables of the same type. Structure members are accessed using the *structure member operator* (.), also called the *dot operator,* between the structure name and the member name. Thus, to have the structure named first refer to a screen location that has coordinates x=50, y=100, you could write

```
first.x = 50;
first.y = 100;
```

To display the screen locations stored in the structure second, you could write

```
printf("%d,%d", second.x, second.y);
```

At this point, you might be wondering what the advantage is of using structures rather than individual variables. One major advantage is that you can copy information between structures of the same type with a simple equation statement. Continuing with the preceding example, the statement

```
first = second;
```

is equivalent to this statement:

```
first.x = second.x;
first.y = second.y;
```

When your program uses complex structures with many members, this notation can be a great time-saver. Other advantages of structures will become apparent as you learn some advanced techniques. In general, you'll find structures to be useful whenever information of different variable types needs to be treated as a group. For example, in a mailing list database, each entry could be a structure, and each piece of information (name, address, city, and so on) could be a structure member.

The struct Keyword

```
struct tag {
structure_member(s);
/* additional statements may go here */
} instance;
```

The struct keyword is used to declare structures. A structure is a collection of one or more variables *(structure_member*s) that have been grouped under a single name for easy man-ipulation. The variables don't have to be of the same variable type, nor do they have to be simple variables. Structures also can hold arrays, pointers, and other structures. The keyword struct identifies the beginning of a structure definition. It's followed by a tag that is the name given to the structure. Following the tag are the structure members, enclosed in braces. An *instanc*e, the actual declaration of a structure, can also be defined. If you define the structure without the instance, it's just a template that can be used later in a program to declare structures. Here is a template's format:

```
struct tag {
structure_member(s);
/* additional statements may go here */
};
```

To use the template, you use the following format:

```
struct tag instance;
```

To use this format, you must have previously declared a structure with the given tag.

Example 9.1

```
/* Declare a structure template called SSN */
struct SSN {
int first_three;
char dash1;
int second_two;
char dash2;
int last_four;
}
/* Use the structure template */
struct SSN customer_ssn;
```

Example 9. 2

```
/* Declare and initialize a structure */
struct time {
int hours;
int minutes;
int seconds;

} time_of_birth = { 8, 45, 0 };
```

9.4 Arrays as structure members

One of the member of the structure can be of type array.The following example shows arrays as members of the structure.

Example 9.3

```
/* Declare a structure and instance together */
struct date {
char month[2];
char day[2];
char year[4];
} current_date;
```

9.5 Arrays of structures

The structure variable name itself can be of type array for creating multiple number of records.This structure is refered as arrays of structures.Here is an example using an array of structures like the one before. A function is used to read characters from the program's standard input and return an appropriately initialized structure. When a newline has been read or the array is full, the structures are sorted into order depending on the character value, and then printed out.

```
Example 9.4
#include <stdio.h>
#include <stdlib.h>

#define ARSIZE 10

struct wp_char{
      char wp_cval;
      short wp_font;
      short wp_psize;
}ar[ARSIZE];

/*
* type of the input function -
* could equally have been declared above;
* it returns a structure and takes no arguments.
*/
struct wp_char infun(void);

main(){
      int icount, lo_indx, hi_indx;

      for(icount = 0; icount < ARSIZE; icount++){
              ar[icount] = infun();
              if(ar[icount].wp_cval == '\n'){
                      /*
                       * Leave the loop.
                       * not incrementing icount means that the
                       * '\n' is ignored in the sort
                       */
```

```
                        break;
                }
        }

        /* now a simple exchange sort */

        for(lo_indx = 0; lo_indx <= icount-2; lo_indx++)
                for(hi_indx = lo_indx+1; hi_indx <= icount-1; hi_indx++){
                        if(ar[lo_indx].wp_cval > ar[hi_indx].wp_cval){
                                /*
                                 * Swap the two structures.
                                 */
                                struct wp_char wp_tmp = ar[lo_indx];
                                ar[lo_indx] = ar[hi_indx];
                                ar[hi_indx] = wp_tmp;
                        }
                }

        /* now print */
        for(lo_indx = 0; lo_indx < icount; lo_indx++){
                printf("%c %d %d\n", ar[lo_indx].wp_cval,
                                ar[lo_indx].wp_font,
                                ar[lo_indx].wp_psize);
        }
        exit(EXIT_SUCCESS);
}

struct wp_char
infun(void){
        struct wp_char wp_char;

        wp_char.wp_cval = getchar();
        wp_char.wp_font = 2;
        wp_char.wp_psize = 10;

        return(wp_char);
}
```

Once it is possible to declare structures it seems pretty natural to declare arrays of them, use them as members of other structures and so on. In fact the only restriction is that a structure cannot contain an example of itself as a member—in which case its size would be an interesting concept for philosophers to debate, but hardly useful to a C programmer.

Example 9.5

```
/* Demonstrates using arrays of structures. */

#include <stdio.h>

/* Define a structure to hold entries. */

struct entry {
char fname[20];
char lname[20];
char phone[10];
};

/* Declare an array of structures. */

struct entry list[4];

int i;

main()
{

/* Loop to input data for four people. */

for (i = 0; i < 4; i++)
{
printf("\nEnter first name: ");
scanf("%s", list[i].fname);
printf("Enter last name: ");
scanf("%s", list[i].lname);
printf("Enter phone in 123-4567 format: ");
scanf("%s", list[i].phone);
}


/* Print two blank lines. */


printf("\n\n");

/* Loop to display data. */

for (i = 0; i < 4; i++)
{
printf("Name: %s %s", list[i].fname, list[i].lname);
printf("\t\tPhone: %s\n", list[i].phone);
}

return 0;
}
```

```
Enter first name: Bradley
Enter last name: Jones
Enter phone in 123-4567 format: 555-1212
Enter first name: Peter
Enter last name: Aitken
Enter phone in 123-4567 format: 555-3434
Enter first name: Melissa
Enter last name: Jones
Enter phone in 123-4567 format: 555-1212
Enter first name: Deanna
Enter last name: Townsend
Enter phone in 123-4567 format: 555-1234
Name: Bradley Jones Phone: 555-1212
Name: Peter Aitken Phone: 555-3434
Name: Melissa Jones Phone: 555-1212

Name: Deanna Townsend Phone: 555-1234
```

9.6 Structures with in Structures

As mentioned earlier, a C structure can contain any of C's data types. For example, a structure can contain other structures. The previous example can be extended to illustrate this. Assume that your graphics program needs to deal with rectangles. A rectangle can be defined by the coordinates of two diagonally opposite corners. You've already seen how to define a structure that can hold the two coordinates required for a single point. You need two such structures to define a rectangle. You can define a structure as follows (assuming, of course, that you have already defined the type coord structure):

```
struct rectangle {
struct coord topleft;
struct coord bottomrt;
};
```

This statement defines a structure of type rectangle that contains two structures of type coord. These two type coord structures are named topleft and bottomrt. The preceding statement defines only the type rectangle structure. To declare a structure, you must then include a statement such as

```
struct rectangle mybox;
```

You could have combined the definition and declaration, as you did before for the type coord:

```
struct rectangle {
struct coord topleft;
struct coord bottomrt;
} mybox;
```

To access the actual data locations (the type int members), you must apply the member operator (.) twice. Thus, the expression

```
mybox.topleft.x
```

refers to the x member of the topleft member of the type rectangle structure named mybox. To define a rectangle with coordinates (0,10),(100,200), you would write

```
mybox.topleft.x = 0;
mybox.topleft.y = 10;
mybox.bottomrt.x = 100;
mybox.bottomrt.y = 200;
```

Example 9.6 A demonstration of structures that contain other structures.

```
/* Demonstrates structures that contain other structures. */

/* Receives input for corner coordinates of a rectangle and
calculates the area. Assumes that the y coordinate of the
upper-left corner is greater than the y coordinate of the
lower-right corner, that the x coordinate of the lower-
right corner is greater than the x coordinate of the upper-
left corner, and that all coordinates are positive. */


#include <stdio.h>

int length, width;
long area;

struct coord{
int x;
int y;
};

struct rectangle{
struct coord topleft;
struct coord bottomrt;
} mybox;

main()
{
/* Input the coordinates */

printf("\nEnter the top left x coordinate: ");
scanf("%d", &mybox.topleft.x);

printf("\nEnter the top left y coordinate: ");
scanf("%d", &mybox.topleft.y);

printf("\nEnter the bottom right x coordinate: ");
scanf("%d", &mybox.bottomrt.x);

printf("\nEnter the bottom right y coordinate: ");
scanf("%d", &mybox.bottomrt.y);

/* Calculate the length and width */

width = mybox.bottomrt.x - mybox.topleft.x;
44: length = mybox.bottomrt.y - mybox.topleft.y;
45:
46: /* Calculate and display the area */
47:
48: area = width * length;
49: printf("\nThe area is %ld units.\n", area);
50:
51: return 0;
52: }
Enter the top left x coordinate: 1
Enter the top left y coordinate: 1
Enter the bottom right x coordinate: 10
Enter the bottom right y coordinate: 10
The area is 81 units.
```

The coord structure is defined in lines 15 through 18 with its two members, x and y. Lines 20 through 23 declare and define an instance, called mybox, of the rectangle structure. The two members of the rectangle structure are topleft and bottomrt, both structures of type coord. Lines 29 through 39 fill in the values in the mybox structure. At first it might seem that there are only two values to fill, because mybox has only two members. However, each of mybox's members has its own members. Topleft and bottomrt have two members each, x and y from the coord structure. This gives a total of four members to be filled. After the members are filled with values, the area is calculated using the structure and member names. When using the x and y values, you must include the structure instance name. Because x and y are in a structure within a structure, you must use the instance names of both structures--mybox.bottomrt.x, mybox.bottomrt.y, mybox.topleft.x, and mybox.topleft.y--in the calculations.

C places no limits on the nesting of structures. While memory allows, you can define structures that contain structures that contain structures that contain structures--well, you get the idea! Of course, there's a limit beyond which nesting becomes unproductive. Rarely are more than three levels of nesting used in any C program.

9.7 Structures as Function Arguments

A structure can be passed as a function argument just like any other variable. This raises a few practical issues. Where we wish to modify the value of members of the structure, we must pass a pointer to that structure. This is just like passing a pointer to an int type argument whose value we wish to change.

If we are only interested in one member of a structure, it is probably simpler to just pass that member. This will make for a simpler function, which is easier to re-use. Of course if we wish to change the value of that member, we should pass a pointer to it.

When a structure is passed as an argument, each member of the structure is copied. This can prove expensive where structures are large or functions are called frequently. Passing and working with pointers to large structures may be more efficient in such cases.

For example,

```
struct mystruct
        {
                int a;
                char b;
        };
Main()
{
  struch mystruct svn;
  --------------
    myfunction(svn)
  ----------------
  ----------------
}

void myfunction( svn1)
struch mystruct svn1;
{
   ----------------
   ----------------
}
```

In this example, the formal argument svn1 should be declared as structre of type “mystruct”.

9.8 Pointers and structures

If what the last paragraph says is true, that it is more common to use pointers to structures than to use the structures directly. we need to know how to do it. Declaring pointers is easy of course:

```
struct wp_char *wp_p;
```

gives us one straight away. But how do we access the members of the structure? One way might be to look through the pointer to get the whole structure, then select the member:

```
/* get the structure, then select a member */
(*wp_p).wp_cval
```

that would certainly work (the parentheses are there because . has a higher precedence than `*`). It's not an easy notation to work with though, so C introduces a new operator to clean things up; it is usually known as the ‘pointing-to’ operator. Here it is being used:

```
/* the wp_cval in the structure wp_p points to */
wp_p->wp_cval = 'x';
```

and although it might not look a lot easier than its alternative, it pays off when the structure contains pointers, as in a linked list. The pointing-to syntax is much easier if you want to follow two or three stages down the links of a linked list. If you haven't come across linked lists before, you're going to learn a lot more than just the use of structures before this chapter finishes!

If the thing on the left of the `.` or `->` operator is qualified (with `const` or `volatile`) then the result is also has those qualifiers associated with it. Here it is, illustrated with pointers; when the pointer points to a qualified type the result that you get is also qualified:

Example 9.7

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

struct somestruct{
      int i;
};

main(){
      struct somestruct *ssp, s_item;
      const struct somestruct *cssp;

      s_item.i = 1;   /* fine */
      ssp = &s_item;
      ssp->i += 2;    /* fine */
      cssp = &s_item;
      cssp->i = 0;    /* not permitted - cssp points to const objects
*/

      exit(EXIT_SUCCESS);
}
```

Not all compiler writers seem to have noticed that requirement the compiler that we used to test the last example failed to warn that the final assignment violated a constraint.

Here is the example rewritten using pointers, and with the input function infun changed to accept a pointer to a structure rather than returning one. This is much more likely to be what would be seen in practice.

(It is fair to say that, for a really efficient implementation, even the copying of structures would probably be dropped, especially if they were large. Instead, an array of pointers would be used, and the pointers exchanged until the sorted data could be found by traversing the pointer array in index order. That would complicate things too much for a simple example.)

```
Example 9.8
#include <stdio.h>
#include <stdlib.h>
```

```
#define ARSIZE 10

struct wp_char{
      char wp_cval;
      short wp_font;
      short wp_psize;
}ar[ARSIZE];

void infun(struct wp_char *);

main(){
      struct wp_char wp_tmp, *lo_indx, *hi_indx, *in_p;

      for(in_p = ar; in_p < &ar[ARSIZE]; in_p++){
              infun(in_p);
              if(in_p->wp_cval == '\n'){
                      /*
                       * Leave the loop.
                       * not incrementing in_p means that the
                       * '\n' is ignored in the sort
                       */
                      break;
              }
      }

      /*
       * Now a simple exchange sort.
       * We must be careful to avoid the danger of pointer underflow,
       * so check that there are at least two entries to sort.
       */

      if(in_p-ar > 1) for(lo_indx = ar; lo_indx <= in_p-2; lo_indx++){
              for(hi_indx = lo_indx+1; hi_indx <= in_p-1; hi_indx++){
                      if(lo_indx->wp_cval > hi_indx->wp_cval){
                              /*
                               * Swap the structures.
                               */
                              struct wp_char wp_tmp = *lo_indx;
                              *lo_indx = *hi_indx;
                              *hi_indx = wp_tmp;
                      }
              }
      }

      /* now print */
      for(lo_indx = ar; lo_indx < in_p; lo_indx++){
              printf("%c %d %d\n", lo_indx->wp_cval,
                              lo_indx->wp_font,
                              lo_indx->wp_psize);
      }
      exit(EXIT_SUCCESS);
}

void
infun( struct wp_char *inp){

      inp->wp_cval = getchar();
```

```
        inp->wp_font = 2;
        inp->wp_psize = 10;

        return;
}
```

The next issue is to consider what a structure looks like in terms of storage layout. It's best not to worry about this too much, but it is sometimes useful if you have to use C to access record-structured data written by other programs. The `wp_char` structure will be allocated storage as shown in Figure 9.8.1.

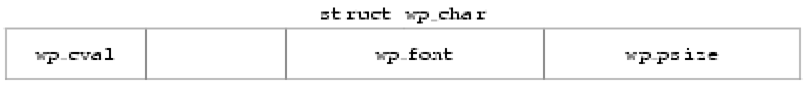


Figure 9.8.1. Storage Layout of a Structure

The diagram assumes a number of things: that a `char` takes 1 byte of storage; that a `short` needs 2 bytes; and that `shorts` must be aligned on even byte addresses in this architecture. As a result the structure contains an unnamed 1-byte member inserted by the compiler for architectural reasons. Such addressing restrictions are quite common and can often result in structures containing ‘holes’.

The Standard makes some guarantees about the layout of structures and unions:

- Members of a structure are allocated within the structure in the order of their appearance in the declaration and have ascending addresses.
- There must not be any padding in front of the first member.
- The address of a structure is the same as the address of its first member, provided that the appropriate cast is used. Given the previous declaration of `struct wp_char`, if item is of type `struct wp_char`, then `(char *)item == &item.wp_cval`.
- Bit fields don't actually have addresses, but are conceptually packed into *units* which obey the rules above.

9.9 Linked lists and other structures

The combination of structures and pointers opens up a lot of interesting possibilities. This is not a textbook on complex linked data structures, but it will go on to describe two very common examples of the breed: linked lists and trees. Both have a feature in common: they consist of structures containing pointers to other structures, all the structures typically being of the same type. Figure 9.9.1 shows a picture of a linked list.

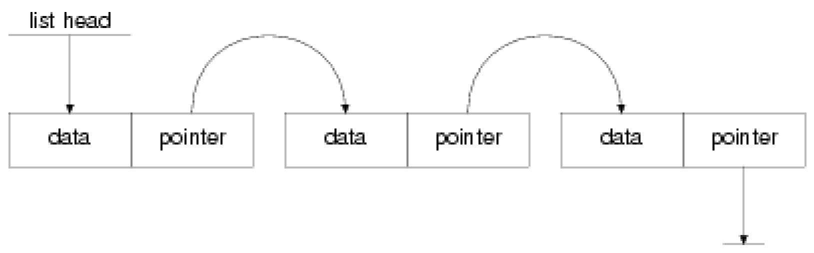


Figure 9.9.1. List linked by pointers

The sort of declaration needed for that is this:

```
struct list_ele{
      int data;       /* or whatever you like here */
      struct list_ele *ele_p;
};
```

Now, at first glance, it seems to contain itself—which is forbidden—but in fact it only contains a *pointer* to itself. How come the pointer declaration is allowed? Well, by the time the compiler reaches the pointer declaration it already knows that there is such a thing as a `struct list_ele` so the declaration is permitted. In fact, it is possible to make a incomplete declaration of a structure by saying

```
struct list_ele;
```

at some point before the full declaration. A declaration like that declares an *incomplete type*. This will allow the declaration of pointers before the full declaration is seen. It is also important in the case of cross-referencing structures where each must contain a pointer to the other, as shown in the following example.

```
struct s_1;     /* incomplete type */

struct s_2{
      int something;
      struct s_1 *sp;
};

struct s_1{     /* now the full declaration */
      float something;
      struct s_2 *sp;
};
```

This illustrates the need for incomplete types. It also illustrates an important thing about the names of structure members: they inhabit a name-space per structure, so element names can be the same in different structures without causing any problems.

Incomplete types may only be used where the size of the structure isn't needed yet. A full declaration must have been given by the time that the size is used. The later full declaration mustn't be in an inner block because then it becomes a new declaration of a different structure.

```
Example 9.9
struct x;       /* incomplete type */

/* valid uses of the tag */
struct x *p, func(void);

void f1(void){
      struct x{int i;};       /* redeclaration! */
}

/* full declaration now */
struct x{
      float f;
}s_x;

void f2(void){
      /* valid statements */
      p = &s_x;
      *p = func();
      s_x = func();
}

struct x
func(void){
      struct x tmp;
      tmp.f = 0;
      return (tmp);
}
```

There's one thing to watch out for: you get a incomplete type of a structure *simply by mentioning its name!* That means that this works:

```
struct abc{ struct xyz *p;};
      /* the incomplete type 'struct xyz' now declared */
struct xyz{ struct abc *p;};
      /* the incomplete type is now completed */
```

There's a horrible danger in the last example, though, as this shows:

```
struct xyz{float x;} var1;

main(){
      struct abc{ struct xyz *p;} var2;

      /* AAAGH - struct xyz REDECLARED */
      struct xyz{ struct abc *p;} var3;
}
```

The result is that `var2.p` can hold the address of `var1`, but emphatically not the address of `var3` which is of a different type! It can be fixed (assuming that it's not what you wanted) like this:

```
struct xyz{float x;} var1;
```

```
main(){
      struct xyz;     /* new incomplete type 'struct xyz' */
      struct abc{ struct xyz *p;} var2;
      struct xyz{ struct abc *p;} var3;
}
```

The type of a structure or union is completed when the closing } of its declaration is seen; it must contain at least one member or the behaviour is undefined.

The other principal way to get incomplete types is to declare arrays without specifying their size, their type is incomplete until a later declaration provides the missing information:

```
int ar[];       /* incomplete type */
int ar[5];      /* completes the type */
```

If you try that out, it will only work if the declarations are outside any blocks (external declarations), but that's for other reasons.

Back to the linked list. There were three elements linked into the list, which could have been built like this:

```
struct list_ele{
      int data;
      struct list_ele *pointer;
}ar[3];

main(){

      ar[0].data = 5;
      ar[0].pointer = &ar[1];
      ar[1].data = 99;
      ar[1].pointer = &ar[2];
      ar[2].data = -7;
      ar[2].pointer = 0;      /* mark end of list */
      return(0);
}
```

and the contents of the list can be printed in two ways. The array can be traversed in order of index, or the pointers can be used as in the following example.

Example 9.10

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

struct list_ele{
      int data;
      struct list_ele *pointer;
}ar[3];
```

```
main(){

      struct list_ele *lp;

      ar[0].data = 5;
      ar[0].pointer = &ar[1];
      ar[1].data = 99;
      ar[1].pointer = &ar[2];
      ar[2].data = -7;
      ar[2].pointer = 0;      /* mark end of list */

      /* follow pointers */
      lp = ar;
      while(lp){
              printf("contents %d\n", lp->data);
              lp = lp->pointer;
      }
      exit(EXIT_SUCCESS);
}
```

It's the way that the pointers are followed which makes the example interesting. Notice how the pointer in each element is used to refer to the next one, until the pointer whose value is `0` is found. That value causes the `while` loop to stop. Of course the pointers can be arranged in any order at all, which is what makes the list such a flexible structure. Here is a function which could be included as part of the last program to sort the linked list into numeric order of its data fields. It rearranges the pointers so that the list, when traversed in pointer sequence, is found to be in order. It is important to note that the data itself is not copied. The function must return a pointer to the head of the list, because that is not necessarily at `ar[0]` any more.

```
struct list_ele *
sortfun( struct list_ele *list )
{

      int exchange;
      struct list_ele *nextp, *thisp, dummy;

      /*
       * Algorithm is this:
       * Repeatedly scan list.
       * If two list items are out of order,
       * link them in the other way round.
       * Stop if a full pass is made and no
       * exchanges are required.
       * The whole business is confused by
       * working one element behind the
       * first one of interest.
       * This is because of the simple mechanics of
       * linking and unlinking elements.
       */

      dummy.pointer = list;
      do{
              exchange = 0;
              thisp = &dummy;
              while( (nextp = thisp->pointer)
                      && nextp->pointer){
                      if(nextp->data < nextp->pointer->data){
                              /* exchange */
                              exchange = 1;
                              thisp->pointer = nextp->pointer;
                              nextp->pointer =
                                      thisp->pointer->pointer;
                              thisp->pointer->pointer = nextp;
                      }
                      thisp = thisp->pointer;
              }
      }while(exchange);

      return(dummy.pointer);
}
```

Expressions such as `thisp->pointer->pointer` are commonplace in list processing. It's worth making sure that you understand it; the notation emphasizes the way that links are followed.

9.10 Union

Unions allow one same portion of memory to be accessed as different data types, since all of them are in fact the same location in memory. Its declaration and use is similar to the one of structures but its functionality is totally different:

```
union union_name {
  member_type1 member_name1;
  member_type2 member_name2;
  member_type3 member_name3;
  .
  .
} object_names;
```

All the elements of the `union` declaration occupy the same physical space in memory. Its size is the one of the greatest element of the declaration. For example:

```
union mytypes_t {
  char c;
  int i;
  float f;
  } mytypes;
```

defines three elements:

```
mytypes.c
mytypes.i
mytypes.f
```

each one with a different data type. Since all of them are referring to the same location in memory, the modification of one of the elements will affect the value of all of them. We cannot store different values in them independent from each other.

One of the uses a union may have is to unite an elementary type with an array or structures of smaller elements. For example:

```
union mix_t {
  long l;
  struct {
    short hi;
    short lo;
    } s;
  char c[4];
} mix;
```

defines three names that allow to access the same group of 4 bytes: `mix.l`, `mix.s` and `mix.c` and which we can use according to how we want to access these bytes, as if they were a single `long`-type data, as if they were two `short` elements or as an array of `char` elements, respectively. I have mixed types, arrays and structures in the union so that you can see the different ways that we can access the data. For a *little-endian* system (most PC platforms), this union could be represented as follows:

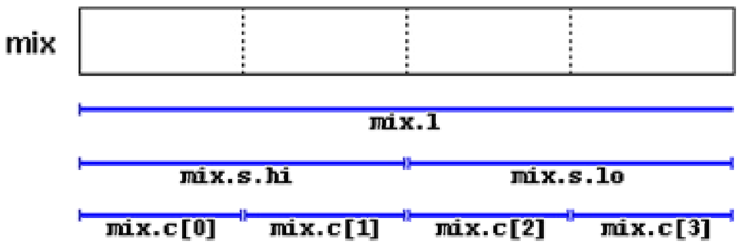


Figure 9.10.1 Memory specification for union

The exact alignment and order of the members of a union in memory is platform dependant. Therefore be aware of possible portability issues with this type of use.

9.10.1 Using Structures within Unions:

Unions can contain any types of variables, including structures. Be aware that the length of a union is the length of its longest variable. If the name in the previous example had only been 3 characters long, there would have been an undefined slack byte after name, since an integer occupies 4 bytes.

If you have a more complex situation than a simple redefine of one scalar variable with another, you may need to use structures in the union. Let's say an area in the record could contain either one long integer or two short integers; one solution would be to define a structure for the two shorts.

```
typedef struct twoshorts
  {
  short smallamount1;
  short smallamount2;
  } TwoShorts;

typedef union udata
  {
  TwoShorts smallamounts;
  int       bigamount;
  } Udata;
```

9.11 Let us Sum up

In this lesson, we described about

Structures and its importance
Accessing Structure Members using dot operator
Arrays as structure members
Arrays of structures for creating multiple records
Structures with in Structures like nesting of statement
Structures as Function Arguments
How to use pointers to structures
Linked lists and other structures
Union and Using Structures within Unions

9.12 Points for discussion

Differintate structure and union
Define structure
What is meant by self referential structure?
Define nesting of structure

9.13 Check your progress

1. Define the term union

Unions are similar to structures. Unions allow one same portion of memory to be accessed as different data types, since all of them are in fact the same location in memory. Its declaration and use is similar to the one of structures but its functionality is totally different:

```
union union_name {
  member_type1 member_name1;
  member_type2 member_name2;
  member_type3 member_name3;
  .
  .
} object_names;
```

All the elements of the `union` declaration occupy the same physical space in memory. Its size is the one of the greatest element of the declaration.

2. What is meant by self referential structure?

If any one of the structure member is of type pointer and that points to same structure then the structure is referred as self referential structure.

```
Struct mystruct
{
        int data;
        struct mystruct *ptr;
};
```

here, ptr is pointer which points to the same structure.

9.14 Lesson-end Activities

1. How structure will differ from array?
2. Do you feel union is better than structure?

9.15 References

Peter Aitken, Teach Yourself C in 21 Days, Fourth Edition, Sams Publisher.
Brian W. Kernighan and Dennis M. Ritchie, The C programming Language, Prentice-Hall in 1988
E.Balagursamy, Programming in Ansi C, TATA MCGraw Hill
http://www.cs.cf.ac.uk/Dave/C/
http://www.oreilly.com/catalog/pcp3/
http://www.cs.utah.edu/dept/old/texinfo/cpp
http://vergil.chemistry.gatech.edu/resources/programming/c-tutorial
http://sysprog.net/
http://www.mycplus.com/cplus

LESSON - 10 : FILES

CONTENTS

10.0 Aims and Objectives

The objective of this lesson is to help programmers to understand about about files which are very important for large-scale data processing. Data are stored in data files and programs are stored in program files. This lesson helps readers to implement their programs using files effectively.

10.1 File handling in C Language – an Introduction.

What is a File?

Abstractly, a file is a collection of bytes stored on a secondary storage device, which is generally a disk of some kind. The collection of bytes may be interpreted, for example, as characetrs, words, lines, paragraphs and pages from a textual document; fields and records belonging to a database; or pixels from a graphical image. The meaning attached to a particular file is determined entirely by the data structures and operations used by a program to process the file. It is conceivable (and itsometimes happens) that a graphics file will be read and displayed by a program designed to process textual data. The result is that no meaningful output occurs (probably) and this is to be expected. A file is simply a machine decipherable storage media where programs and data are stored for machine usage.

Essentially there are two kinds of files that programmers deal with text files and binary files. These two classes of files will be discussed in the following sections.

10.1.1 ASCII Text files

A text file can be a stream of characters that a computer can process sequentially. It is not only processed sequentially but only in forward direction. For this reason a text file is usually opened for only one kind of operation (reading, writing, or appending) at any given time.

Similarly, since text files only process characters, they can only read or write data one character at a time. (In C Programming Language, Functions are provided that deal with lines of text, but these still essentially process data one character at a time.) A text stream in C is a special kind of file. Depending on the requirements of the operating system, newline characters may be converted to or from carriage-return/linefeed combinations depending on whether data is being written to, or read from, the file. Other character conversions may also occur to satisfy the storage requirements of the operating system. These translations occur transparently and they occur because the programmer has signalled the intention to process a text file.

10.1.2 Binary files

A binary file is no different to a text file. It is a collection of bytes. In C Programming Language a byte and a character are equivalent. Hence a binary file is also referred to as a character stream, but there are two essential differences.

1. No special processing of the data occurs and each byte of data is transferred to or from the disk unprocessed.
2. C Programming Language places no constructs on the file, and it may be read from, or written to, in any manner chosen by the programmer.

Binary files can be either processed sequentially or, depending on the needs of the application, they can be processed using random access techniques. In C Programming Language, processing a file using random access techniques involves moving the current file position to an appropriate place in the file before reading or writing data. This indicates a second characteristic of binary files – they a generally processed using read and write operations simultaneously.

For example, a database file will be created and processed as a binary file. A record update operation will involve locating the appropriate record, reading the record into memory, modifying it in some way, and finally writing the record back to disk at its appropriate location in the file. These kinds of operations are common to many binary files, but are rarely found in applications that process text files.

10.1.3 Steps to be considered during file processing

You can have many files on your disk. If you wish to use a file in your programs, then you must specify which file or files you wish to use.

Specifying the file you wish to use is referred to as ***opening*** the file.

When you open a file you must also specify what you wish to do with it i.e. **Read** from the file, **Write** to the file, or both.

Because you may use a number of different files in your program, you must specify when reading or writing which file you wish to use. This is accomplished by using a variable called a **file pointer.**

Every file you open has its own file pointer variable. When you wish to write to a file you specify the file by using its file pointer variable.

10.2 Opening a file pointer using fopen()

C communicates with files using a new datatype called a file pointer. This type is defined within stdio.h, and written as FILE *. A file pointer called output_file is declared in a statement like

```
  FILE *output_file;
```

The function **`fopen`** is one of the Standard Library functions and returns a file pointer which you use to refer to the file you have opened.

General format is

Filepointer = fopen (“file name”, “mode”);

File name will be a valied file and mode may be of different form.

Generally we will use following modes frequently,

r – read only
w – write only
a – append only
r+ - similar to read except for both read and write
w+ - similar to write except for both read and write
a+ - similar to append except for both read and write

Reading (r)

The second parameter is the file attribute and can be any of three letters, r, w, or a, and must be lower case. When an r is used, the file is opened for reading, a w is used to indicate a file to be used for writing, and an a indicates that you desire to append additional data to the data already in an existing file. Most C compilers have other file attributes available; check your Reference Manual for details. Using the r indicates that the file is assumed to be a text file. Opening a file for reading requires that the file already exist. If it does not exist, the file pointer will be set to NULL and can be checked by the program.

Writing (w)

When a file is opened for writing, it will be created if it does not already exist and it will be reset if it does, resulting in the deletion of any data already there. Using the w indicates that the file is assumed to be a text file.

Appending (a)

When a file is opened for appending, it will be created if it does not already exist and it will be initially empty. If it does exist, the data input point will be positioned at the end of the present data so that any new data will be added to any data that already exists in the file. Using the a indicates that the file is assumed to be a text file.

e.g.

```
fp = fopen( “prog.c”,  “r”) ;
```

The above statement **opens** a file called `prog.c` for **reading** and associates the file pointer `fp` with the file.

When we wish to access this file for I/O, we use the file pointer variable `fp` to refer to it.

You can have up to about 20 files open in your program - you need one file pointer for each file you intend to use.

Consider another example, If the file cannot be opened for any reason then the value NULL will be returned. You will usually use fopen as follows

```
if ((output_file = fopen("output_file", "w")) == NULL)
        fprintf(stderr, "Cannot open %s\n", "output_file");
```

10.3 Closing a file using fclose()

The fclose command can be used to disconnect a file pointer from a file. This is usually done so that the pointer can be used to access a different file. Systems have a limit on the number of files which can be open simultaneously, so it is a good idea to close a file when you have finished using it.

This would be done using a statement like

```
fclose(output_file);
```

If files are still open when a program exits, the system will close them for you. However it is usually better to close the files properly.

10.4 Input and Output using file pointers

Having opened a file pointer, you will wish to use it for either input or output. C supplies a set of functions to allow you to do this. All are very similar to input and output functions that you have already met.

Character Input and Output with Files

This is done using equivalents of getchar and putchar which are called **getc** and **putc**. Each takes an extra argument, which identifies the file pointer to be used for input or output.

The routine `getc()` is similar to `getchar()`
and `putc()` is similar to `putchar().`

Thus the statement

```
     c = getc(fp);
```

reads the next character from the file referenced by `fp` and the statement

```
     putc(c,fp);
```

For example,

writes the character `c` into file referenced by `fp.`

```
Examplr 10.1
/* file.c: Display contents of a file on screen */

#include <stdio.h>

void main()
{
     FILE *fopen(), *fp;
     int c ;

     fp = fopen( “prog.c”, “r” );
     c = getc( fp ) ;
     while (  c != EOF )
     {
          putchar( c );
          c = getc ( fp );
     }

     fclose( fp );
}
```

In this program, we open the file `prog.c` for reading.

We then read a character from the file. This file must exist for this program to work.

If the file is empty, we are at the end, so `getc` returns `EOF` a special value to indicate that the end of file has been reached. (Normally -1 is used for `EOF`)

The while loop simply keeps reading characters from the file and displaying them, until the end of the file is reached.

The function **`fclose`** is used to ***close*** the file i.e. indicate that we are finished processing this file.

This program is slightly modified as follows

Example 10.2

```
/* Prompt user for filename and display file on screen */

#include <stdio.h>

void main()
{
     FILE *fopen(), *fp;
     int c ;
     char filename[40] ;

     printf(“Enter file to be displayed: “);
     gets( filename ) ;

     fp = fopen( filename, “r”);

     c = getc( fp ) ;

     while (  c != EOF )
     {
          putchar(c);
          c = getc ( fp );
     }

     fclose( fp );
}
```

In this program, we pass the name of the file to be opened which is stored in the array called `filename`, to the `fopen` function. In general, anywhere a string constant such as `“prog,c”` can be used so can a character array such as `filename`. (Note the **reverse** is **not** true).

The above programs suffer a major limitation. They **do not** check whether the files to be used exist or not.

If you attempt to read from an non-existent file, your program will crash!!

The `fopen` function was designed to cope with this eventuality. It checks if the file can be opened appropriately. If the file **cannot be opened**, it returns a **NULL** pointer. Thus by checking the file pointer returned by `fopen`, you can determine if the file was opened correctly and take appropriate action e.g.

```
fp = fopen (filename, “r”) ;

if ( fp  ==  NULL)
{
     printf(“Cannot open %s for reading \n”, filename );
     exit(1) ; /*Terminate program: Commit suicide !!*/
}
```

The above code fragment show how a program might check if a file could be opened appropriately.

The function **`exit()`** is a special function which terminates your program immediately.

exit(0) mean that you wish to indicate that your program terminated successfully whereas a nonzero value means that your program is terminating due to an error condition.

Alternatively, you could prompt the user to enter the filename again, and try to open it again:

```
fp = fopen (fname, “r”) ;

while ( fp  ==  NULL)
{
     printf(“Cannot open %s for reading \n”, fname );

     printf(“\n\nEnter filename :” );
     gets( fname );

     fp = fopen (fname, “r”) ;
}
```

In this code fragment, we keep reading filenames from the user until a valid existing filename is entered.

To read and write Integer values

Similar to getc() and putc(), the functions getw() and putw() are used to read and write an integer with specified file.

Example 10.3 /* program to write and read numbers in to a file pointed by file pointer */

```
#include <stdio.h>

void main()
{
     FILE *fopen(), *fp;
     int c ;

     fp = fopen( “numbers”, “w” );
     scanf(“%d”,&c) /* reading a number */
     while (  c != 0 )
     {
          putw( c,fp );
          scanf(“%d”,&c)
     }

     fclose( fp );
     fp = fopen( “numbers”, “r” );

          while (  (c = getw(fp))!=feof(fp) )
     {
          c= getw( fp );
          printf(“%d”,c)
     }
fclose(fp);
}
```

Example 10.4

```
/*count.c : Count characters in a file*/
#include <stdio.h>

void main()
     /* Prompt user for file and count number of characters
        and lines in it*/
{
     FILE *fopen(), *fp;
     int c , nc, nlines;
     char filename[40] ;

     nlines = 0 ;
     nc = 0;

     printf(“Enter file name: “);
     gets( filename );

     fp = fopen( filename, “r” );

     if ( fp == NULL )
     {
          printf(“Cannot open %s for reading \n”, filename );
          exit(1);      /* terminate program */
     }

     c = getc( fp ) ;
     while (  c != EOF )
     {
          if ( c  ==  ‘\n’  )
               nlines++ ;

          nc++ ;
          c = getc ( fp );
     }

     fclose( fp );

     if ( nc != 0 )
     {
          printf(“There are %d characters in %s \n”, nc,
                                             filename );
          printf(“There are %d lines \n”, nlines );
     }
     else
          printf(“File: %s is empty \n”, filename );
}
```

Example 10.5 : Write a program to compare two files specified by the user, displaying a message indicating whether the files are identical or different. This is the basis of a **compare** command provided by most operating systems. Here our file processing loop is as follows:

```
read character ca from file A;
read character cb from file B;

while ca == cb and not EOF file A and not EOF file B
begin
     read character ca from file A;
     read character cb from file B;
```

```
end

if ca == cb then
          printout(“Files identical”);
else
          printout(“Files differ”);
```

This program illustrates the use of I/O with two files. In general you can manipulate up to 20 files, but for most purposes not more than 4 files would be used. All of these examples illustrate the usefulness of processing files character by character. As you can see a number of Operating System programs such as compare, type, more, copy can be easily written using character I/O. These programs are normally called **system programs** as they come with the operating system. The important point to note is that these programs are in no way special. They are no different in nature than any of the programs we have constructed so far.

```
Example 10.6
/* compare.c : compare two files */

#include <stdio.h>
void main()
{
     FILE *fp1, *fp2, *fopen();
     int ca, cb;
     char fname1[40], fname2[40] ;

     printf(“Enter first filename:”) ;
     gets(fname1);
     printf(“Enter second filename:”);
     gets(fname2);
     fp1 = fopen( fname1,  “r” );       /* open for reading */
     fp2 = fopen( fname2,  “r” ) ;       /* open for writing */

     if ( fp1 == NULL )      /* check does file exist etc */
     {
          printf(“Cannot open %s for reading \n”, fname1 );
          exit(1);    /* terminate program */
     }
     else if ( fp2 == NULL )
     {
          printf(“Cannot open %s for reading \n”, fname2 );
          exit(1);    /* terminate program */
     }
     else           /* both files opened successfully  */
     {
          ca  =  getc( fp1 ) ;
          cb  =  getc( fp2 ) ;

          while ( ca != EOF   &&   cb != EOF   &&   ca == cb  )
          {
               ca  =  getc( fp1 ) ;
               cb  =  getc( fp2 ) ;
          }
          if (  ca == cb )
               printf(“Files are identical \n”);
          else if ( ca !=  cb )
               printf(“Files differ \n” );
          fclose ( fp1 );
          fclose ( fp2 );
     }
}
```

Example 10.7

```
/* filecopy.c : Copy prog.c to prog.old */

#include <stdio.h>
void main()
{
     FILE *fp1, *fp2, *fopen();
     int c ;

     fp1 = fopen( “prog.c”,  “r” );       /* open for reading */
     fp2 = fopen( “prog.old”, “w” ) ; ../* open for writing */


     if ( fp1 == NULL )      /* check does file exist etc */
     {
          printf(“Cannot open prog.c for reading \n” );
          exit(1);    /* terminate program */
     }
     else if ( fp2 == NULL )
     {
          printf(“Cannot open prog.old for writing \n”);
          exit(1);    /* terminate program */
     }
     else           /* both files O.K. */
     {
          c = getc(fp1) ;
          while ( c != EOF)
          {
               putc( c,  fp2);    /* copy to prog.old */
               c =  getc( fp1 ) ;
          }

     fclose ( fp1 );          /* Now close files */
     fclose ( fp2 );
     printf(“Files successfully copied \n”);
     }
}
```

10.5 Formatted Input Output with File Pointers

Similarly there are equivalents to the functions printf and scanf which read or write data to files. These are called fprintf and fscanf. The functions are used in the same way, except that the fprintf and fscanf take the file pointer as an additional first argument.

fprintf() is used to write a line of text to the file specified by file pointer

fprintf(filepointer, “control string”, variable list);

Example:

fprintf(fp,”%d %f %s”, empno, sal,empname);

this statement writes three dta specified in the variable list to the file pointed by file pointer fp.

Similarly fscanf() is used to read a line of text from the file specified by file pointer

fscanf(filepointer, “control string”, variable list);

Example:

fprintf(fp,”%d %f %s”, &empno, &sal,empname);

Formatted Input Output with Strings

These are the third set of the printf and scanf families. They are called sprintf and sscanf.

sprintf
: puts formatted data into a string which must have sufficient space allocated to hold it. This can be done by declaring it as an array of char. The data is formatted according to a control string of the same form as that for p rintf.

sscanf

takes data from a string and stores it in other variables as specified by the control string. This is done in the same way that scanf reads input data into variables. sscanf is very useful for converting strings into numeric v values.

Whole Line Input and Output using File Pointers

Predictably, equivalents to gets and puts exist called fgets and fputs. The programmer should be careful in using them, since they are incompatible with gets and puts. gets requires the programmer to specify the maximum number of characters to be read. fgets and fputs retain the trailing newline character on the line they read or write, wheras gets and puts discard the newline.

When transferring data from files to standard input / output channels, the simplest way to avoid incompatibility with the newline is to use fgets and fputs for files and standard channels too.

For Example, read a line from the keyboard using

```
  fgets(data_string, 80, stdin);
```

and write a line to the screen using

```
  fputs(data_string, stdout);
```

Special Characters

C makes use of some 'invisible' characters which have already been mentioned. However a fuller description seems appropriate here.

NULL, The Null Pointer or Character

NULL is a character or pointer value. If a pointer, then the pointer variable does not reference any object (i.e. a pointer to nothing). It is usual for functions which return pointers to return NULL if they failed in some way. The return value can be tested. See the section on fopen for an example of this.

NULL is returned by read commands of the gets family when they try to read beyond the end of an input file.

Where it is used as a character, NULL is commonly written as '\0'. It is the string termination character which is automatically appended to any strings in your C program. You usually need not bother about this final \0', since it is handled automatically. However it sometimes makes a useful target to terminate a string search. There is an example of this in the string_length function example in the section on Functions in C.

EOF, The End of File Marker

EOF is a character which indicates the end of a file. It is returned by read commands of the getc and scanf families when they try to read beyond the end of a file.

Direct Input and Output Functions

*size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream)*

fread reads from stream into the array ptr at most nobj objects of size size. fread returns the number of objects read; this may be less than the number requested. feof and ferror must be used to determine status.

*size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *stream)*

fwrite writes, from the array ptr, nobj objects of size size on stream. It returns the number of objects written, which is less than nobj on error.

10.6 File Positioning Functions

The following function are related to random access to the file

*int fseek(FILE *stream, long offset, int origin)*

fseek sets the file position for stream; a subsequent read or write will access data beginning at the new position. For a binary file, the position is set to offset characters from origin, which may be SEEK_SET (beginning), SEEK_CUR (current position), or SEEK_END (end of file). For a text stream, offset must be zero, or a value returned by ftell (in which case origin must be SEEK_SET). fseek returns non-zero on error.

*long ftell(FILE *stream)*

ftell returns the current file position for stream, or -1 on error.

*void rewind(FILE *stream)*

rewind(fp) is equivalent to fseek(fp, 0L, SEEK_SET); clearerr(fp).

*int fgetpos(FILE *stream, fpos_t *ptr)*

fgetpos records the current position in stream in *ptr, for subsequent use by fsetpos. The type fpos_t is suitable for recording such values. fgetpos returns non-zero on error.

*int fsetpos(FILE *stream, const fpos_t *ptr)* `fsetpos` positions `stream` at the position recorded by `fgetpos` in `*ptr`. `fsetpos` returns non-zero on error.

10.7 Error Functions

Many of the functions in the library set status indicators when error or end of file occur. These indicators may be set and tested explicitly. In addition, the integer expression errno (declared in <errno.h>) may contain an error number that gives further information about the most recent error.

`void clearerr(FILE *stream)`

`clearerr` clears the end of file and error indicators for `stream`.
`int feof(FILE *stream)`

`feof` returns non-zero if the end of file indicator for `stream` is set.

`int ferror(FILE *stream)`
`ferror` returns non-zero if the error indicator for `stream` is set.

`void perror(const char *s)`
`perror(s)` prints `s` and an implementation-defined error message corresponding to the integer in `errno`, as if by

```
fprintf(stderr, "%s: %s\n", s, "error message");
```

10.8 Command-line Arguments

In environments that support C, there is a way to pass command-line arguments or parameters to a program when it begins executing. When main is called, it is called with two arguments. The first (conventionally called argc, for argument count) is the number of command-line arguments the program was invoked with; the second (argv, for argument vector) is a pointer to an array of character strings that contain the arguments, one per string. We customarily use multiple levels of pointers to manipulate these character strings.

The simplest illustration is the program echo, which echoes its command-line arguments on a single line, separated by blanks. That is, the command

```
echo hello, world
```

prints the output

```
hello, world
```

By convention, argv[0] is the name by which the program was invoked, so argc is at least 1. If argc is 1, there are no command-line arguments after the program name. In the example above, argc is 3, and argv[0], argv[1], and argv[2] are "echo", "hello,", and "world" respectively. The first optional argument is argv[1] and the last is argv[argc-1]; additionally, the standard requires that argv[argc] be a null pointer.

The first version of `echo` treats `argv` as an array of character pointers:

```
#include <stdio.h>
/* echo command-line arguments; 1st version */
main(int argc, char *argv[])
{
int i;
for (i = 1; i < argc; i++)
printf("%s%s", argv[i], (i < argc-1) ? " " : "");
printf("\n");
return 0;

}
```

Since argv is a pointer to an array of pointers, we can manipulate the pointer rather than index the array. This next variant is based on incrementing argv, which is a pointer to pointer to char, while argc is counted down:

```
#include <stdio.h>
/* echo command-line arguments; 2nd version */
main(int argc, char *argv[])
{
while (--argc > 0)
printf("%s%s", *++argv, (argc > 1) ? " " : "");
printf("\n");
return 0;
}
```

Since argv is a pointer to the beginning of the array of argument strings, incrementing it by 1 (++argv) makes it point at the original argv[1] instead of argv[0]. Each successive increment moves it along to the next argument; *argv is then the pointer to that argument. At the same time, argc is decremented; when it becomes zero, there are no arguments left to print.Alternatively, we could write the printf statement as

```
printf((argc > 1) ? "%s " : "%s", *++argv);
```

This shows that the format argument of printf can be an expression too.

The following program is used to print lines that match pattern from arg.

Example 10.8

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

```
#define MAXLINE 1000
int getline(char *line, int max);
/* find: print lines that match pattern from 1st arg */
main(int argc, char *argv[])
{
char line[MAXLINE];
int found = 0;
if (argc != 2)
printf("Usage: find pattern\n");
else
while (getline(line, MAXLINE) > 0)
if (strstr(line, argv[1]) != NULL) {
printf("%s", line);
found++;
}
return found;
}
```

The standard library function strstr(s,t) returns a pointer to the first occurrence of the string t in the string s, or NULL if there is none. It is declared in <string.h>.

10.9 I/O Redirection

Your assignment shell must support i/o-redirection on both *stdin* and *stdout*. i.e. the command line:

```
programname arg1 arg2 < inputfile > outputfile
```

will execute the program programname with arguments arg1 and arg2, the *stdin* FILE stream replaced by inputfile and the *stdout* FILE stream replaced by outputfile.

With output redirection, if the redirection character is > then the outputfile is created if it does not exist and truncated if it does. If the redirection token is >> then outputfile is created if it does not exist and appended to if it does.

Note: you can assume that the redirection symbols, < , > & >> will be delimited from other command line arguments by white space - one or more spaces and/or tabs.

I/O redirection is accomplished in the child process immediately after the fork and before the exec command. At this point, the child has inherited all the filehandles of its parent and still has access to a copy of the parent memory. Thus, it will know if redirection is to be performed, and, if it does change the *stdin* and/or *stdout* file streams, this will only effect the child not the parent.

You can use open to create file descriptors for inputfile and/or outputfile and then use dup or dup2 to replace either the *stdin* descriptor (STDIN_FILENO from stdio.h) or the *stdout* descriptor (STDOUT_FILENO from stdio.h).

However, the easiest way to do this is to use freopen. This function is one of the three functions you can use to open a standard I/O stream.

```
#include <stdio.h>

FILE *fopen(const char *pathname, const char * type);

FILE *freopen(const char * pathname, const char * type, FILE *fp);

FILE *fdopen(int filedes, const char * type);

                    All three return: file pointer if OK, NULL on error
```

The differences in these three functions are as follows:

1. fopen opens a specified file.
2. freopen opens a specified file on a specified stream, closing the stream first if it is already open. This function is typically used to open a specified file as one of the predefined streams, stdin, stdout, or stderr.
3. fdopen takes an existing file descriptor (obtained from open, etc) and associates a standard I/O stream with that descriptor - useful for associating pipes etc with an I/O stream.

The type string is the standard open argument:

type	Description
r or rb	open for reading
w or wb	truncate to 0 length or create for writing
a or ab	append; open for writing at end of file, or create for writing
r+ or r+b or rb+	open for reading and writing
w+ or w+b or wb+	truncate to 0 length or create for reading and writing
a+ or a+b or ab+	open or create for reading and writing at end of file

where b as part of type allows the standard I/O system to differentiate between a text file and a binary file.

Thus:

freopen("inputfile", "r", stdin);

would open the file inputfile and use it to replace the standard input stream, stdin.

You may want to use the access function to check on existence or not of the files:

```
#include <unistd.h>

int access(const char *pathname, int mode);

                                    Returns: 0 if OK, -1 on error
```

The mode is the bitwise OR of any of the constants below:

mode	Description
R_OK	test for read permission
W_OK	test for write permission
X_OK	test for execute permission
F_OK	test for existence of file

Looking at the assignment specification, stdout redirection should also be possible for the internal commands: dir, environ, echo, & help.

10.10 Let us Sum up

In this lesson, we discussed about basic concepts of files, various classification of files, reading and writing data with files. We also discussed about, how to interact with program from command line using command line argument. We also elaborated about the purpose of using I/O direction. Errors during implementation of files were also discussed.

10.11 Points for discussion

1. Define text file

2. How to open a file

3. Explain about various modes of opening a file.

4. How to identify errors in file handling?

10.12 Check your progress

1. How to identify errors in file handling?

Each function in the library set status indicators when error or end of file occur. These indicators may be set and tested explicitly. In addition, the integer expression errno (declared in <errno.h>) may contain an error number that gives further information about the most recent error.

`void clearerr(FILE *stream)`

`clearerr` clears the end of file and error indicators for `stream`.
`int feof(FILE *stream)`

`feof` returns non-zero if the end of file indicator for `stream` is set.

`int ferror(FILE *stream)`
`ferror` returns non-zero if the error indicator for `stream` is set.

2. How to close a file

Once the purpose of opening a file is over then it is the responsibility of user to close the corresponding file using the command fclose.

fclose(file pointer);

10.13 Lesson-end Activities

1. Do you feel the file in c can acts as the replacement of database?

2. what is the purpose of using files?

10.14 References

Peter Aitken, Teach Yourself C in 21 Days, Fourth Edition, Sams Publisher.
Brian W. Kernighan and Dennis M. Ritchie, The C programming Language, Prentice-Hall in 1988
E.Balagursamy, Programming in Ansi C, TATA MCGraw Hill
http://vergil.chemistry.gatech.edu/resources/programming/c-tutorial
http://sysprog.net/
http://www.mycplus.com/cplus
http://www.programmersheaven.com/download/
http://en.literateprograms.org/
http://www.cs.utk.edu/~plank/plank/classes/

UNIT – IV:

LESSON - 11 : LINEAR DATA STRUCTURE

CONTENTS

11.0 Aims and Objectives

The objective of this lesson is to make the reader to understand about fundamental concepts of data structures and its implementation, various traversal and searching algorithms and importance of stack.

11.1 Introduction

A *data structure* is an arrangement of data in a computer's memory or even disk storage. An example of several common data structures are arrays, linked lists, queues, stacks, binary trees, and hash tables.

11.2 Characteristics of Data Structures

Data Structure	Advantages	Disadvantages
Array	Quick inserts Fast access if index known	Slow search Slow deletes Fixed size
Ordered Array	Faster search than unsorted array	Slow inserts Slow deletes Fixed size
Stack	Last-in, first-out acces	Slow access to other items
Queue	First-in, first-out access	Slow access to other items
Linked List	Quick inserts Quick deletes	Slow search
Binary Tree	Quick search Quick inserts Quick deletes *(If the tree remains balanced)*	Deletion algorithm is complex
Red-Black Tree	Quick search Quick inserts Quick deletes *(Tree always remains balanced)*	Complex to implement

2-3-4 Tree	Quick search Quick inserts Quick deletes *(Tree always remains balanced)* *(Similar trees good for disk storage)*	Complex to implement
Hash Table	Very fast access if key is known Quick inserts	Slow deletes Access slow if key is not known Inefficient memory usage
Heap	Quick inserts Quick deletes Access to largest item	Slow access to other items
Graph	Best models real-world situations	Some algorithms are slow and very complex
NOTE: The data structures shown above (with the exception of the array) can be thought of as Abstract Data Types (ADTs).		

11.3 Abstract Data Types

An *Abstract Data Type* (ADT) is more a way of looking at a data structure: focusing on what it does and ignoring how it does its job. A stack or a queue is an example of an ADT. It is important to understand that both stacks and queues can be implemented using an array. It is also possible to implement stacks and queues using a linked list. This demonstrates the "abstract" nature of stacks and queues: how they can be considered separately from their implementation.

To best describe the term Abstract Data Type, it is best to break the term down into "data type" and then "abstract".

11.3.1 Data type

When we consider a primitive type we are actually referring to two things: a data item with certain characteristics and the permissible operations on that data. An int in Java, for example, can contain any whole-number value from -2,147,483,648 to +2,147,483,647. It can also be used with the operators +, -, *, and /. The data type's permissible operations are an inseparable part of its identity; understanding the type means understanding what operations can be performed on it.

In Java, any class represents a data type, in the sense that a class is made up of data (fields) and permissible operations on that data (methods). By extension, when a data storage structure like a stack or queue is represented by a class, it too can be referred to as a data type. A stack is different in many ways from an int, but they are both defined as a certain arrangement of data and a set of operations on that data.

11.3 2 Abstract

Now lets look at the "abstract" portion of the phrase. The word abstract in our context stands for *"considered apart from the detailed specifications or implementation"*.

In Java, an Abstract Data Type is a class considered without regard to its implementation. It can be thought of as a "description" of the data in the class and a list of operations that can be carried out on that data and instructions on how to use these operations. What is excluded though, is the details of how the methods carry out their tasks. An end user (or class user), you should be told what methods to call, how to call them, and the results that should be expected, but not HOW they work.

We can further extend the meaning of the ADT when applying it to data structures such as a stack and queue. In Java, as with any class, it means the data and the operations that can be performed on it. In this context, although, even the fundamentals of how the data is stored should be invisible to the user. Users not only should not know how the methods work, they should also not know what structures are being used to store the data.

Consider for example the stack class. The end user knows that push() and pop() (amoung other similar methods) exist and how they work. The user doesn't and shouldn't have to know how push() and pop() work, or whether data is stored in an array, a linked list, or some other data structure like a tree.

11.4 List

A list is a sequential data structure, i.e. a collection of items accessible one after another beginning at the head and ending at the tail. It is a widely used data structure for applications which do not need random access. It differs from the stack and queue data structures in that additions and removals can be made at any position in the list.

We can define a list is a sequence of zero or more elements of certain type. We usually denote a list as a1, a2, . . . , an. Here n is the number of elements in the list which we also call as length of the list. a1 is the first element, an is the last element. There is an order among the elements defined by their position in the list. For instance, for each 1 _ i < n − 1, ai + 1 precedes ai+2 and succeeds ai. Note that there may be more than one type of order which can be defined on a set of elements; one such order could be based on the value of the elements (smaller valued lements precede larger valued elements). However, in the case of list, the order defined is just defined by the numbering of the elements based on their positions in the list. Now we define the operations which can be performed on a list. Later we shall see that there are two implementations possible for supporting these operations. One implementation is array based and another one is linked list based. In the following functional description of list, L denotes the given list, x denotes an

element, and p denotes the position. Note that p essentially will be address of some element in the list and its actual type will depend upon the implementation. Please note that the following operations serve as an example for functional description of a list and are not necessarily unique. You may define another set of similar operations with possibly different description
.
• **CreateList(L)** : creates an empty list L.

• **MakeNULL(L)** : converts L into an empty list.

• **Locate(x,L)** : returns the location of an element x in list L if it is present and otherwise reports error.

• **Retrieve(p,L)** : returns the elements present at location p in list L.

• **Insert(x, p,L)** : insert a new element x after the element at position p in the list L

• **Delete(p,L)** : delete the element that follows the element at position p in list L.

• **Next(p,L)** : return the position of the element succeeding the element stored at position p.

• **Previous(p,L)** : return the position of the element preceding the element stored at position p.

• **IsEmpty(L)** : return true if the list L is empty

Note that position p used in operations defined above depends upon the specific implementation we have for the data structure as well as the specific application. For example, in case of array based implementation, position may refer to an index in the array. However, for Linked list based implementation, p is the address of some specific element in the list.

11.4.1 Array based implementation of lists

It is quite easy to implement List using an array and a variable for keeping the count of the number of elements in the list at any moment of time. For sake of simplicity, we assume that the elements of the list are integers.the following function will represent some general functions on list.

Example11.1

```
static final int INIT SIZE = 10;
int MyList[];
INT ListSize;
```

```
ArrayList(){
MyList = new int[INIT SIZE];
ListSize = 0;
}

void resize()
{
int temp[] = new int[size() * 2];
arraycopy(MyList, 0, temp, 0, size());
MyList = temp;
}

void Insert(int x, int pos)
 {
if( ListSize == MyList.length )
resize();
if( pos<ListSize) {
for(int i = ListSize; i > pos+1; i–){
MyList[i] = MyList[i-1];
}
MyList[pos] = x;
ListSize++;
}
}

int Remove(int pos)
{
int current = MyList[pos];
for(int i = pos+1; i < ListSize-1; i++){
MyList[i] = MyList[i+1];
}
ListSize–;
2
return current;
}

int Retrieve(int pos)
{
return MyList[pos];
}

int Locate(int x)
 {
while(int i = 0, int flag==0; i< ListSize && flag==0) {
if(MyList[i]==x) flag=1; else i++;
}
if(flag==1) return i; else return -1;
}
int Next(int pos) {
if(pos >=1 && pos < ListSize)
return pos+1;
}

int Previous(int pos)
 {
if(pos >1 && pos < ListSize)
return pos-1;
}
int Size() {
return ListSize;
}

boolean IsEmpty()
 {
return ListSize==0
}

void MakeNULL()
{
MyList = new int[10];
ListSize = 0;
}
}
```

It is easy to verify that the array based implementation described above achieves following time complexities for operations on list.

11.4.2 Operation Time complexity and operations

Insert	O(n)
Delete	O(n)
Locate()	O(n)
Retrieve()	O(1)
Next()	O(1)
Previous()	O(1)
IsEmpty	O(1)

11.5 More about List

The list is a flexible abstract data type. It is particularly useful when the number of elements to be stored is not known before running a program or can change during running. It is well suited to sequential processing of elements because the next element of a list is readily accessible from the current element. It is less suitable for random access to elements as this requires a slow linear search. Two forms of list are considered, flat lists of elements and hierarchical.

The definition of list can be read as follows: a list of e is (=) either `nil' or (|) is `cons' applied to an element of type e and a list. Nil represents the empty list. Null tests for the empty list. The head of a list is the first element in the list. The tail of a list is everything but the first element and is a list not an element. Cons *cons*tructs a list given an element and a list.

Given the basic operations on lists, a number of useful functions can be defined.

```
length(nil) = 0
length(cons(e,L)) = 1+length(L)
```

Note that the length function is recursively defined, as is the list type. While there are infinitely many lists, there are only two cases that length must consider. One case is the empty list and the other is the non-empty list. These two cases correspond to the two cases in the definition of the list type - nil and cons(e,L

Append joins two lists together; for example, append([1,2], [3,4]) = [1,2,3,4]. If the first list is empty the result is the second list, otherwise the result is the first element of the first list followed by the result of appending the tail of the first list to the second list. Note that if the length of the first list is n, append takes O(n) time. It makes a copy of this list which finally points to the second; it does not change the first one. In this way append does not cause any side-effect.

```
L ----> 1--->2nil                     L := [1,2]

L'----------->----------> 3--->4nil    L':= [3,4]
                  ^
                  |
L"----> 1--->2--->|

           after L":=append(L,L')  = [1, 2, 3, 4]
```

Note however that if the contents of L' are now changed, by some means, then so also are the contents of L". This kind of side-effect is a common cause of program bugs.

The merge function produces one sorted list when given two sorted lists.

```
merge(nil, nil) = nil   |
merge(nil, L)   = L     |
merge(L, nil)   = L     |
merge(L, L')    =            % ~null(L) & ~null(L')
   if head L < head L' then
```

```
         cons(head L , merge(tail L, L'))
      else {head L >= head L'}
         cons(head L', merge(L, tail L'))

   %this version allows duplicates
```

The original lists are unchanged. Note the similarities with the array and file merge operations.

The *map* function applies a function f to every element of a list and produces a new list. In this way map f processes a single data structure as a whole.

```
   map(f, nil) = nil   |
   map(f, (cons(e,L))) = cons(f(e), map(f, L) )

   eg. map(factorial, [1,2,3,4]) = [1,2,6,24]
```

Map is known as mapcar in some works on Lisp.

The *reduce* function inserts a binary function, or operator, between every element in a list. If the list is empty, reduce returns an identity or zero element z.

```
   reduce(f, z, nil) = z   |
   reduce(f, z, cons(e,L)) = f(e, reduce(f, z, L))
```

Reduce(plus, 0, L) adds up the elements in a list of integers.

```
   eg. reduce(plus, 0, [1,2,3,4])
          where plus(a,b)=a+b
     = 1 + reduce(plus, 0, [2,3,4])
     = 1 +(2 + reduce(plus, 0, [3,4]))
     = 1 +(2 +(3 + reduce(plus, 0, [4])))
     = 1 +(2 +(3 +(4 + reduce(plus, 0, nil))))
     = 1 +(2 +(3 +(4 + 0)))
     = 10
```

Reduce(times, 1, L) multiplies the elements in a list of integers together.

```
   eg. reduce(times, 1, [1,2,3,4]) = 24
          where times(a,b)=a*b
     = 24
```

These functions for manipulating lists can be combined to give many short but powerful programs. For example, the program

```
   lookup(x, L) = reduce(or, false, map(equals_x, L))
      where equals_x(y) = x=y
```

is a search program and returns true only when x is in the list L. Map(equals_x, L) returns a list of boolean values, at least one value is true only when x is in L. Reduce or's all these values together. Processing lists and other linked data structures is important in functional programming languages such as Haskell, Lisp and ML. This style of combining simple functions on these data structures to achieve complex objectives is a powerful programming technique.

As always, the issue of efficiency must be attended to. For example, there are various ways to reverse a list. The simplest way but not the best is:

```
reverse(nil) = nil  |
reverse(cons(e,L)) = append( reverse(L), [e] )
```

To reverse the empty list do nothing. To reverse a non-empty list, reverse the tail of the list and append the first element of the original list. Given a list of length n, an append operation takes O(n) time on average. There are n calls to append so this reverse takes O(n^2) time. Note that the element `e' and the list containing it `[e]' have different types.

A fast list-reversal algorithm takes O(n) time and uses the *accumulating parameter* technique.

```
reverse(L) = reverse1(L, nil)
  where
    reverse1(nil,L) = L  |
    reverse1(cons(e,L),L') = reverse1(L,cons(e,L'))
```

The local function reverse1 does the real work. Its first parameter is the input list which shrinks at each step. Its second parameter is the accumulating parameter which grows at each step as the head of the first is attached to it. When the first list has shrunk to nothing, the second contains all the original elements in reverse order and this is the final result. There are n calls to reverse1 each of which does a constant amount of work so the algorithm takes O(n) time overall.

Note that although there are infinitely many possible lists there are really only two types of list - empty ones and non-empty ones. Almost every function that manipulates a list must deal with these two cases.

```
f(nil) = ..... |
f(cons(e,L)) = ....
```

If one of these cases is missing then there is *probably* an error. Processing the empty list is usually easy. Many functions return a constant - 0, 1, true, false or nil - in this case. Processing the non-empty list, cons(e,L), varies from application to application but often involves e or f(L) or both.

```
f(nil) = some constant |        -- a very
f(cons(e,L)) = ...e...f(L)...   -- common pattern
```

Look back at the various functions defined to identify this pattern in each case.

11.5.1 Implementation of Flat Lists

The most natural way to implement lists is by means of records/structs and pointers. They can also be implemented by arrays if necessary.

Flat Lists by Records and Pointers

The list type is defined as a pointer to a record (structure).

Example 11.2

```
#include "List.h"

List cons(ListElementType E, List L)  /*A*/
 { List L2;                           /*l*/
   L2 = (List)malloc(sizeof(Cell));   /*l*/
   L2->hd = E;                        /*i*/
   L2->tl = L;                        /*s*/
   return L2;                         /*o*/
 }/*cons*/                            /*n*/

int null(List L)/*predicate*/ { return L==NULL ? 1 : 0; }

ListElementType head(List L)
/* pre: not null(L) */
 { return L->hd; }

List tail(List L)
/* pre: not null(L) */
 { return L->tl; }

/* Basic List Operation */
```

The basic operations manipulate the pointers:

```
#include "List.h"

List cons(ListElementType E, List L)  /*A*/
 { List L2;                           /*l*/
   L2 = (List)malloc(sizeof(Cell));   /*l*/
   L2->hd = E;                        /*i*/
   L2->tl = L;                        /*s*/
   return L2;                         /*o*/
 }/*cons*/                            /*n*/

int null(List L)/*predicate*/ { return L==NULL ? 1 : 0; }

ListElementType head(List L)
/* pre: not null(L) */
 { return L->hd; }

List tail(List L)
/* pre: not null(L) */
 { return L->tl; }
```

```
/* Basic List Operation */
```

The extended operations on lists that were described previously are easily defined.

```
#include "List.h"

int length(List L)
 { if(L==NULL) return 0;
   else return 1+length(L->tl);
 }

/* Length of a List L (recursive) */
```

If the implementation of lists by pointers can be assumed, it is more efficient, if marginally less clear, to use in-line code such as L=nil(List) in place of null(L) and similarly for the other basic operations.

```
#include "List.h"

List append(List L1, List L2)
/* NB. L1 & L2 are not changed but L2 is shared with the result.
L.Allison */
 { if(L1==NULL) return L2;
   /*else*/ return cons(L1->hd, append(L1->tl, L2));
 }

/* Append Two Lists, L1 and L2 */
```

The list reversal functions can be coded as follows:

```
#include "List.h"

/* NB. Below, opL is an `accumulating' parameter  -- L.Allison */

List accumulateAndReverse(List inL, List opL)
 { if(inL == NULL) return opL;
   /*else*/ return accumulateAndReverse(inL->tl, cons(inL->hd, opL));
 }

List reverse(List L)
 { return accumulateAndReverse(L, NULL); }

/* O(|L|)-time List Reversal */
```

If the element type of a list can be printed by a library routine, a list can be printed by repeated calls. There are three cases to consider. The empty list should appear as `[ ]', a list of a single element should appear as `[e]' and a list of two or more elements should have the elements separated by commas `[a,b,...]'. The empty list can be treated as a special case by an **if** statement. The last two cases can be dealt with by using a loop and placing the printing of the current element before the **exit** and the printing of the comma after the exit. In this way one fewer commas than elements are printed.

```
#include "List.h"
```

```
/* NB. Assumes a list of int */

void WriteList(List L)
 { printf("[");              /* [ */
   if( L!=NULL )
      while(1)
       { printf("%d", L->hd); /* element */
         L = L->tl;
         if( L==NULL ) break;
         printf(",");         /* comma */
       }
   printf("]");              /* ] */
 }

/* Write a List, e.g. [] or [1] or [1,2,3] */
```

11.5.2 Flat Lists by Arrays

A list can be implemented by an array of records. Rather than using a pointer to indicate the next element, an integer is used to hold the index of the next element. Some convention, such as the use of zero, indicates the null list. This implementation has the disadvantage that an array of the maximum size of the list must be declared even if the list(s) grow and shrink. This method is useful in a language without dynamic storage.

11.5.3 Recursion and Iteration

Many of the operations on lists and other recursive data types are naturally expressed as recursive routines. Recursive routines have fewer state variables than iterative routines and are frequently easier to prove correct. However recursion does require system-stack space to operate. Iterative versions of many routines, particularly for the simpler operations, are straightforward. Note that we saw an iterative routine to print a list.

```
#include "List.h"

int lengthItr(List L)
 { int len;  len=0;
   while(L!=NULL)
    { len++; L=L->tl; }
   return len;
 }

/* Length of a List L (iterative) */
```

11. 6 Introduction to Linked Lists

Linked list is another form of list structure. We have been using arrays to store similar data linearly. While arrays are simple to understand and easy to implement in common situations, they do suffer from some drawbacks which are listed below:

- Arrays have fixed dimensions, even if we dynamically allocate the dimension it remains constant throughout. So there is a limit to the number of elements it can store.
- Operations such as insertion and deletion are pretty much difficult to implement and increases the overhead because these operations require elements in the array to be physically shifted.

Linked lists overcome these drawbacks and are commonly used to store linear data.

Actually elements of linked lists (called as nodes) store two information, data and the link (pointer) pointing to the next elements (node).

The elements (nodes) are linked sequentially with the help of link pointers. So we can say that linked lists are collection of nodes which have data and are linked sequentially so that all the nodes or elements are grouped together.

In programming sense, linked lists are classes whose general form is:

```
class node
{
 public:
   data-type info;
   node *link;
};
```

Here info stores the actual data while link stores the memory address of the next node, which forms the link between the nodes.

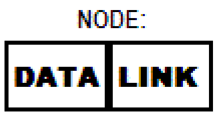


FIGURE 11.6.1.: Graphical representation of a node

Following figure illustrates the growing of linked lists, the node which has its link as NULL is the last element in the linked list.

Start Pointer
NULL
Red color denotes Memory address

FIGURE 11.6.2.: Linked lists Initial stage

11.6.1 Let us now discuss a bit about how a linked list grows:

1. We have a pointer that stores the memory address of the first element in the linked list, represented as the start pointer (of type node). It is NULL to begin with as we don't have any element in the list.

2. As an element is added, the start pointer is made to point at it and since for now the first element is the last element therefore its link is made to be NULL

3. After the addition of each element the link pointer of the previously last element is made to point at new last element. This step continues…

In this way the number of nodes in a linked list can grow or shrink over time as far as memory permits.
Just as variables of pre-def data type are dynamically allocated, same way objects of class can also be dynamically allocated. While the method of doing this is no different but the way members of such objects are accessed is a bit different. This is why I thought of writing a separate article on it.
In the previous lesson *Introduction to Linked Lists*, we introduced the basic concept of linked list. To make the program (an the article) as simple as possible, we discussed only the addition and display of nodes in the linked list although necessary we didn't't discussed the deletion of node in the previous discussion. Here is the procedure for deletion.

11.6.2 Deletion of node (elements) from a linked list

The node to be deleted can be represented by many ways but here we will be representing it by its info. So if we have the following linked list

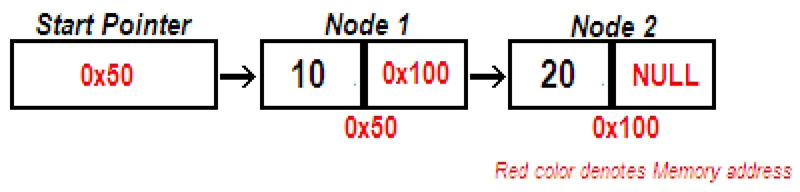


Figure.11.6.3 List expansion

And we want to delete node1 then we will express it by its info part (i.e. 10).

The main theory behind deletion of nodes is pretty simple. We need to make the link pointer of the node before the target node (to be deleted) to point at the node after the target node. Suppose if we wish to delete node having info as 10 from the above linked list then it will be accomplished as below:

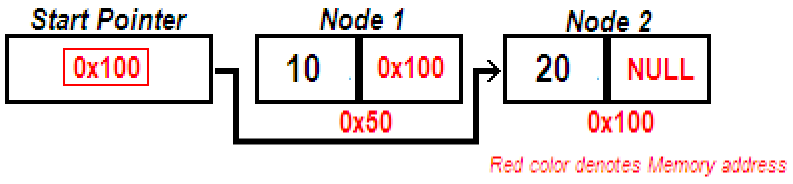


Figure.11.6.4 Deletion of a node

Now since node1 is orphan and has no meaning, it can be deleted to free-up memory as represented below:

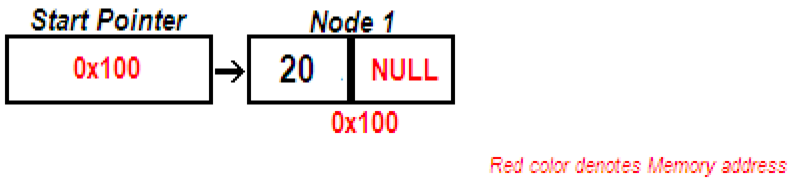


Figure.12.3.3 After deleting a node

11.7 Let us Sum up

In this lesson, we discussed about fundamental concepts of data structures and its implementation, data types and abstract data types, concept of list various operations involved in the list and its algorithmic implementations.

11.8 Points for discussion

1. Define the term abstract data type.
2. Explain various characteristics of data structure.
3. How to define list.
4. Explain various operatios involved with list.

11.9 Check you progress

1. What is meant by data structure?

A *data structure* is an arrangement of data in a computer's memory or even disk storage. An example of several common data structures are arrays, linked lists, queues, stacks, binary trees, and hash table

2. What is meant by list?

The list is a flexible abstract data type. It is particularly useful when the number of elements to be stored is not known before running a program or can change during running. It is well suited to sequential processing of elements because the next element of a list is readily accessible from the current element. It is less suitable for random access to elements as this requires a slow linear search. Two forms of list are considered, flat lists of elements and hierarchical.

11. 10 Lesson-end activities

1. Explain any two applications of list
2. What is meant by primitive data structures?
3. Write any one of operation on list?
4. Is it possible to implement list using pointers?

11.11 References

Peter Aitken, Teach Yourself C in 21 Days, Fourth Edition, Sams Publisher.
Brian W. Kernighan and Dennis M. Ritchie, The C programming Language, Prentice-Hall in 1988
E.Balagursamy, Programming in Ansi C, TATA MCGraw Hill
http://sysprog.net/
http://www.mycplus.com/cplus
http://www.programmersheaven.com/download/
http://en.literateprograms.org/
http://www.cs.utk.edu/~plank/plank/classes/
http://www.go4expert.com/forums/
http://www.cs.bu.edu/teaching/c/

LESSON - 12 : STACK

CONTENTS

12.0 Aims and Objectives

The objective of this lesson is to motivate the readers to understand the concept of stack and its applications, operations on stack. This lesson also describes the usage of queue also. This helps readers to get clear information about stack and queue.

12.1 Stack : Introduction

Represents stacks of arbitrary objects. This class reflects a view of a stack as a kind of list, albeit one with some unique operations (push, pop, etc.), and in which some of the traditional list operations (find, delete, etc.) do nothing. The overall result is a class whose instances behave like standard stacks. This class was created to support the text *Algorithms & Data Structures: The Science of Computing* by Doug Baldwin and Greg Scragg.

12.2 Operations on Stack

Operation	***Description***	***Requirement***
Push	This operation adds or pushes another item onto the stack.	The number of items on the stack is less than n.
Pop:	This operation removes an item from the stack.	The number of items on the stack must be greater than 0.
Top:	This operation returns the *value* of the item at the top of the stack.	Note: It does not remove that item.
Is Empty:	This operation returns true if the stack is empty and false if it is not.	
Is Full:	This operation returns true if the stack is full and false if it is not.	

These are the basic operations that can be performed on a stack. Let us now go on and look at them in detail.

12. 3 Simple representation of a stack

In computer science, a **stack** is a temporary abstract data type and data structure based on the principle of *Last In First Out (LIFO)*. Stacks are used extensively at every level of a modern computer system. For example, a modern PC uses stacks at the architecture level, which are used in the basic design of an operating system for interrupt handling and operating system function calls. Among other uses, stacks are used to run a Java Virtual Machine, and the Java language itself has a class called "Stack", which can be used by the programmer. The stack is ubiquitous.

A *stack-based* computer system is one that stores temporary information primarily in stacks, rather than hardware CPU registers (a *register-based* computer system).

Abstract data type

As an abstract data type, the stack is a container of nodes and has two basic operations: *push* and *pop*. *Push* adds a given node to the top of the stack leaving previous nodes below. *Pop* removes and returns the current top node of the stack. A frequently used metaphor is the idea of a stack of plates in a spring loaded cafeteria stack. In such a stack, only the top plate is visible and accessible to the user, all other plates remain hidden. As new plates are added, each new plate becomes the top of the stack, hiding each plate below, *pushing* the stack of plates down. As the top plate is removed from the stack, they can be used, the plates *pop* back up, and second plate becomes the top of the stack. Two important principles are illustrated by this metaphor, the Last In First Out principle is one. The second is that the contents of the stack are hidden. Only the top plate is visible, so to see what is on the third plate, the first and second plates will have to be removed.

Operations

In modern computer languages, the stack is usually implemented with more operations than just "push" and "pop". The length of a stack can often be returned as a parameter. Another helper operation *top* (also known as *peek* and *peak*) can return the current top element of the stack without removing it from 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*).

***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--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. Let's see what order that is by looking at a stack of letters...

Suppose we have a stack that can hold letters, call it `stack`. What would a particular sequence of `Push` and `Pops` do to this 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
```

Now let's remove an item, `letter = Pop(stack)`, giving:

```
-----              -----
| A |  <-- top     | B |
-----              -----
stack              letter
```

And finally, one more addition, `Push(stack, C)`, giving:

```
-----
| C |  <-- top
```

```
-----
| A |
-----
stack
```

You'll notice that the stack enforces a certain *order* to the use of its contents, i.e., the *Last* thing *In* is the *First* thing *Out*. Thus, we say that a stack enforces **LIFO** order.

Now we can see one of the uses of a stack...To reverse the order of a set of objects.

12.4 Implementing a stack with an array:

Let's think about how to implement this stack in the C programming language.

First, if we want to store letters, we can use type `char`. Next, since a stack usually holds a bunch of items with the same type (e.g., `char`), we can use an array to hold the contents of the stack.

Now, consider how we'll use this array of characters, call it `contents`, to hold the contents of the stack. At some point we'll have to decide how big this array is; keep in mind that a normal array has a fixed size.

Let's choose the array to be of size 4 for now. So, an array getting **A**, then **B**, will look like:

```
-----------------
| A | B |   |   |
-----------------
  0   1   2   3
contents
```

Is this array sufficient, or will we need to store more information concerning the stack?

Answer: We need to keep track of the *top* of the stack since not all of the array holds stack elements.

*What **type** of thing will we use to keep track of the top of the stack?*

Answer: One choice is to use an integer, `top`, which will hold the array index of the element at the top of the stack.

Example:

Again 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
letter = 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`.

Let's use this implementation of the stack with **contents** and **top** fields.

What happens if we apply the following set of operations?

1. `Push(stack, 'D')`
2. `Push(stack, 'E')`
3. `Push(stack, 'F')`
4. `Push(stack, 'G')`

giving:

```
stack (made up of 'contents' and 'top')
-----------------   -----
| D | E | F | G |   | 3 |
-----------------   -----
  0   1   2   3      top
contents
```

*and then try to add **H** with* `Push(stack, 'H')`*?*

Thus, what is the disadvantage of using an array to implement a stack?

Dynamically-sized stack:

Now, we will add one more *choice* to how we'll implement our stack. We want to be able to decide the maximum size of the stack at run-time (not compile-time).

Thus, we cannot use a regular array, but must use a pointer to a dynamically-allocated array.

*Now, will we need to keep track of any more information besides the **contents** and **top**?*

Answer: Yes! We'll need to keep the size of this array, i.e., the *maximum size* of the stack. We'll see why this is necessary as we write the code.

12.4.1 C code implementation:

Now, let's think about how to actually code this *stack of characters* in C.

It is usually convenient to put a data structure in its own module, thus, we'll want to create files `stack.h` and `stack.c`.

Now, there are 2 main parts to a C data structure: the *data types* needed to keep track of a stack and the *functions* needed to implement stack operations.

1. The main *data type* we need is a type that people can use to declare new stacks, as in:
2. `type-of-a-stack s1, s2;`
3. Some of the functions we'll need come directly from the operations needed for an abstract stack, like:
4. `StackPush(ref-to-s1, 'A');`
5. `ch = StackPop(ref-to-s2);`

   Notice how each stack operation needs some way to refer to a specific stack (so that we can have more than one stack at a time).

We may need to add a few other operations to help implement a stack. These will become apparent as we start to implement the stack. Remember that we need to put prototypes for each stack function in `stack.h` and put the function definitions (bodies) in `stack.c`.

Before we ponder the details of the stack functions, we should decide on the types we need...

12.4.2 Data types for a stack:

For the array implementation, we need to keep track of (at least) the array **contents** and a **top** index. *How could we combine these 2 into a single C construct of type **stackT**?*

Answer: Put them into a `struct`.

So, our stack types become:

```
typedef char stackElementT;  /* Give it a generic name--makes  */
                             /* changing the type of things in */
                             /* the stack easy.                */

typedef struct {
  stackElementT *contents;
  int top;
  /* Other fields? */
} stackT;
```

Note that the *contents* is a pointer since it will be dynamically-allocated.

Are any other fields needed? Well, remember that the maximum size of the array is determined at run-time...We'll probably need to keep that value around so that we can tell when the stack is full... The final type, thus, is:

```
typedef struct {
  stackElementT *contents;
  int top;
  int maxSize;
} stackT;
```

These types should go in the interface, `stack.h`.

12.4.3 Filling in stack functions:

Now that we've decided on the data types for a stack, let's think about the functions we need...

First, we need the standard stack operations:

```
        StackPush()
        StackPop()
        StackIsEmpty()
```

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:

```
        StackInit()
        StackDestroy()
```

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:

```
StackIsFull()
```

12.4.4 StackInit():

The first function we'll implement is `StackInit()`. It will need to set up a `stackT` structure so that it represents an *empty stack*.

Here is what the prototype for `StackInit()` looks like...

```
void StackInit(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 StackInit(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.

12.4.5 StackDestroy():

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 */
}
```

12.4.6 StackIsEmpty() and StackIsFull():

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 StackIsEmpty(stackT stack);
int StackIsFull(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 StackIsEmpty(stackT *stackP);
int StackIsFull(stackT *stackP);
```

Emptiness

Now, testing for emptyness 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 StackIsEmpty(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 StackIsFull(stackT *stackP)
{
  return stackP->top >= stackP->maxSize - 1;
}
```

This illustrates the importance of keeping the maximum size around in a field like `maxSize`.

12.4.7 StackPush():

Now, pushing onto the stack requires the stack itself as well as *something to push*. So, its prototype will look like:

```
void StackPush(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 StackPush(stackT *stackP, stackElementT element)
{
  if (StackIsFull(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.

12.4.8 StackPop():

Finally, popping from a stack only requires a stack parameter, but the value popped is typically returned. So, its prototype will look like:

```
stackElementT StackPop(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 StackPop(stackT *stackP)
{
  if (StackIsEmpty(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.

Stack module:

Finally, don't forget that we are putting this stack in its own module. The stack types and function prototypes should go in `stack.h`. The stack function definitions should go in `stack.c`.

People that need to use the stack must include `stack.h` and link their code with `stack.c` (really, link their code with its object file, `stack.o`).

Finally, since we wrote the types and functions for a stack, we know how to use a stack. For example, when you need stacks, declare stack variables:

```
stackT s1, s2;
```

When you pass a stack to stack functions, pass it by reference:

```
StackInit(&s1, 10);
StackPush(&s1, 'Z');
```

Example 12.1

This program is another example for operations on stack with integer data.

```
#include<stdio.h>
#define length 10
#include<conio.h>
#include<stdlib.h>
int top=-1;
int stack[length];
void push()
{
    int data;
    if(top+1==length)
    {
        printf("stack overflow\n");
        return;
    }
    top++;
    printf("enter the data ");
    scanf("%d",&data);
    stack[top]=data;
}
int pop()
{
    int temp_var;
```

```
        if(top==-1)
        {
                puts("stack is underflow");
                return(-78799);
        }
        temp_var=stack[top];
        top--;
        return(temp_var);
}
void main()
{
 int i,ch;
 clrscr();
 do{
 printf("\n");
 puts("1.push");
 puts("2.pop");
 puts("3.exit");
 puts("enter choice");
 scanf("%d",&ch);
//} while(1);
switch(ch)
{
case 1:push();
printf("list is ");
for(i=0;i!=top+1;i++)
printf("%d ",stack[i]);
break;
case 2:printf("data poped=%d  ",pop());
printf("list is ");
for(i=0;i!=top+1;i++)
printf("%d ",stack[i]);

break;
case 3:exit(0);
}
} while(1);
getch();
}
```

12.5 Implementation of stack using linked list

Let us first see how we can implement a stack without being concerned with header objects. For the purpose of this section, we assume that we have a linked list contained in a variable named `l` with the elements of the stack (if you do not appreciate the choice of such a short name, read the section on naming conventions). We also assume that we implement the operations as side effects on the variable `l`. For the implementation of `push` we further assume that a variable `element` contains the element to push onto the stack. We can take advantage of the operation `cons` that we defined in the section on linked lists, which gives the following code:

```
  l = cons(element, l);
```

The implementation of `empty` is particularly simple:

```
  return l == NULL;
```

or simply:

```
  return !l;
```

For the implementation of `pop` we need to check that the stack is not empty before attempting to remove the top element. Here is the code:

```
  assert(!empty(l));
  {
    list temp = l;
    l = l -> next;
    free(temp);
  }
```

If we have a garbage collector, we can do even better, like this:

```
  assert(!empty(l));
  l = l -> next;
```

Finally, for the `top` operation, we also need to check whether the stack is empty before attempting the operation:

```
  assert(!empty(l));
  return l -> element;
```

12.5.1 Turning it into a module

Now that we got our programming idioms done, let us look at how to turn these functions into a module. Recall that we divide a module into two distinct files, the *header* file (or the `.h` file) and the *implementation* file (or the `.c` file). Let us call our module `stack` and use that as a prefix for the operations.

Here is the header file:

```
  #ifndef STACK_H
  #define STACK_H

  struct stack;
  typedef struct stack *stack;

  /* create a new, empty stack */
  extern stack stack_create(void);

  /* push a new element on top of the stack */
  extern void stack_push(stack s, void *element);

  /* pop the top element from the stack.  The stack must not be
     empty. */
  extern void stack_pop(stack s);

  /* return the top element of the stack */
  extern void *stack_top(stack s);
  /* return a true value if and only if the stack is empty */
  extern int stack_empty(stack s);

  #endif
```

We have added comments describing briefly what the module does, and describing what each interface function does. The phrases *must be* and *must not be* mean that a program that does not respect what these phrases say, is a program with errors in it. This module is therefore free to do whatever it thinks reasonable (including nothing at all) when such a condition is violated. In our implementation, we will call `assert` and abort the execution of the program.

For the implementation, we use a header object:

```
  #include "stack.h"
  #include "list.h"
  #include < assert.h>
  #include < stdlib.h>

  typedef struct stack *stack;

  struct stack
  {
    list elements;
  };

  stack  stack_create(void)
  {
    stack temp = malloc(sizeof(struct stack));
    temp -> elements = NULL;
    return temp;
  }

  void stack_push(stack s, void *element)
  {
    s -> elements = cons(element, s -> elements);
  }

int  stack_empty(stack s)
  {
    return s -> elements == NULL;
  }

  void stack_pop(stack s)
  {
    assert(!empty(s));
    s -> elements = cdr_and_free(s -> elements);
  }

  void * stack_top(stack s)
  {
    assert(!empty(s));
    return s -> elements -> element;
  }
```

12.6 Let us sum up

In this lesson, we briefly discussed about the concept of stack, various representation of stack, operations on stack, implementation of stack using array and implementation of stack using linked list. This lesson will help programmers to understand about stack data structure.

12.7 Points for discussion

1. How to implement push operation?
2. How to implement pop operation?
3. What is meant by stack overflow and underflow?
4. What will be the condition for checking empty stack?

12.8 Check your progress

1. Check empty stack

When *top* field reaches -1 then we can say the stack is empty. Here's a simple implementation of the function...

```
int StackIsEmpty(stackT *stackP)
{
  return stackP->top < 0;
}
```

2. Check the stack full

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 StackIsFull(stackT *stackP)
{
  return stackP->top >= stackP->maxSize - 1;
}
```

12.9 Lesson-end activities

1. What are all the way we can implement stack?

2. Find out one suitable real time example for stack?

3. Remember any two operations on stack.

12.10 References

Ellis HoroWitz and Sartaj Sahni: Fundamentals of Data structure, Galgotia book source.
Ashok N Kamthane, Programming and Data structures, Pearson Education
M.Tanenbaum, Data structure using C, PHI pub.
http://sysprog.net/
http://www.mycplus.com/cplus
http://www.programmersheaven.com/download/
http://en.literateprograms.org/
http://www.cs.utk.edu/~plank/plank/classes/
http://www.go4expert.com/forums/
http://www.cs.bu.edu/teaching/c/

LESSON - 13 : LINKED LIST

CONTENTS

13.0 Aims and Objectives

This lesson will help readers to understand about the concept of linked list adding and deleting an element with linked list. This lesson also discuss about predecessor and successor problems and its implementation.

13.1 Introduction

The Linked List is stored as a sequence of linked nodes. As in the case of the stack, each node in a linked list contains data AND a reference to the next node. The Linked List has the following properties:

• The list can grow and shrink as needed.

• The position of each element is given by an index from 0 to *n*-1, where n is the number of elements.

• Given any index, the time taken to access an element with that index depends on the index. This is because each element of the list must be traversed until the required index is found.

• The time taken to add an element at any point in the list does not depend on the size of the list, as no shifts are required. It does, however, depend on the index.Additions near the end of the list take longer than additions near the middle orstart. The same applies to the time taken to remove an element.

A list needs a reference to the front node

The following diagram describes the storage of a linked list called *LinkedList.* Each node consists of data (*DataItem*) and a reference (*NextNode*).

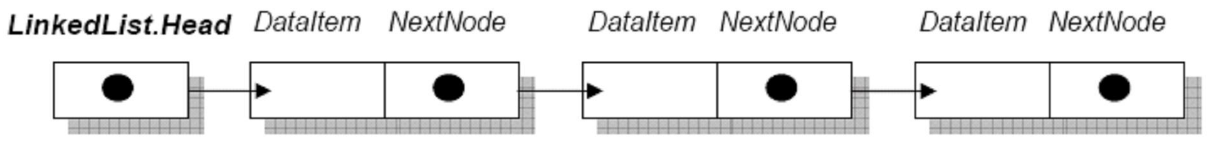


Figure 13.1.1 Linked list

• The first node is accessed using the name ***LinkedList.Head***
• Its data is accessed using ***LinkedList.Head.DataItem***
• The second node is accessed using ***LinkedList.Head.NextNode***

13.2 Types of linked lists

13.2.1 Singly-linked list

The simplest kind of linked list is a **singly-linked list** (or **slist** for short), which has one link per node. This link points to the next node in the list, or to a null value or empty list if it is the final node.

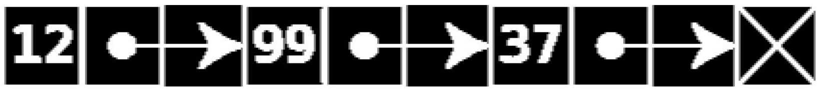


Figure 13.2.1 A singly-linked list containing three integer values

13.2.2 Doubly-linked list

A more sophisticated kind of linked list is a **doubly-linked list** or **two-way linked list**. Each node has two links: one points to the previous node, or points to a null value or empty list if it is the first node; and one points to the next, or points to a null value or empty list if it is the final node.

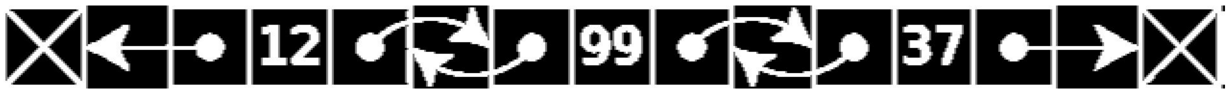


Figure 13.2.2 *A doubly-linked list containing three integer values*

In some very low level languages, Xor-linking offers a way to implement doubly-linked lists using a single word for both links, although the use of this technique is usually discouraged.

13.2.3 Circularly-linked list

In a **circularly-linked list**, the first and final nodes are linked together. This can be done for both singly and doubly linked lists. To traverse a circular linked list, you begin at any node and follow the list in either direction until you return to the original node. Viewed another way, circularly-linked lists can be seen as having no beginning or end. This type of list is most useful for managing buffers for data ingest, and in cases where you have one object in a list and wish to see all other objects in the list.

The pointer pointing to the whole list may be called the access pointer.

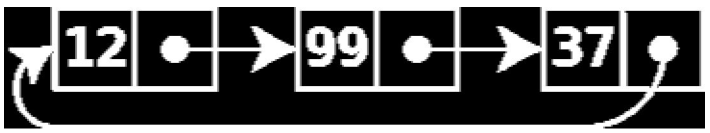


Figure 13.2.3 *A circularly-linked list containing three integer values*

13.3 Creating and manipulating a list

13.3.1 Nodes

Each node in the list contains a data pointer, and a pointer to the next element. The data pointer is of type `void *`, so that it could point to any data. A node with `next==NULL` is the last node in the list.

Any node can be viewed as representing the list beginning at that element, so we do not need a special structure to represent the whole list. The structure for a single node is:

```
typedef struct node_s {
        void *data;
        struct node_s *next;
} NODE;
```

13.3.2 Creating a node

Creating a new node is simple. The memory needed to store the node is allocated, and the pointers are set up. This function leaves allocation of data to the user.

```
NODE *list_create(void *data)
{
        NODE *node;
        if(!(node=malloc(sizeof(NODE)))) return NULL;
        node->data=data;
        node->next=NULL;
        return node;
}
```

13.3.3 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 insert a new node after an existing node.

```
NODE *list_insert_after(NODE *node, void *data)
{
        NODE *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:

```
NODE *list_insert_beginning(NODE *list, void *data)
{
        NODE *newnode;
        newnode=list_create(data);
        newnode->next = list;
        return newnode;
}
```

13.3.4 Removing a node

Note: This will not free the data associated with the node. *Is this able to remove the head node?*

```
int list_remove(NODE *list, NODE *node)
{
        while(list->next && list->next!=node) list=list->next;
        if(list->next) {
                list->next=node->next;
                free(node);
                return 0;
        } else return -1;
}
```

13.4 More Operating on the entire list

13.4.1 Traversing a list

The user-supplied function pointer will be called once for each element in the list.

```
int list_foreach(NODE *node, int(*func)(void*))
{
        while(node) {
                if(func(node->data)!=0) return -1;
                node=node->next;
        }
        return 0;
}
```

13.4.2 Searching a list

This function will return the first node to which the supplied function pointer returns a positive number.

```
NODE *list_find(NODE *node, int(*func)(void*,void*), void *data)
{
        while(node) {
                if(func(node->data, data)>0) return node;
                node=node->next;
        }
        return NULL;
}
```

Examples of user-supplied functions

Here are a few examples of user-supplied functions that might be used in list traversals and searches. The first function prints a value to `stdout`, and can be used with a traversal to print the entire contents of a list.

```
int printstring(void *s)
{
        printf("%s\n", (char *)s);
        return 0;
}
```

The second function tests to see if the value of a given node matches some string.

```
int findstring(void *listdata, void *searchdata)
{
        return strcmp((char*)listdata, (char*)searchdata)?0:1;
}
```

Complete program will be as follows,

Example 13.1

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
int main()
{
        NODE *list, *second, *inserted;
        NODE *match;

        /* Create initial elements of list */
        list=list_create((void*)"First");
        second=list_insert_after(list, (void*)"Second");
        list_insert_after(second, (void*)"Third");

        printf("Initial list:\n");
        list_foreach(list, printstring);
        putchar('\n');

        /* Insert 1 extra element in front */
        list=list_insert_beginning(list, "BeforeFirst");
        printf("After list_insert_beginning():\n");
        list_foreach(list, printstring);
        putchar('\n');

        /* Insert 1 extra element after second */
        inserted=list_insert_after(second, "AfterSecond");
        printf("After list_insert_after():\n");
        list_foreach(list, printstring);
        putchar('\n');
        /* Remove the element */
        list_remove(list, inserted);
        printf("After list_remove():\n");
        list_foreach(list, printstring);
        putchar('\n');

        /* Search */
        if((match=list_find(list, findstring, "Third")))
                printf("Found \"Third\"\n");
        else printf("Did not find \"Third\"\n");

        return 0;
}
```

The following figures shows diagrammatic representation of adding and deleting a node in the linked list.

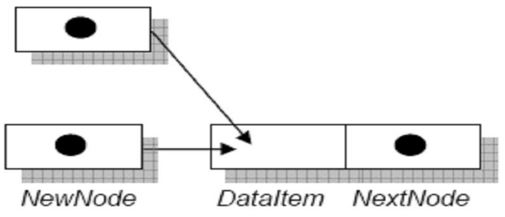


Figure 13.4.1 The first node is set to be the new node if list is empty

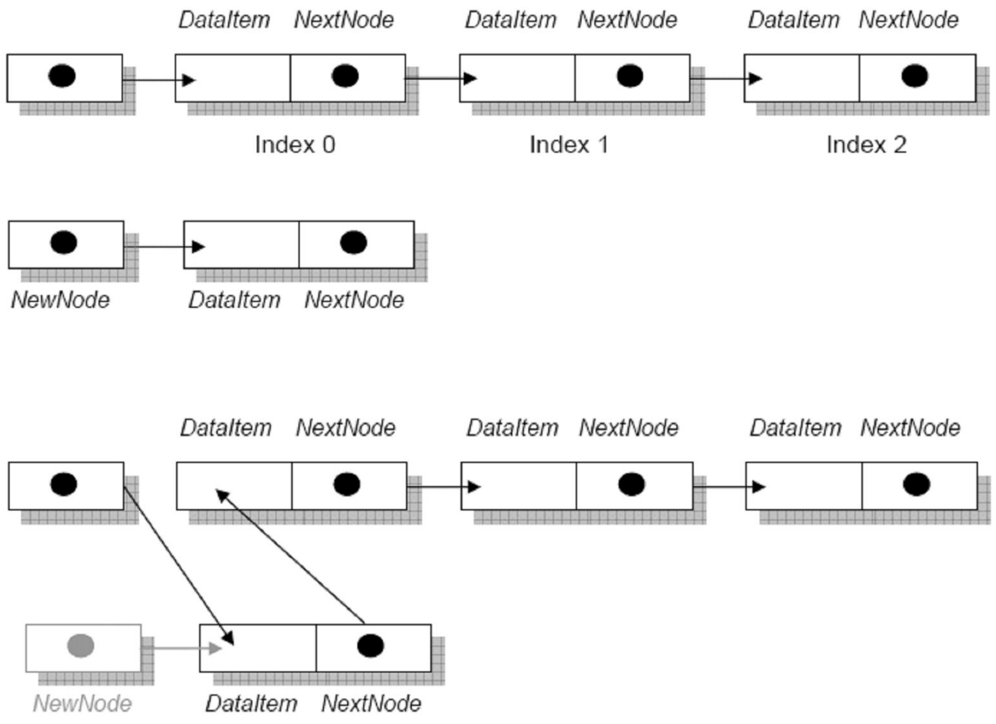


Figure 13.4.2 Node is added at the beginning

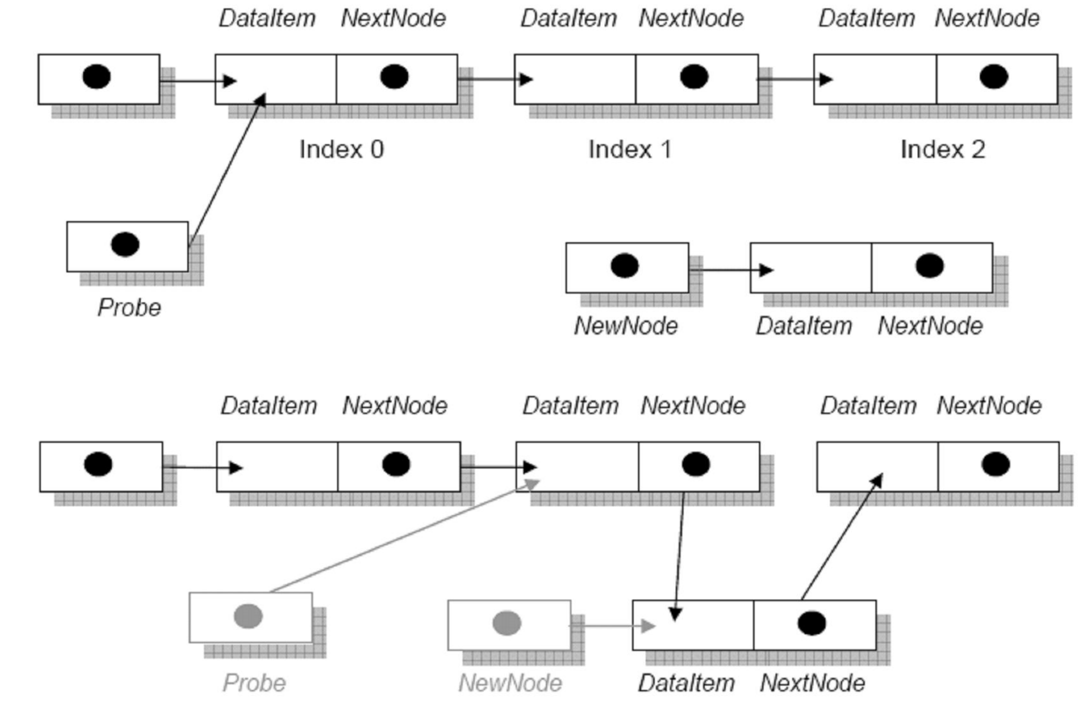
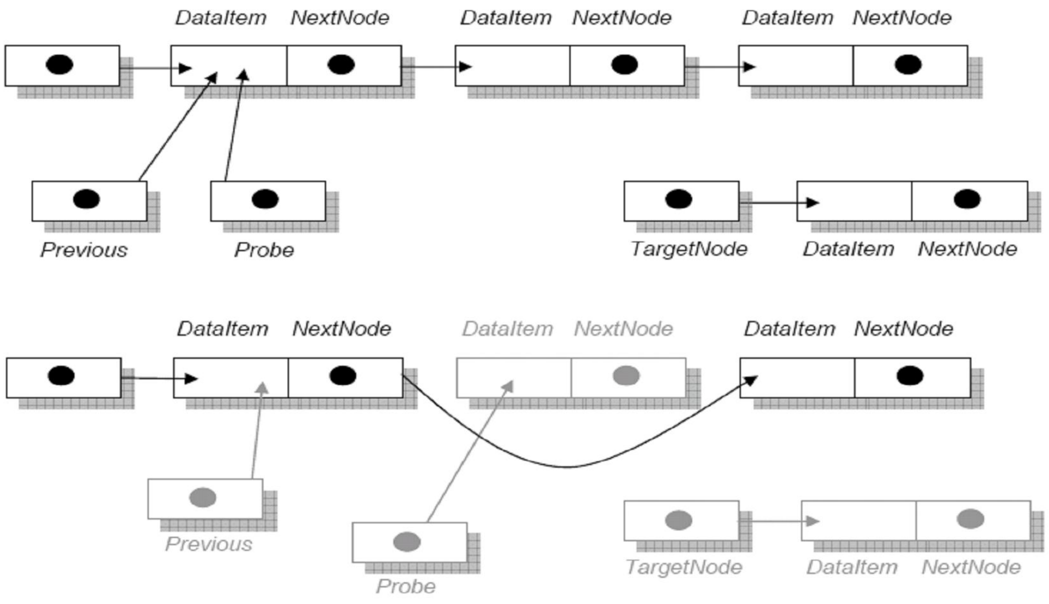


Figure 13.4.3 Node is added at the end of list. Prob is a pointer to identify the location of insertion. Figure 13.4.4 Deleting a node



13.5 Doubly linked lists

Doubly linked lists are like singly linked lists, except each node has two pointers -- one to the next node, and one to the previous node. This makes life nice in many ways:

- You can traverse lists forward and backward.
- You can insert anywhere in a list easily. This includes inserting before a node, after a node, at the front of the list, and at the end of the list.
- You can delete nodes very easily.

The following program shows various operation involved in doubly linked list
Figure 13.2

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
struct node
{
char info;
struct node *next,*prev;

} *head=NULL,*newnode ,*Traverse;

insertdata(char chh1)
{


if (head == NULL)
{
newnode=(struct node *)malloc (sizeof(struct node));
newnode->info=chh1;
newnode->next=newnode->prev=NULL;
//newnode->next=head;
//head->prev=newnode;
head=newnode ;

}


else
{
newnode= (struct node *)malloc (sizeof(struct node));
newnode->info=chh1;
newnode->next=newnode->prev=NULL;
Traverse=head;
while(Traverse->next !=NULL)
Traverse=Traverse->next;

Traverse->next=newnode;
newnode->prev=Traverse;

}

return ;
}
```

```
deletedata (char chh1)
{
Traverse=head;
if (Traverse->info==chh1)
{

head=head->next;

free(Traverse);
}
else
{
while(Traverse->info!=chh1)
Traverse=Traverse->next;

Traverse->next->prev=Traverse->prev;
Traverse->prev->next=Traverse->next;
free(Traverse) ;
//printf("\nRecord deleted for %c ",chh1);
}
return ;
}

showdata()
{
Traverse=head;
while (Traverse->info !=NULL)
{
printf("%c",Traverse->info);
Traverse=Traverse->next;
}

return ;
}
void main(void)
{
char chh;
char i;
clrscr();
while(1)
{

printf("\n1.insert 2.Delete ");
printf("3.show 4.exit \n ");
printf("\n\nEnter any one of the choice :--");
scanf("\n%c",&i);
switch(i)
{
case '1' :
printf("\nEnter a CHAR--");

scanf ("\n%c",&chh);
//scanf ("%c",&chh);
insertdata(chh);

break;

case '2' :

printf("\nEnter the CHAR U want to delete :: ");
scanf ("\n%c", &chh);
//scanf ("%c", &chh);
deletedata(chh);
break;

case '3' :

showdata();
break;

case '4' :
exit(0);
break ;
default :
printf("\n invalid choice\n" );
break;
}
}
}
```

Example 13.3

This program illustrate implementation of doubly linked list for mark statement preparation

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
#define N 100
```

```
struct dlinklist
{
    struct dlinklist *prev;  /** Stores address of previous node **/
    int roll_no;             /** stores roll number **/
    char name[N];            /** stores Name **/
    float marks;             /** stores Marks **/
    struct dlinklist *next;  /** stores address of next node **/
};

/** Redefining dlinklist as node **/
typedef struct dlinklist node;

void init(node*);     /** Input function **/
void ins_aft(node*);  /** Function inserting before **/
node* ins_bef(node*); /** Function inserting after **/
node* del(node*);     /** Function deleting a node **/
void search(node*);   /** Function for searching node **/
void disp(node*);     /** Function for displaying node **/
void rollsrch(node*); /** Function for searching node by roll number
**/
void namesrch(node*); /** Function for searching node by name **/
void marksrch(node*); /** Function for searching node by marks **/



void main()
{
    node *head;
    char ch;                        /* Choice inputing varible */
    int opt;                        /* Option inputing variable*/
    static int flag=0;              /* Unchanged after iniialization
*/
    clrscr();
    head=(node*)malloc(sizeof(node));
    head->next=NULL;
    head->prev=NULL;
    do
    {
again:
    printf("\nEnter your option\n");
    printf("\n1. Initialize the node\n");
    printf("\n2. Insert before a specified node\n");
    printf("\n3. Insert after a specified node\n");
    printf("\n4. Delete a particular node\n");
    printf("\n5. Search the nodes\n");
    printf("\n6. Display all the nodes\n");
    scanf("%d",&opt);
    if(flag==0 && opt!=1)
    {
        printf("\nNo. You must first initialize at least one node\n");

        goto again;
    }
    if(flag==1 && opt==1)
    {
        printf("\nInitialisation can occur only once.\n");
        printf("\nNow you can insert a node\n");
        goto again;
    }
    if(opt==4 && head->next==NULL)
    {
        printf("\nYou cannot delete the one and only the single
node\n");
        goto again;
    }
    if(flag==0 && opt==1)
        flag=1;
    switch(opt)
    {
    case 1:
        init(head);
        break;
    case 2:
        head=ins_bef(head);
        break;
    case 3:
        ins_aft(head);
        break;
    case 4:
        head=del(head);
        break;
    case 5:
        search(head);
        break;
    case 6:
        disp(head);
        break;
    }
    printf("\nDo you wish to continue[y/n]\n");
    ch=getche();
    }while(ch=='Y' || ch=='y');
    printf("\nDone by \"SHABBIR\"\n");
    printf("\nPress any key to exit\n");
    getch();
}

void init(node *current)
{
    current->prev=NULL;
    printf("\nEnter Roll number\n");
    scanf("%d",&current->roll_no);
    printf("\nEnter the name\n");
    fflush(stdin);
    gets(current->name);
    printf("\nEnter the marks\n");
    scanf("%f",&current->marks);
    current->next=NULL;
}

void ins_aft(node *current)
{
    int rno;             /* Roll number for inserting a node*/
    int flag=0;
    node *newnode;
    newnode=(node*)malloc(sizeof(node));
    printf("\nEnter the roll number after which you want to insert a
node\n");
    scanf("%d",&rno);
    init(newnode);
    while(current->next!=NULL)
    {
        /***  Insertion checking for all nodes except last  ***/
        if(current->roll_no==rno)
        {
            newnode->next=current->next;
            current->next->prev=newnode;
            current->next=newnode;
            newnode->prev=current;
            flag=1;
        }
        current=current->next;
    }
    if(flag==0 && current->next==NULL && current->roll_no==rno)
    {
        /***  Insertion checking for last nodes  ***/
        newnode->next=current->next;
        current->next=newnode;
        flag=1;
    }
    else if(flag==0 && current->next==NULL)
        printf("\nNo match found\n");
}

node* ins_bef(node *current)
{
    int rno;            /* Roll number for inserting a node*/
    node *newnode,*temp;
    newnode=(node*)malloc(sizeof(node));
    printf("\nEnter the roll number before which you want to insert a
node\n");
    scanf("%d",&rno);
    init(newnode);
    if(current->roll_no==rno)
    {
        /*** Insertion checking for first node ***/
        newnode->next=current;
        current->prev=newnode;
        current=newnode;
        return(current);
    }
    temp=current;
    while(temp->next!=NULL)
    {
        /*** Insertion checking for all node except first ***/
        if(temp->next->roll_no==rno)
        {
            newnode->next=temp->next;
            temp->next->prev=newnode;
            temp->next=newnode;
            newnode->prev=temp;
            return(current);
        }
        temp=temp->next;
    }
    /*
    If the function does not return from any return statement.
    There is no match to insert before the input  roll number.
    */
    printf("\nMatch not found\n");
    return(current);
}

node* del(node *current)
{
    int rno;                /* Roll number for deleting a node*/
    node *newnode,*temp;
    printf("\nEnter the roll number whose node you want to delete\n");
    scanf("%d",&rno);
    newnode=current;
    if(current->roll_no==rno)
    {
        /***  Checking condition for deletion of first node  ***/
        newnode=current; /*  Unnecessary step  */
        current=current->next;
        current->prev=NULL;
        free(newnode);
        return(current);
    }
    else
    {
        while(newnode->next->next!=NULL)
        {
            /***  Checking condition for deletion of   ***/
            /*** all nodes except first and last node  ***/
            if(current->next->roll_no==rno)
            {
                newnode=current;
                temp=current->next;
                newnode->next=newnode->next->next;
                newnode->next->prev=current;
                free(temp);
                return(current);
            }
            newnode=newnode->next;
        }
        if(newnode->next->next==NULL && newnode->next->roll_no==rno)
        {
            /***  Checking condition for deletion of last node  ***/
            temp=newnode->next;
            free(temp);
            newnode->next=NULL;
            return(current);
        }
    }
    printf("\nMatch not found\n");
    return(current);
}

void search(node *current)
{
    int ch;                  /* Choice inputing variable */
    printf("\nEnter the criteria for search\n");
    printf("\n1. Roll number\n");
    printf("\n2. Name\n");
    printf("\n3. Marks\n");
    scanf("%d",&ch);
    switch(ch)
    {
    case 1:
        rollsrch(current);
        break;
    case 2:
        namesrch(current);
        break;
    case 3:
        marksrch(current);
        break;
    default:
        rollsrch(current);
    }
}

void rollsrch(node *current)
{
    int rno;
    printf("\nEnter the roll number to search\n");
    scanf("%d",&rno);
    while(current->next!=NULL)
    {
        if(current->roll_no==rno)
            printf("\n%d\t%s\t%f\n",current->roll_no,current-
>name,current->marks);
        current=current->next;
    }
    if(current->next==NULL && current->roll_no==rno)
        printf("\n%d\t%s\t%f\n",current->roll_no,current->name,current-
>marks);
}


void namesrch(node *current)
{
    char arr[20];
```

```
    printf("\nEnter the name to search\n");
    fflush(stdin);
    gets(arr);
    while(current->next!=NULL)
    {
        if(strcmp(current->name,arr)==NULL)
```

```
            printf("\n%d\t%s\t%f\n",current->roll_no,current-
>name,current->marks);
        current=current->next;
    }
    if(current->next==NULL && strcmp(current->name,arr)==NULL)
        printf("\n%d\t%s\t%f\n",current->roll_no,current->name,current-
>marks);
}

void marksrch(node *current)
{
    float marks;
    printf("\nEnter the marks to search\n");
    scanf("%f",&marks);
    while(current->next!=NULL)
    {
        if(current->marks==marks)
            printf("\n%d\t%s\t%f\n",current->roll_no,current-
>name,current->marks);
        current=current->next;
    }
    if(current->next==NULL && current->marks==marks)
```

```
printf("\n%d\t%s\t%f\n",current->roll_no,
                   current->name,current->marks);
```

```
}

void disp(node *current)
{
    while(current!=NULL)
    {
        printf("\n%d\t%s\t%f",current->roll_no,current->name,current-
>marks);
        current=current->next;
    }
}
```

13.6 Let us sum up

In the lesson, we discussed about linked list and its classifications, difference between various forms of linked list, implementation of single linked list and algorithamic representation, the concept of doubly linked list and its application.we also discussed about the way of implementing doubly linked list.

13.7 Points for discussion

1. How to implement single linked list?
2. What is the speciality of circular linked list?
3. What is meant by node?
4. Define doubly linked list

13.8 Check your progress

1. Define the term circular queu

In a **circularly-linked list**, the first and final nodes are linked together. This can be done for both singly and doubly linked lists. To traverse a circular linked list, you begin at any node and follow the list in either direction until you return to the original node. Viewed another way, circularly-linked lists can be seen as having no beginning or end.

2. How to define structure for a node

```
typedef struct node_s
 {
       void *data;
       struct node_s *next;
 } NODE;
```

13.9 Lesson-end activities

1. How many classification of linked lists are available?
2. Why we need the support of linked list?
3. what do you meant by node?

13.10 References

Ellis HoroWitz and Sartaj Sahni: Fundamentals of Data structure, Galgotia book source.
Ashok N Kamthane, Programming and Data structures, Pearson Education
M.Tanenbaum, Data structure using C, PHI pub.
http://en.literateprograms.org/
http://www.cs.utk.edu/~plank/plank/classes/
http://www.go4expert.com/forums/
http://www.cs.bu.edu/teaching/c/

LESSON - 14 : QUEUE

CONTENTS

14.0 Aims and Objectives

The objective of this lesson is to make readers to know about the concept of Queue. This lesson will helps the readers to implement various applications using the concept of queue.

14.1 Introduction

Queue is a linear data structure in which data can be added to one end and retrieved from the other. Just like the queue of the real world, the data that goes first into the queue is the first one to be retrieved. That is why queues are sometimes called as **First-In-First-Out** data structure.
In case of queues, we saw that data is inserted both from one end but in case of Queues; data is added to one end (known as REAR) and retrieved from the other end (known as FRONT).

The data first added is the first one to be retrieved while in case of queues the data last added is the first one to be retrieved.

14.2 A few points regarding Queues:

1. **Queues:** It is a linear data structure; linked lists and arrays can represent it. Although representing queues with arrays have its shortcomings but due to simplicity, we will be representing queues with arrays in this article.
2. **Rear:** A variable stores the index number in the array at which the new data will be added (in the queue).
3. **Front:** It is a variable storing the index number in the array where the data will be retrieved.

Let us have look at the process of adding and retrieving data in the queue with the help of an example.

Consider a queue containing four elements A,B,C,D

```
Front     A        B           C           D    Rear
```

In the above queue, queue, A is at the front of the queue and D is at the rear.

After deleting an element

Front B C D Rear

Since elements may be deleted only from the front of the queue, A is removed. Now B is at the front.

After Adding an element

Front B C D E Rear

Since elements may be inserted only from the rear the of the queue, E s added at the rear end. Now E is at the rear.

14.3 Operations on queue

The operations available on a queue are insert(), remove() and empty().

Assume a queue q and an item x. Then the operation insert(q,x) inserts an item x at the rear end of the queue q. The operation remove (q) removes the front element from the queue q and returns it. Empty(q) operation returns true if the queue is empty; otherwise it returns false.

14.4 C implementation of queues

14.4.1 Representation

A queue is represented by an array. The array is used to hold the elaments. Also two variables are used to implement the queue. They are (i) front and (ii) rear; variable "front" is used to store the position of the first element of the queue and the variable "rear" is used to store the position of the last element of the queue.

A queue of integers may be declared and initialized as

```
# define MAXSIZE 100
struct queue
int items[MAXSIZE];
 int front;
int rear;
};
```

```
main()
{
    struct queue q ;
    q.front MAXSIZE;
     q.rear MAXSIZE ;

}
```

Assume the array which hold the queue as a circle; ie., assume that first element of array immediately follows its last element. In this case if front=rear then queue is empty.

Note that q.front and .rear are initialized to the last index of the array[last index is MAXSIZE]

14.4.2 Empty Operation

The coding for empty function is as follows

```
empty(struct queue 9)
{
int flag = 0;
if (q.front == q.rear) /* queue is empty */
 return (1);
else
return (0);
}
```

If the above function is available, an empty queue is tested by the statement.

```
        If (empty(q))
        printf ("queue is empty");
        else
                printf("queue is not empty");
```

14.4.3 Remove operation

Remove operation is possible in the queue if it is not empty.

```
remove (struct queue q, int uf, int x)
        uf = 0;
if ((empty(q)) == 1)
{
printf ("queue is underfloor \n");
uf = 1;
                }
else
{
```

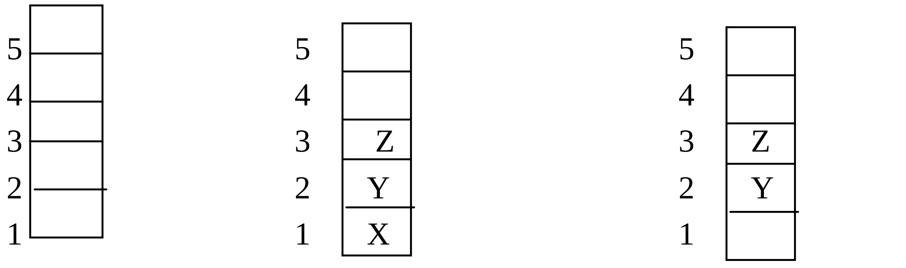
```
if (q.front == MAXSIZE)
   q.front = 1;
else
q.front = q.front + 1
x = q.item (q.front];
return (x);
                  }
         }
```

14.4.4 Insert Operation

In case of insert operation, overflow condition is to be considered. It is assumed that the queue can grow only as large as the size of and array. The function for insert procedure is as follows,

```
insert(struct queue q, int ovf, int x)
{
         ovf=0;
         if(q.rear==MAXSIZE)
         q.rear=1;
         else
         q.rear=q.rear+1;
         if(q.rear==q.front)
         {
                  printf("Queue overflow");
                  ovf = 1;
                  q.rear=q.rear + 1;
         }
         else
                  q.items[q.rear]=x;
}
```

Example



Queue empty After insertion(X,Y,Z) After deletion(X)

14.5 Priority queue

In both stack and queue, the items are orderly placed based on the sequence in which they have been inserted. If there is any intrinsic order (numeric order or alphabetic order) among the elements stored in the stack or queue, these order is ignored in the stack or queue operation. In stacks and queues, it is possible to retrieve the element at one end of the list.

The priority queue is also a data structure. In priority queue, the intrinsic ordering of the elements determines the result of its basic operation. In the priority queue, it is possible to retrieve the smallest element or the largest element, if they are placed at any position.

Two types of Priority queues are available. They are (i) ascending priority queue and (ii) descending priority queue.

14.5.1 Ascending priority queue

An ascending priority queue, items are inserted arbitrarily and only the smallest item can be removed. If apq is an ascending priority queue, the operation pqinsert(apq, x) inserts an item x into apq and pqmin delete(apq) removes the smallest element from apq and returns it value. Applying the function pqmindelete() successively, retrieves the elements in ascending order.

14.5.2 Descending priority queue

In descending priority queue, items are inserted arbitrarily and allows the removal of only the largest element. If dpq is a descending priority queue, then the operatin pqinsert(dpq, x) inserts an item x into dpq and the operation pqmaxdelete (dpq) removes the largest element from dpq and returns it value. Applying the functions pqmaxdelete continuously retrieves the element in descending order.

The operation empty(pq) is used to determine whether a priority queue is empty or not. The elements of the priority queue can be numbers or characters or complex structures that are ordered on one or several fields.

14.6 Array Implementation of a priority queue

Assume the n elements of a priority queue ^{p}q are placed in position 1 to n of an array of pq.items. Assume the size of the queue is MAXSIZEPQ. Then pq.rear equals the first empty array position. Then the coding for pqinsert(pq, x) is very simple and as follows.

```
if (pq.rear >= MAXSIZEPQ)
```

```
{
printf ("priority queue overflow");
exit (1);
          }
pq. items[pq. rear] = x;
pq. rear = pq. rear + 1;
```

In the above method, the elements of the priority queue are not kept ordered in the array.

Deleting of items from the priority queue involves the following two steps.

(i) In the case of ascending priority queue, locate the smaller element first. For doing this, every element of the array from pq.items[1] through pq. items[pq. rear] must be examined. Therefore deletion requires accessing every elements of the priority queue

(ii). Removal of the above element from the array.

Deletion problem is solved in the following four ways

1. A special "empty" symbol can be placed into the position of the deleted element. Care must be taken to select this symbol. For example, if the priority queue of non-negative numbers, choose indicator as –1. *Disadvantage of this method is as follows:* Each search process examines these deleted positions in addition to the actual elements. To delete number of items, the deletion operation accesses many more array elements than exist in the priority queue

2. The insertion operation is modified to insert a new item'in the first "empty" position. Now the insertion operation will become complex. So the insertion efficiency is decreased. This is a major draw hack of this method.

3. After each deletion operation, shift all the elements beyond the deleted element by one position and then decrementing pq.rear by 1. In this method, for each deletion, or on the average, half of all priority queue elements are shifted. So deletion becomes quite inefficient.

4. Maintain the priority queue as an ordered, circular array i.e., the elements are stored in ascending order for ascending priority queue.

```
# defies MAXSIZE PQ 50
struct pqueue
{
int items[MAXS!ZE PQ];
int front, rear;
}q;
```

q.front is the position of the smallest element. q.rear is 1 greater than the position of the largest. In the ascending priority queue, to delete an item, decrease the value of q.front and return the above value.

In this method, insertion of an element will take many steps. When inserting an element into the priority queue, the correct location to be found. Then the new element is inserted after shifting all the elements from that location by one position.

14.7. Let us sum up

In this lesson, we described about queue and its representation, implementation of queue using array, various operations on queue and its implementation, priority queues, importance of priority queues.

14.8 Points for discussion

1. Define the term queue
2. How to check that queue is empty?
3. what are all the ways to implement queue?

14.9 Check your progress

1. Explain various operations on queue.

Assume a queue q and an item x. Then the operation insert(q,x) inserts an item x at the rear end of the queue q. The operation remove (q) removes the front element from the queue q and returns it. Empty(q) operation returns true if the queue is empty; otherwise it returns false.

2. what is meant by ascending priority queue?

An ascending priority queue, items are inserted arbitrarily and only the smallest item can be removed. If apq is an ascending priority queue, the operation pqinsert(apq, x) inserts an item x into apq and pqmin delete(apq) removes the smallest element from apq and returns it value. Applying the function pqmindelete() successively, retrieves the elements in ascending order.

14.10 Lesson end activities

1. What is the difference between queue and stack?
2. Why we need queue?
3. What are all the operations on queue?
4. How to construct a queue?

14.11 References

Ellis HoroWitz and Sartaj Sahni: Fundamentals of Data structure, Galgotia book source.
Ashok N Kamthane, Programming and Data structures, Pearson Education
M.Tanenbaum, Data structure using C, PHI pub.
http://www.mycplus.com/cplus
http://www.programmersheaven.com/download/
http://en.literateprograms.org/
http://www.cs.utk.edu/~plank/plank/classes/

UNIT -V : SEARCHING TECHNIQUES

LESSON - 15

CONTENTS

15.0 Aims and objectives

the objective of this lesson is to make people to understand about importance of various searching techniques and its implementation. This lesson will also helps reader to know about two common searching techniques, such as, linear and binary searching.

15.1 Introduction : Searching

The task of searching is one of most frequent operations in computer programming. It also provides an ideal ground for application of the data structures so far encountered. There exist several basic variations of the theme of searching, and many different algorithms have been developed on this subject. The basic 34 assumption in the following presentations is that the collection of data, among which a given element is to be searched, is fixed. We shall assume that this set of N elements is represented as an array, say as

a: ARRAY N OF Item

Typically, the type item has a record structure with a field that acts as a key. The task then consists of finding an element of a key field is equal to a given search argument x. The resulting index i, satisfying a[i].key = x, then permits access to the other fields of the located element. Since we are here interested in the task of searching only, and do not care about the data for which the element was searched in the first place, we shall assume that the type *Item* consists of the key only, i.e. *is* the key.

15. 2 Linear Search

15.2.1 Definition

The linear search is a search algorithm, also known as **sequential search**, that is suitable for searching a set of data for a particular value. It operates by checking every element of a list one at a time in sequence until a match is found. Linear search runs in O(N). If the data are distributed randomly, on average (N+1)/2 comparisons will be needed. The best case is that the value is equal to the first element tested, in which case only 1 comparison is needed. The worst case is that the value is not in the list (or is the last item in the list), in which case N comparisons are needed.

15.2.2 Characteristics of linear search

The simplicity of the linear search means that if just a few elements are to be searched it is less trouble than more complex methods that require preparation such as sorting the list to be searched or more complex data structures, especially when entries may be subject to frequent revision. Another possibility is when certain values are much more likely to be searched for than others and it can be arranged that such values will be amongst the first considered in the list.

15.2.3 Algorithm

The following pseudocode describes the linear search technique.

```
For each item in the list:

  Check to see if the item you're looking for matches the item in the list.

        If it matches.
              Return the location where you found it (the index).
        If it does not match.
              Continue searching until you reach the end of the list.

If we get here, we know the item does not exist in the list. Return -1.
```

In computer implementations, it is usual to search the list in order, from element 1 to N (or 0 to N - 1, if array indexing starts with zero instead of one) but a slight gain is possible by the reverse order. Suppose an array *A* having elements 1 to N is to be searched for a value *x* and if it is not found, the result is to be zero.

```
for i:=N:1:-1 do             %Search from N down to 1. (The step is -1)
 if A[i] = x then QuitLoop i;
next i;
Return(i);          %Or otherwise employ the value.
```

Implementations of the loop must compare the index value *i* to the final value to decide whether to continue or terminate the loop. If this final value is some variable such *N* then a subtraction *(i - N)* must be done each time, but in going down from *N* the loop termination condition is for a constant, and moreover a special constant. In this case, zero. Most computer hardware allows the sign to be tested, especially the sign of a value in a register, and so execution would be faster. In the case where the loop was for arrays indexed from zero, the loop would be *for i:=N - 1:0:-1 do* and the test on the index variable would be for it negative, not zero.

The pseudocode as written relies on the value of the index variable being available when the for-loop's iteration is exhausted, as being the value it had when the loop condition failed, or a 'QuitLoop' was executed. Some compilers take the position that on exit from a for-loop no such value is defined, in which case it would be necessary to copy the index variable's value to a reporting variable before exiting the loop, or to use another control structure such as a *while* loop, or else explicit code with *go to* statements in pursuit of the fastest-possible execution.

15.2.4 Implementation

The following code example for the “C” programming language is a simple implementation of a linear search.

```
 int linearSearch(int a[], int valueToFind)
{
   //a[] is an array of integers to search.
   //valueToFind is the number that will be found.
   //The function returns the position of the value if found.
   //The function returns -1 if valueToFind was not found.
   for (int i=0; i<length; i++) {
     if (valueToFind == a[i]) {
       return i;
     }
   }
   return -1;
 }
```

The List module in the OCaml standard library defines a function called "mem" that returns true if the given element is in the given list or false if not. This function could be defined as:

```
let rec mem x = function
    [] -> false
  | h :: t -> h=x || mem x t
```

Mathematica's unusually powerful pattern matching allows linear search to be implemented by a pattern match:

```
Mem[x_, {___, x_, ___}] := True
Mem[_, _] := False
```

Linear search can be used to search an unordered list. The more efficient binary search can only be used to search an ordered list.

If more than a small number of searches are needed, it is advisable to use a more efficient data structure. One approach is to sort and then use binary searches. Another common one is to build up a hash table and then do hash lookups.

15.3 Binary Search

15.3.1 About Binary Search

The most common application of binary search is to find a specific value in a sorted list**.** To cast this in the frame of the guessing game (see Example below), realize that we are now guessing the *index*, or numbered place, of the value in the list. This is useful because, given the index, other data structures will contain associated information. Suppose a data structure containing the classic collection of name, address, telephone number and so forth has been accumulated, and an array is prepared containing the names, numbered from one to *N*. A query might be: what is the telephone number for a given name *X*. To answer this the array would be searched and the index (if any) corresponding to that name determined, whereupon it would be used to report the associated telephone number and so forth. Appropriate provision must be made for the name not being in the list (typically by returning an *index* value of zero), indeed the question of interest might be only whether *X* is in the list or not.

If the list of names is in sorted order, a binary search will find a given name with far fewer probes than the simple procedure of probing each name in the list, one after the other in a linear_search, and the procedure is much simpler than organising a hash table though that would be faster still, typically averaging just over one probe. This applies for a uniform distribution of search items but if it is known that some few items are *much* more likely to be sought for than the majority then a linear search with the list ordered so that the most popular items are first may do better.

The binary search begins by comparing the sought value *X* to the value in the middle of the list; because the values are sorted, it is clear whether the sought value would belong before or after that middle value, and the search then continues through the correct half in the same way. Only the sign of the difference is inspected: there is no attempt at an interpolation search based on the size of the differences.

15.3.2. Implementation of Binary search using Recursion

The most straightforward implementation is recursive, which recursively searches the subrange dictated by the comparison: In the following code, Multiple return statements is a bad programming style

```
  BinarySearch(A[0..N-1], value, low, high)
 {
      if (high < low)
          return not_found
      mid = (low + high) / 2
      if (A[mid] > value)
          return BinarySearch(A, value, low, mid-1)
      else if (A[mid] < value)
          return BinarySearch(A, value, mid+1, high)
      else
          return mid
  }
```

It is invoked with initial `low` and `high` values of `0` and `N-1`.

15.3.3. Implementation of Binary search without Recursion

We can eliminate the tail recursion above and convert this to an iterative implementation:

```
  BinarySearch(A[0..N-1], value) {
      low = 0
      high = N - 1
      while (low <= high)
```

```
        {
            mid = (low + high) / 2
            if (A[mid] > value)
                high = mid - 1
            else if (A[mid] < value)
                low = mid + 1
            else
                return mid
       }
       return not_found
   }
```

Some implementations may not include the early termination branch, preferring to check at the end if the value was found, shown below. Checking to see if the value was found *during* the search (as opposed to at the *end* of the search) may seem a good idea, but there are extra computations involved in each iteration of the search. Also, with an array of length *N* using the *low* and *high* indices, the probability of actually finding the value on the first iteration is 1 / *N*, and the probability of finding it later on (before the end) is the about 1 / (*high* - *low*). The following checks for the value at the end of the search:

```
       low = 0
       high = N
       while (low < high)
        {
           mid = (low + high)/2;
           if (A[mid] < value)
               low = mid + 1;
           else
                //can't be high = mid-1: here A[mid] >= value,
                //so high can't be < mid if A[mid] == value
                high = mid;
       }

       if (low < N) and (A[low] == value)
           return low
       else
           return not_found
```

This algorithm has two other advantages. At the end of the loop, *low* points to the first entry greater than or equal to *value*, so a new entry can be inserted if no match is found. Moreover, it only requires one comparison; which could be significant for complex keys in languages which do not allow the result of a comparison to be saved.

In practice, one frequently uses a three-way comparison instead of two comparisons per loop. Also, real implementations using fixed-width integers with modular arithmetic need to account for the possibility of overflow. One frequently-used technique for this is to compute mid, so that two smaller numbers are ultimately added:

```
           mid = low + ((high - low) / 2)
```

Equal elements

The elements of the list are not necessarily all unique. If one searches for a value that occurs multiple times in the list, the index returned will be of the first-encountered equal element, and this will not necessarily be that of the first, last, or middle element of the run of equal-key elements but will depend on the positions of the values. Modifying the list even in seemingly unrelated ways such as adding elements elsewhere in the list may change the result.

To find all equal elements an upward and downward linear search can be carried out from the initial result, stopping each search when the element is no longer equal. Thus, e.g. in a table of cities sorted by country, we can find all cities in a given country.

15.3.4 Sort key

A list of pairs (p,q) can be sorted based on just p. Then the comparisons in the algorithm need only consider the values of p, not those of q. For example, in a table of cities sorted on a column "country" we can find cities in Germany by comparing country names with "Germany", instead of comparing whole rows. Such partial content is called a sort key.

15.3.5 Correctness and testing

Binary search is one of the trickiest "simple" algorithms to program correctly. A study has shown that an astounding 91.379 percent of professional programmers fail to code a binary search correctly after a whole hour of working on it, and another study shows that accurate code for it is only found in five out of twenty textbooks. (Kruse, 1999) Given this insight, it is important to remember that the best way to verify the correctness of a binary search algorithm is to thoroughly test it on a computer. It is difficult to visually analyze the code without making a mistake.

To that end, the following code will thoroughly test a binary search at every index for many multiple lengths of arrays:

```
int offset, value, index, length;
bool passed=true;
for(offset=1; offset<5; offset++)
{       //tests with an offset between 1 and 2 for various amounts.
    for(length = 1; length < 2049; length++)
        { //make array longer on each iteration
                int A[length];
              for(int i = 0; i < length; i++)
                    //init array values from 0 to length-1
                        A[i] = i*offset;
                for(value = 0; value < length; value++)
                       {     //search for every array value
                        index = binarySearch(A, value*offset);
                        if (!(index==value))
                        passed=false;   //if this line executes, BUG in
                                              binary search
```

```
                              }
                }
}
```

In the above C test-code, if *passed* is ever false, then the binary search function has a bug. Note that this code assumes that you are returning index of search value with array; in addition it does not test for values not within the array, proper handling of duplicate values within your array, or errors that could be caused by more randomly distributed values. As such this should not be considered a complete proof of correctness, merely an aid for testing.

15.3.5 Performance

Binary search is a logarithmic algorithm and executes in O(log2(n)) time. Specifically, $1 + \log_2 N$ iterations are needed to return an answer. In most cases it is considerably faster than a linear search. It can be implemented using recursion or iteration, as shown above. In some languages it is more elegantly expressed recursively; however, in some C-based languages tail recursion is not eliminated and the recursive version requires more stack space.

Binary search can interact poorly with the memory hierarchy (i.e. caching), because of its random-access nature. For in-memory searching, if the interval to be searched is small, a linear search may have superior performance simply because it exhibits better locality of reference. For external searching, care must be taken or each of the first several probes will lead to a disk seek. A common technique is to abandon binary searching for linear searching as soon as the size of the remaining interval falls below a small value such as 8 or 16.

When multiple binary searches are to be performed with the same key in related lists, fractional cascading can be used to speed up successive searches after the first one.

15.4 Let us sum up

In this lesson, I am sure that, you would clearly understand about the concept of searching techniques, implementation importance and of searching techniques, the concept of linear search, advantages and disadvantages, difference between linear search and binary search, implementation with recursion and without recursion, complexities of algorithm and importance of binary search.

15.5 Points for discussion

1. Why and where we need searching techniques?

2. What are all the advantages and disadvantages of linear search?

3. Define the term binary search.

4. Differentiate linear search and binary search

15.6 Check your progress

1. Define the term binary search.

The binary search is one of the common searching technique, where it begins by comparing the sought value *X* to the value in the middle of the list; because the values are sorted, it is clear whether the sought value would belong before or after that middle value, and the search then continues through the correct half in the same way. Only the sign of the difference is inspected: there is no attempt at an interpolation search based on the size of the differences.

2. What is meant by sort key

A list of pairs (p,q) can be sorted based on just p. Then the comparisons in the algorithm need only consider the values of p, not those of q. For example, in a table of cities sorted on a column "country" we can find cities in Germany by comparing country names with "Germany", instead of comparing whole rows. Such partial content is called a sort key.

15.7 Lesson end Activities

1. Do we need these many searching techniques?

2. Which searching technique you feel is effective?

15.8 References

Ellis HoroWitz and Sartaj Sahni: Fundamentals of Data structure, Galgotia book source.
Ashok N Kamthane, Programming and Data structures, Pearson Education
M.Tanenbaum, Data structure using C, PHI pub.
http://www.cs.utk.edu/~plank/plank/classes/
http://www.go4expert.com/forums/
http://www.cs.bu.edu/teaching/c/

LESSON - 16 : SORTING TECHNIQUES

CONTENTS

16.0 Aims and Objectives

The objective of this lesson is to give clear information about various sorting teching, which will help the programmers to implement applications effectively using the sorting techniques.

16.1 Introduction : Sorting

The primary purpose of this chapter is to provide an extensive set of examples illustrating the use of the data structures introduced in the preceding chapter and to show

how the choice of structure for the underlying data profoundly influences the algorithms that perform a given task. Sorting is also a good example to show that such a task may be performed according to many different algorithms, each one having certain advantages and disadvantages that have to be weighed against each other in the light of the particular application.

Sorting is generally understood to be the process of rearranging a given set of objects in a specific order. The purpose of sorting is to facilitate the later search for members of the sorted set. As such it is an almost universally performed, fundamental activity. Objects are sorted in telephone books, in income tax files, in tables of contents, in libraries, in dictionaries, in warehouses, and almost everywhere that stored objects have to be searched and retrieved. Even small children are taught to put their things "in order", and they are confronted with some sort of sorting long before they learn anything about arithmetic.

Hence, sorting is a relevant and essential activity, particularly in data processing. What else would be easier to sort than data! Nevertheless, our primary interest in sorting is devoted to the even more fundamental techniques used in the construction of algorithms. There are not many techniques that do not occur somewhere in connection with sorting algorithms. In particular, sorting is an ideal subject to demonstrate a great diversity of algorithms, all having the same purpose, many of them being optimal in some sense, and most of them having advantages over others. It is therefore an ideal subject to demonstrate the necessity of performance analysis of algorithms. The example of sorting is moreover well suited for showing how a very significant gain in performance may be obtained by the development of sophisticated algorithms when
obvious methods are readily available.

16.2 Classification of sorting

The dependence of the choice of an algorithm on the structure of the data to be processed -- an ubiquitous phenomenon -- is so profound in the case of sorting that sorting methods are generally classified into two categories, namely, sorting of arrays and sorting of (sequential) files. The two classes are often called *internal* and *external sorting* because arrays are stored in the fast, high-speed, random-access "internal" store of computers and files are appropriate on the slower, but more spacious "external" stores based on mechanically moving devices (disks and tapes). The importance of this distinction is obvious from the example of sorting numbered cards. Structuring the cards as an array corresponds to laying them out in front of the sorter so that each card is visible and individually accessible (see Fig. 2.1).

Structuring the cards as a file, however, implies that from each pile only the card on the top is visible (see Fig. 2.2). Such a restriction will evidently have serious consequences on the sorting method to be used, but it is unavoidable if the number of cards to be laid out is larger than the available table.

Before proceeding, we introduce some terminology and notation to be used throughout this lesson. If we are given n items

$a_0, a_1, \ldots , a_{n-1}$

sorting consists of permuting these items into an array

$a_{k0}, a_{k1}, \ldots , a_{k[n-1]}$

such that, given an ordering function f,

$f(a_{k0}) \leq f(a_{k1}) \ldots \leq f(a_{k[n-1]})$

Ordinarily, the ordering function is not evaluated according to a specified rule of computation but is stored as an explicit component (field) of each item. Its value is called the *key* of the item. As a consequence, the record structure is particularly well suited to represent items and might for example be declared as follows:

```
TYPE Item = RECORD key: INTEGER;
        (*other components declared here*)
END
```

The other components represent relevant data about the items in the collection; the key merely assumes the purpose of identifying the items. As far as our sorting algorithms are concerned, however, the key is the only relevant component, and there is no need to define any particular remaining components. In the following discussions, we shall therefore discard any associated information and assume that the type *Item* be defined as INTEGER. This choice of the key type is somewhat arbitrary. Evidently, any type on which a total ordering relation is defined could be used just as well.

A sorting method is called *stable* if the relative order if an item with equal keys remains unchanged by the sorting process. Stability of sorting is often desirable, if items are already ordered (sorted) according to some secondary keys, i.e., properties not reflected by the (primary) key itself.

16.3 Insertion

16.3.1 Method

This method is widely used by card players. The items (cards) are conceptually divided into a destination sequence a1 ... ai-1 and a source sequence ai ... an. In each step, starting with i = 2 and incrementing i by unity, the i th element of the source sequence is picked and transferred into the destination sequence by inserting it at the appropriate place.

Initial Keys:	44	55	12	42	94	18	06	67
i=1	44	55	12	42	94	18	06	67
i=2	12	44	55	42	94	18	06	67
i=3	12	42	44	55	94	18	06	67
i=4	12	42	44	55	94	18	06	67
i=5	12	18	42	44	55	94	06	67
i=6	06	12	18	42	44	55	94	67
i=7	06	12	18	42	44	55	67	94

Table 16.3.1 A Sample Process of Straight Insertion Sorting.

The process of sorting by insertion is shown in an example of eight numbers chosen at random (see Table 16.3.1). The algorithm of straight insertion is

```
        FOR i := 1 TO n-1 DO
          x := a[i];
        insert x at the appropriate place in a0 ... ai
END
```

In the process of actually finding the appropriate place, it is convenient to alternate between comparisons and moves, i.e., to let x sift down by comparing x with the next item aj, and either inserting x or moving aj to the right and proceeding to the left. We note that there are two distinct conditions that may cause the termination of the sifting down process:

1. An item aj is found with a key less than the key of x.
2. The left end of the destination sequence is reached.

```
PROCEDURE Straight Insertion;
        VAR i, j: INTEGER; x: Item;
        BEGIN
        FOR i := 1 TO n-1 DO
        x := a[i]; j := i;
WHILE (j > 0) & (x < a[j-1] DO a[j] := a[j-1]; DEC(j) END ;
a[j] := x
END
END Straight Insertion
```

16.3.2 Analysis of straight insertion.

The number Ci of key comparisons in the i-th sift is at most i-1, at least 1, and -- assuming that all permutations of the n keys are equally probable -- i/2 in the average. The number Mi of moves (assignments of items) is Ci + 2 (including the sentinel). Therefore, the total numbers of comparisons and moves are C_{min} = n-1 Mmin = 3*(n-1)

C_{ave} = (n2 + n - 2)/4 M_{ave} = (n2 + 9n - 10)/4

C_{max} = (n2 + n - 4)/4 M_{max} = (n2 + 3n - 4)/2

The minimal numbers occur if the items are initially in order; the worst case occurs if the items are initially in reverse order. In this sense, sorting by insertion exhibits a truly natural behavior. It is plain that the given algorithm also describes a stable sorting process: it leaves the order of items with equal keys unchanged.

The algorithm of straight insertion is easily improved by noting that the destination sequence a0 ... ai-1, in which the new item has to be inserted, is already ordered. Therefore, a faster method of determining the insertion point can be used. The obvious choice is a binary search that samples the destination sequence in the middle and continues bisecting until the insertion point is found. The modified sorting algorithm is called *binary insertion*.

```
PROCEDURE BinaryInsertion(VAR a: ARRAY OF Item; n: INTEGER);
VAR i, j, m, L, R: INTEGER; x: Item;
BEGIN
FOR i := 1 TO n-1 DO
x := a[i]; L := 1; R := i;
WHILE L < R DO
m := (L+R) DIV 2;
IF a[m] <= x THEN L := m+1 ELSE R := m END
END ;
FOR j := i TO R+1 BY -1 DO a[j] := a[j-1] END ;
a[R] := x
END
END BinaryInsertion
```

16.3.3 Analysis of binary insertion.

The insertion position is found if L = R. Thus, the search interval must in the end be of length 1; and this involves halving the interval of length i log(i) times. Thus,
C = **S**i: 0≤In:□log(i)

We approximate this sum by the integral Int (0:n-1) log(x) dx = n*(log n - c) + c where c = log e = 1/ln 2 = 1.44269... . The number of comparisons is essentially independent of the initial order of the items. However, because of the truncating character of the division involved in bisecting the search interval, the true number of comparisons needed with i items may be up to 1 higher than expected. The nature of this bias is such that insertion positions at the low end are on the average located slightly faster than those at the high end, thereby favoring those cases in which the items are originally highly out of order. In fact, the minimum number of comparisons is needed if the items are initially in reverse order and the maximum if they are already in order.Hence, this is a case of unnatural behavior of a sorting algorithm. The number of comparisons is then approximately $C \approx n*(\log n - \log e \pm 0.5)$

Unfortunately, the improvement obtained by using a binary search method applies only to the number of comparisons but not to the number of necessary moves. In fact, since moving items, i.e., keys and associated information, is in general considerably more time-consuming than comparing two keys, the improvement is by no means drastic: the important term M is still of the order n2. And, in fact, sorting the already sorted array takes more time than does straight insertion with sequential search.

This example demonstrates that an "obvious improvement" often has much less drastic consequences than one is first inclined to estimate and that in some cases (that do occur) the "improvement" may actually turn out to be a deterioration. After all, sorting by insertion does not appear to be a very suitable method for digital computers: insertion of an item with the subsequent shifting of an entire row of items by a single position is uneconomical. One should expect better results from a method in which moves of items are only performed upon single items and over longer distances. This idea leads to sorting by selection.

16.4 Selection sort

16.4.1 Definition

Selection sort is the most conceptually simple of all the sorting algorithms. It works by selecting the smallest (or largest, if you want to sort from big to small) element of the array and placing it at the head of the array. Then the process is repeated for the remainder of the array; the next largest element is selected and put into the next slot, and so on down the line.

Because a selection sort looks at progressively smaller parts of the array each time (as it knows to ignore the front of the array because it is already in order), a selection sort is slightly faster than bubble sort, and can be better than a modified bubble sort.

16.4.2 Implementation

This method is based on the following principle:

1. Select the item with the least key.
2. Exchange it with the first item a0.
3. Then repeat these operations with the remaining n-1 items, then with n-2 items, until only one item - the largest -- is left.

This method is shown on the same eight keys as in Table 16.6.1.

Initial keys	44	55	12	42	94	18	06	67
	06	55	12	42	94	18	44	67
	06	12	55	42	94	18	44	67
	06	12	18	42	94	55	44	67
	06	12	18	42	94	55	44	67
	06	12	18	42	44	55	94	67
	06	12	18	42	44	55	94	67
	06	12	18	42	44	55	67	94

Table 16.6.2.1 A Sample Process of Straight Selection Sorting.

The algorithm is formulated as follows:

```
FOR i := 0 TO n-1 DO
   assign the index of the least item of $a_i$ ... $a_{n-1}$ to k;
   exchange $a_i$ with $a_k$
END
```

This method, called *straight selection*, is in some sense the opposite of straight insertion: Straight insertion considers in each step only the one next item of the source sequence and all items of the destination array to find the insertion point; straight selection considers all items of the source array to find the one with the least key and to be deposited as the one next item of the destination sequence..

Example 16.1

```
PROCEDURE Straight Selection;
  VAR i, j, k: INTEGER; x: Item;
BEGIN
  FOR i := 0 TO n-2 DO
    k := i; x := a[i];
    FOR j := i+1 TO n-1 DO
        IF a[j] < x THEN k := j; x := a[k] END
    END ;
    a[k] := a[i]; a[i] := x
  END
```

END Straight Selection

16.4.3 Analysis of straight selection.

Evidently, the number C of key comparisons is independent of the initial order of keys. In this sense, this method may be said to behave less naturally than straight insertion. We obtain

$$C = (n2 - n)/2$$

The number M of moves is at least

$M_{min} = 3*(n-1)$

in the case of initially ordered keys and at most

$$M_{max} = n2/4 + 3*(n-1)$$

if initially the keys are in reverse order. In order to determine Mavg we make the following deliberations: The algorithm scans the array, comparing each element with the minimal value so far detected and, if smaller than that minimum, performs an assignment. The probability that the second element is less than the first, is 1/2; this is also the probability for a new assignment to the minimum. The chance for the third element to be less than the first two is 1/3, and the chance of the fourth to be the smallest is 1/4, and so on. Therefore the total expected number of moves is Hn-1, where H_n is the n the harmonic number

$$H_n = 1 + 1/2 + 1/3 + ... + 1/n$$

Hn can be expressed as

$$H_n = \ln(n) + g + 1/2n - 1/12n2 + ...$$

where g = 0.577216... is Euler's constant. For sufficiently large n, we may ignore the fractional terms and therefore approximate the average number of assignments in the i the pass as

$$F_i = \ln(i) + g + 1$$

The average number of moves M_{avg} in a selection sort is then the sum of F_i with i ranging from 1 to n.

$$M_{avg} = n*(g+1) + (\mathbf{S}i: 1 \leq in: \ln(i))$$

By further approximating the sum of discrete terms by the integral

$$\text{Integral } (1:n) \ln(x)\, dx = n * \ln(n) - n + 1$$

we obtain an approximate value

$M_{avg} = n * (\ln(n) + g)$

16.5 Let us Sum up

In this lesson, we discussed about sorting algorithms and its classifications, Implementation of Insertion sort, algorithm for straight insertion algorithm and its analysis, time and space complexities of sorting algorithm. We also discussed briefly about selection sort, implementation and complexities of selection algorithm.

16.6 Points for discussion

1. Define the term sorting
2. Explain the general classification of sorting
3. How binary insertion differes from straight insertion?
4. Explain the implementation of selection sort/

1678 Check your progress

1. Explain the general classification of sorting

The two classes are often called *internal* and *external sorting* because arrays are stored in the fast, high-speed, random-access "internal" store of computers and files are appropriate on the slower, but more spacious "external" stores based on mechanically moving devices (disks and tapes). The importance of this distinction is obvious from the example of sorting numbered cards.

2. Define selection sort.

Selection sort is the most conceptually simple of all the sorting algorithms. It works by selecting the smallest (or largest, if you want to sort from big to small) element of the array and placing it at the head of the array. Then the process is repeated for the remainder of the array; the next largest element is selected and put into the next slot, and so on down the line.

16.8 Lesson-end Activities

1. Which sorting technique will be effective one?
2. Do we need to arrange data before performing binary search?
3. Compare binary search with linear search.

16.9 Suggested Readings/references/Sources

Ellis HoroWitz and Sartaj Sahni: Fundamentals of Data structure, Galgotia book source.
Ashok N Kamthane, Programming and Data structures, Pearson Education
M.Tanenbaum, Data structure using C, PHI pub.
http://www.programmersheaven.com/download/
http://en.literateprograms.org/
http://www.cs.utk.edu/~plank/plank/classes/
http://www.go4expert.com/forums/
http://www.cs.bu.edu/teaching/c/

LESSON - 17 : BUBBLE SORT

CONTENTS

17.0 Aims and objectives

In previous lesson, we discussed about fundamentals of sorting algorithms and importance of insertion and selection algorithm. The aim of this lesson is to make the readers to to get enough knowledge about some more sorting algorithms like bubble sort and quick sort.

17.1 Bubble sort

17.1.1 Introduction :

Bubble sort is a simple sorting algorithm. It works by repeatedly stepping through the list to be sorted, comparing two items at a time and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which means the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top (i.e. the beginning) of the list via the swaps. Sort by comparing each adjacent pair of items in a *list* in turn, swapping the items if necessary, and repeating the pass through the list until no swaps are done.

17.1.2 Algorithm for bubble sort

This is probably the simplest way sort an array of objects. Unfortunately it is also the slowest way! The basic idea is to compare two **neighboring** objects, and to swap them if they are in the wrong order. Given an array a of numbers, with length n, here's a snippet of C code for bubble sort:

Example 17.1

```
for (i=0; i<n-1; i++)
{
  for (j=0; j<n-1-i; j++)
    if (a[j+1] < a[j]) {  /* compare the two neighbors */
      tmp = a[j];         /* swap a[j] and a[j+1]      */
      a[j] = a[j+1];
      a[j+1] = tmp;
  }
}
```

As we can see, the algorithm consists of two nested loops. The index j in the inner loop travels up the array, comparing adjacent entries in the array (at j and j+1), while the outer loop causes the inner loop to make repeated passes through the array. After the first pass, the largest element is guaranteed to be at the end of the array, after the second pass, the second largest element is in position, and so on. That is why the upper bound in the inner loop (n-1-i) decreases with each pass: we don't have to re-visit the end of the array.

17.1.3 Implementation of Source Code

```
void bubbleSort(int numbers[], int array_size)
{
  int i, j, temp;

  for (i = (array_size - 1); i >= 0; i--)
  {
    for (j = 1; j <= i; j++)
    {
      if (numbers[j-1] > numbers[j])
      {
        temp = numbers[j-1];
        numbers[j-1] = numbers[j];
        numbers[j] = temp;
      }
    }
  }
}
```
Example 17.2

17.2 Quicksort

17.2.1 Definition

As one of the more advanced sorting, you might think that the Quicksort is steeped in complicated theoretical background, but this is not so. Like Insertion Sort, this sort has a fairly simple concept at the core, but is made complicated by the constraints of the array structure.

The basic concept is to pick one of the elements in the array as a pivot value around which the other elements will be rearranged. Everything less than the pivot is moved left of the pivot (into the left partition). Similarly, everything greater than the pivot goes into the right partition. At this point each partition is recursively quicksorted.

The Quicksort algorithm is fastest when the median of the array is chosen as the pivot value. That is because the resulting partitions are of very similar size. Each partition splits itself in two and thus the base case is reached very quickly.

17.2.2 Implementation

In practice, the Quicksort algorithm becomes very slow when the array passed to it is already close to being sorted. Because there is no efficient way for the computer to find the median element to use as the pivot, the first element of the array is used as the pivot. So when the array is almost sorted, Quicksort doesn't partition it equally. Instead, the partitions are lopsided like in Figure 17.2.2.1. This means that one of the recursion branches is much deeper than the other, and causes execution time to go up. Thus, it is said that the more random the arrangement of the array, the faster the Quicksort Algorithm finishes.

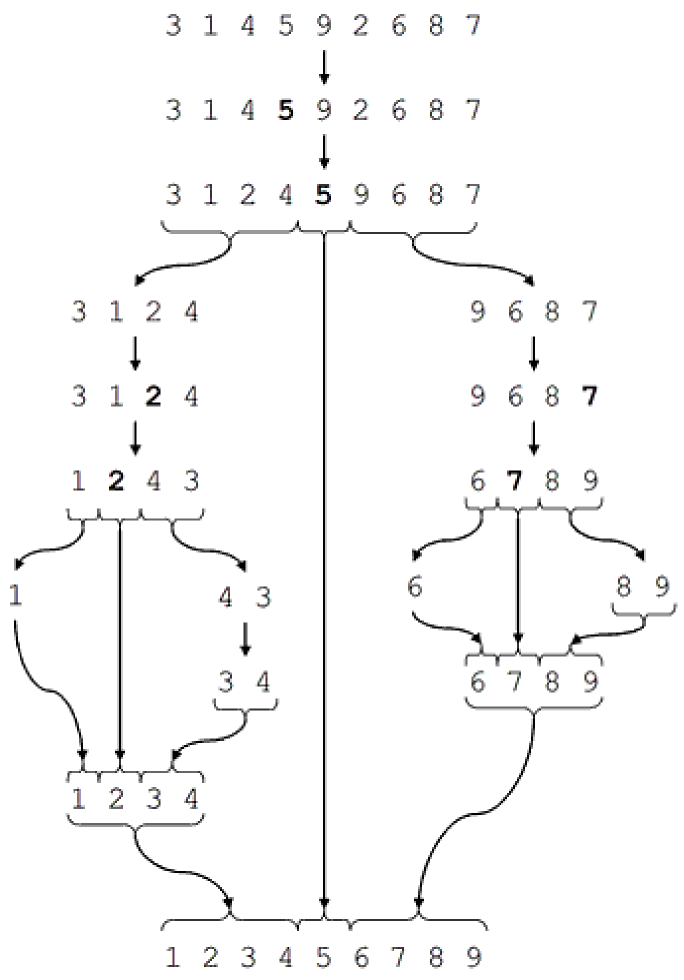


Figure 17.2.2.1: The ideal Quicksort on a random array.

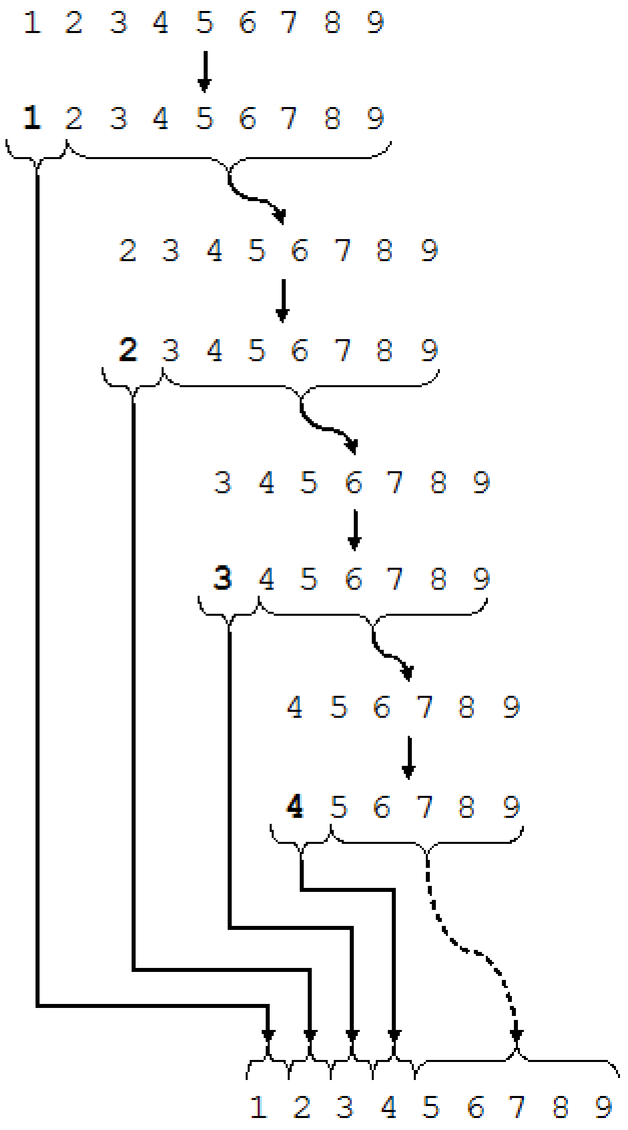


Figure 17.2.2.2: Quicksort on an already sorted array.

These are the steps taken to sort an array using QuickSort

Elements in underline indicate swaps.
Elements in bold indicate comparisons.

3 1 4 5 9 2 6 8 7

17.2.3 Points on Quick sort

A "small" element is one whose value is less than or equal to the value of the pivot. Likewise, a "large" element is one whose value is larger than that of the pivot. At the beginning, the entire array is passed into the quicksort function and is essentially treated as one large partition. At this time, two indices are initialized: the left-to-right search index, *i*, and the right-to-left search index, *k*. The value of *i* is the index of the first element in the partition, in this case 0, and the value of *k* is 8, the index of the last element in the partition. The relevance of these variables will be made apparent in the code below.

3 1 4 5 9 2 6 8 7

The first element in the partition, 3, is chosen as the pivot element, around which two subpartitions will be created. The end goal is to have all the small elements at the front of the partition, in no particular order, followed by the pivot, followed by the large elements. To do this, quicksort will scan rightwards for the first large element. Once this is found, it will look for the first small element from the right. These two will then be swapped. Since *i* is currently set to zero, the pivot is actually compared to itself in the search of the first large element.

3 **1** 4 5 9 2 6 8 7

The search for the first large element continues rightwards. The value of *i* gets incremented as the search moves to the right.

3 1 **4** 5 9 2 6 8 7

Since 4 is greater than the pivot, the rightwards search stops here. Thus the value of *i* remains 2.

3 1 4 5 9 2 6 8 **7**

Now, starting from the right end of the array, quicksort searches for the first small element. And so *k* is decremented with each step leftwards through the partition.

3 1 4 5 9 2 6 **8** 7
3 1 4 5 9 2 **6** 8 7
3 1 4 5 9 **2** 6 8 7

Since 2 is not greater than the pivot, the leftwards search can stop.

3 1 $\underline{2}$ 5 9 $\underline{4}$ 6 8 7

Now elements 4 and 2 (at positions 2 and 5, respectively) are swapped.

3 1 **2** 5 9 4 6 8 7

Next, the rightwards search resumes where it left off: at position 2, which is stored in the index *i*.

3 1 2 **5** 9 4 6 8 7

Immediately a large element is found, and the rightwards search stops with *i* being equal to 3.

3 1 2 5 9 **4** 6 8 7

Next the leftwards search, too, resumes where it left off: *k* was 5 so the element at position 5 is compared to the pivot before *k* is decremented again in search of a small element.

3 1 2 5 9 4 6 8 7

This continues without any matches for some time...

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

The small element is finally found, but no swap is performed since at this stage, *i* is equal to *k*. This means that all the small elements are on one side of the partition and all the large elements are on the other.

<u>2</u> 1 <u>3</u> 5 9 4 6 8 7

Only one thing remains to be done: the pivot is swapped with the element currently at *i*. This is acceptable within the algorithm because it only matters that the small element be to the left of the pivot, but their respective order doesn't matter. Now, elements 0 to (*i* - 1) form the left partition (containing all small elements) and elements *k* + 1 onward form the right partition (containing all large elements. Calling quickSort on elements 0 to 1 The right partition is passed into the quicksort function.

2 1 3 5 9 4 6 8 7

2 is chosen as the pivot. It is also compared to itself in the search for a small element within the partition.

2 1 3 5 9 4 6 8 7

The first, and in this case only, small element is found.

2 1 3 5 9 4 6 8 7

Since the partition has only two elements, the leftwards search begins at the second element and finds 1.

<u>1 2</u> 3 5 9 4 6 8 7

The only swap to be made is actually the final step where the pivot is inserted between the two partitions. In this case, the left partition has only one element and the right partition has zero elements. Calling quickSort on elements 0 to 0 Now that the left partition of the partition above is quicksorted: there is nothing else to be done Calling quickSort on elements 2 to 1 The right partition of the partition above is quicksorted. In this case the starting index is greater than the ending index due to the way these are generated: the right partition starts one past the pivot of its parent partition and goes until the last element of the parent partition. So if the parent partition is empty, the indices generated will be out of bounds, and thus no quicksorting will take place. Calling quickSort on elements 3 to 8 The right partition of the entire array is now being quicksorted 5 is chosen as the pivot.

1 2 3 **5** 9 4 6 8 7
1 2 3 **5** **9** 4 6 8 7

The rightwards scan for a large element is initiated. 9 is immediately found.

1 2 3 **5** 9 4 6 8 **7**

Thus, the leftwards search for a small element begins...

1 2 3 **5** 9 4 6 **8** 7
1 2 3 **5** 9 4 **6** 8 7
1 2 3 **5** 9 **4** 6 8 7

At last, 4 is found. Note *k* = 5.

1 2 3 5 <u>4 9</u> 6 8 7

Thus the first large and small elements to be found are swapped.

1 2 3 **5** **4** 9 6 8 7

The rightwards search for a large element begins anew.

1 2 3 **5** 4 **9** 6 8 7

Now that it has been found, the rightward search can stop.

1 2 3 **5** 4 **9** 6 8 7

Since *k* was stopped at 5, this is the index from which the leftward search resumes.

1 2 3 **5** **4** 9 6 8 7

1 2 3 4 5 9 6 8 7

The last step for this partition is moving the pivot into the right spot. Thus the left partition consists only of the element at 3 and the right partition is spans positions 5 to 8 inclusive.

Calling quickSort on elements 3 to 3
The left partition is quicksorted (although nothing is done.

Calling quickSort on elements 5 to 8
The right partition is now passed into the quicksort function.

1 2 3 4 5 **9** 6 8 7

9 is chosen as the pivot.

1 2 3 4 5 **9** **6** 8 7

The rightward search for a large element begins.

1 2 3 4 5 **9** 6 **8** 7
1 2 3 4 5 **9** 6 8 **7**

No large element is found. The search stops at the end of the partition.

1 2 3 4 5 **9** 6 8 **7**

The leftwards search for a small element begins, but does not continue since the search indices *i* and *k* have crossed.

1 2 3 4 5 7 6 8 9

The pivot is swapped with the element at the position *k*: this is the last step in splitting this partition into left and right subpartitions. Calling quickSort on elements 5 to 7 The left partition is passed into the quicksort function.

1 2 3 4 5 **7** 6 8 9

6 is chosen as the pivot.

1 2 3 4 5 **7** **6** 8 9

The rightwards search for a large element begins from the left end of the partition.

1 2 3 4 5 **7** 6 **8** 9

The rightwards search stops as 8 is found.

1 2 3 4 5 **7** 6 **8** 9

The leftwards search for a small element begins from the right end of the partition.

1 2 3 4 5 **7** **6** 8 9

Now that 6 is found, the leftwards search stops. As the search indices have already crossed, no swap is performed.

1 2 3 4 5 <u>6 7</u> 8 9

So the pivot is swapped with the element at position *k*, the last element compared to the pivot in the leftwards search.

Calling quickSort on elements 5 to 5
The left subpartition is quicksorted. Nothing is done since it is too small.

Calling quickSort on elements 7 to 7
Likewise with the right subpartition.

Calling quickSort on elements 9 to 8

Due to the "sort the partition startitng one to the right of the pivot" construction of the algorithm, an empty partition is passed into the quicksort function. Nothing is done for this base case.

1 2 3 4 5 6 7 8 9

Finally, the entire array has been sorted.

17.3 Program for quick sort

Example 17.3

```
// quickSort.c
#include <stdio.h>

void quickSort( int[], int, int);
int partition( int[], int, int);


void main()
{
        int a[] = { 7, 12, 1, -2, 0, 15, 4, 11, 9};

        int i;
        printf("\n\nUnsorted array is:  ");
        for(i = 0; i < 9; ++i)
                printf(" %d ", a[i]);

        quickSort( a, 0, 8);

        printf("\n\nSorted array is:  ");
        for(i = 0; i < 9; ++i)
                printf(" %d ", a[i]);

}



void quickSort( int a[], int l, int r)
{
   int j;

   if( l < r )
   {
        // divide and conquer
        j = partition( a, l, r);
       quickSort( a, l, j-1);
       quickSort( a, j+1, r);
   }

}



int partition( int a[], int l, int r) {
   int pivot, i, j, t;
   pivot = a[l];
   i = l; j = r+1;

   while( 1)
   {
        do ++i; while( a[i] <= pivot && i <= r );
        do --j; while( a[j] > pivot );
        if( i >= j ) break;
        t = a[i]; a[i] = a[j]; a[j] = t;
   }
   t = a[l]; a[l] = a[j]; a[j] = t;
   return j;
}
```

17.4 Let us Sum up

In this lesson, we discussed about the way to implement bubble sort algorithm, advantages and disadvantages, the importance of quick sort, need and its implementation and difference between bubble sort and quick sort. This lesson helps reader to implement applications based on these two sorting algorithm.

17.5 Points for discussion

1. How bubble sort differ from other sorting methods?
2. What will be the time complexity of quick sort?
3. What are the advantages and disadvantages of quick sort?
4. What is meant by pivot element?

17.6 Check your progress

1. Define quick sort?

The basic concept is to pick one of the elements in the array as a pivot value around which the other elements will be rearranged. Everything less than the pivot is moved left of the pivot (into the left partition). Similarly, everything greater than the pivot goes into the right partition. At this point each partition is recursively quicksorted.

2. Define bubble sort

Bubble sort is a simple sorting algorithm. It works by repeatedly stepping through the list to be sorted, comparing two items at a time and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which means the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top (i.e. the beginning) of the list via the swaps. Sort by comparing each adjacent pair of items in a *list* in turn, swapping the items if necessary, and repeating the pass through the list until no swaps are done.

17.7 Lesson-end Activities

1. What will be the time requirement of bubble sort?
2. which sorting technique is generally suggestable?

17.8 References

Ellis HoroWitz and Sartaj Sahni: Fundamentals of Data structure, Galgotia book source.
Ashok N Kamthane, Programming and Data structures, Pearson Education
M.Tanenbaum, Data structure using C, PHI pub.
http://www.programmersheaven.com/download/
http://en.literateprograms.org/
http://www.cs.utk.edu/~plank/plank/classes/
http://www.go4expert.com/forums/
http://www.cs.bu.edu/teaching/c/

LESSON - 18 : TREE AND HEAP STRUCTURES

CONTENTS

18.0 Aims and Objestives

The objective of this lesson is to make reader to understand above the concept and applications of three and heap structures. This will help programmer to implement various applications based on tree structures.

18.1 Basic Concepts and Definitions

We have seen that sequences and lists may conveniently be defined in the following way: A sequence (list) with base type T is either

1. The empty sequence (list).
2. The concatenation (chain) of a T and a sequence with base type T.

Hereby recursion is used as an aid in defining a structuring principle, namely, sequencing or iteration. Sequences and iterations are so common that they are usually considered as fundamental patterns of structure and behaviour. But it should be kept in mind that they can be defined in terms of recursion, whereas the reverse is not true, for recursion may be effectively and elegantly used to define much more sophisticated structures.

18.2 Representation of Tree

Trees are a well-known example. Let a tree structure be defined as follows: A tree structure with base type T is either

1. The empty structure.
2. A node of type T with a finite number of associated disjoint tree structures of base type T, called subtrees.

From the similarity of the recursive definitions of sequences and tree structures it is evident that the sequence (list) is a tree structure in which each node has at most one subtree. The list is therefore also called a degenerate tree.

There are several ways to represent a tree structure. For example, a tree structure with its base type T ranging over the letters is shown in various ways in Fig. 18.2.1. These representations all show the same structure and are therefore equivalent. It is the graph structure that explicitly illustrates the branching relationships which, for obvious reasons, led to the generally used name tree. Strangely enough, it is customary to depict trees upside down to show the roots of trees. The latter formulation, however, is misleading, since the top node (A) is commonly called the root.

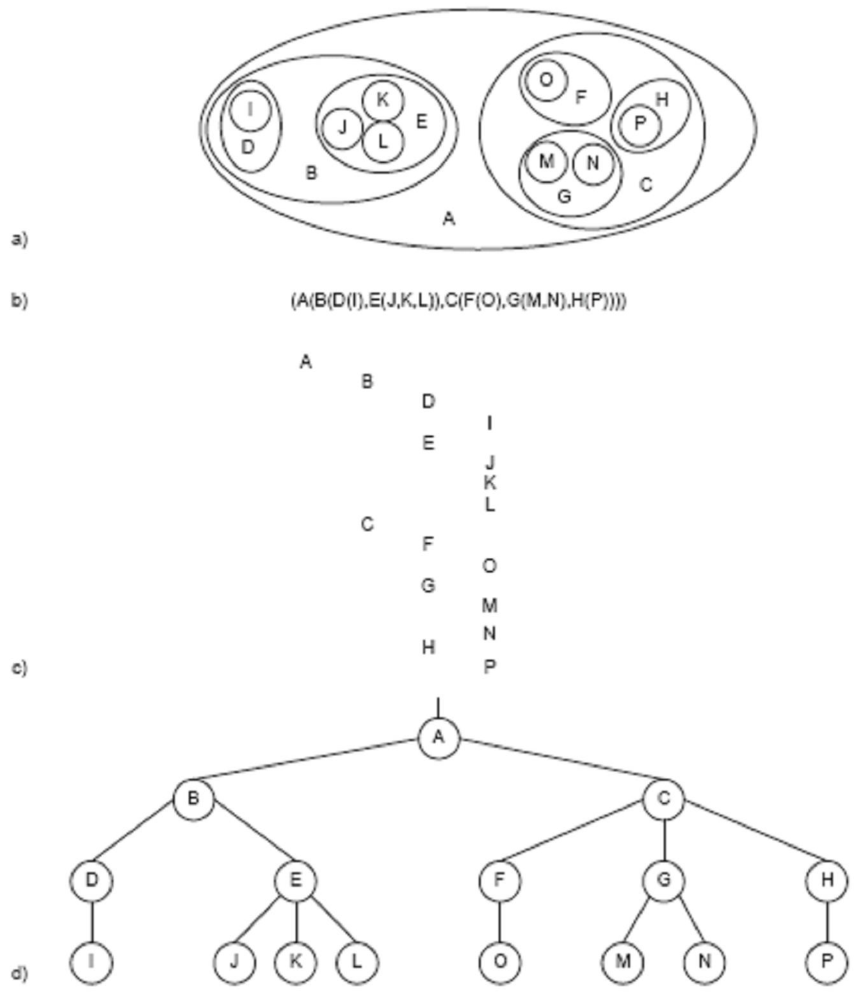


Figure 18.2.1: Representation of tree structure by (a) nested sets,
(b) nested parentheses, (c) indented text, and (d) graph

An ordered tree is a tree in which the branches of each node are ordered. Hence the two ordered trees in Fig 18.2.2 are distinct, different objects. A node y that is directly below node x is called a (direct) *descendant* of x; if x is at level i, then y is said to be at level i+1. Conversely, node x is said to be the (direct) *ancestor* of y. The root of a tree is defined to be at level 0. The maximum level of any element of a tree is said to be its depth or *height.*

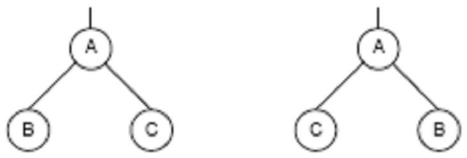


Figure : 18.2.2 Two distinct trees

If an element has no descendants, it is called a *terminal* node or a *leaf*; and an element that is not terminal is an interior node. The number of (direct) descendants of an interior node is called its *degree*. The maximum degree over all nodes is the degree of the tree. The number of branches or edges that have to be traversed in order to proceed from the root to a node x is called the *path length* of x. The root has path length 0, its direct descendants have path length 1, etc. In general, a node at level i has path length i. The path length of a tree is defined as the sum of the path lengths of all its components. It is also called its *internal path length*. The internal path length of the tree shown in Fig., for instance, is 36. Evidently, the average path length is

P_{int} = (**S**i: 1≤in: ni×i) / n

where ni is the number of nodes at level i. In order to define what is called the *external path length*, we extend the tree by a special node wherever a subtree was missing in the original tree. In doing so, we assume that all nodes are to have the same degree, namely the degree of the tree. Extending the tree in this way therefore amounts to filling up empty branches, whereby the special nodes, of course, have no further descendants. The tree of Fig. 18.2.3 extended with special nodes is shown in Fig,in which the special nodes are represented by squares. The external path length is now defined as the sum of the path lengths over all special nodes. If the number of special nodes at level i is mi, then the average external path length is

P_{ext} = (**S**i: 1≤imi×i) / m

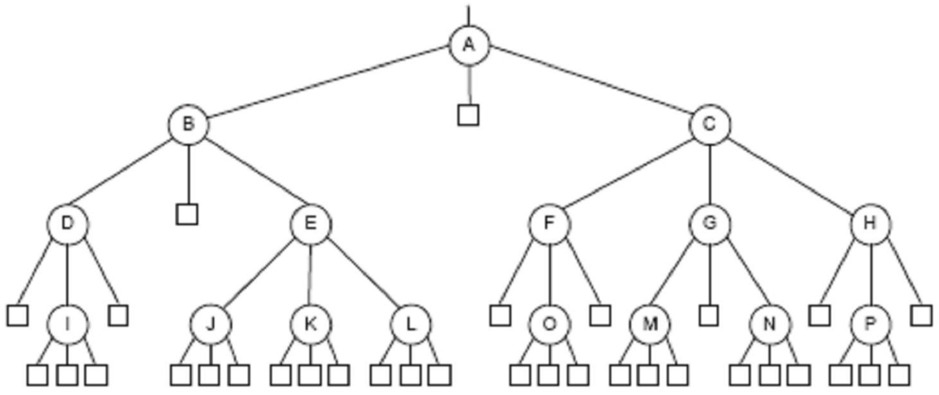


Fig. 18.2.3 Ternary tree extended with special nodes

In the tree shown in figure, the external path length is 120. The number of special nodes m to be added in a tree of degree d directly depends on the number n of original nodes. Note that every node has exactly one edge pointing to it. Thus, there are m+n edges in the extended tree. On the other hand, d edges are emanating from each original node, none from the special nodes. Therefore, there exist d*n + 1 edges, the 1 resulting from the edge pointing to the root. The two results yield the following equation between the number m of special nodes and n of original nodes: $d \times n + 1 = m+n$, or

$$m = (d-1) \times n + 1$$

The maximum number of nodes in a tree of a given height h is reached if all nodes have d subtrees, except those at level h, all of which have none. For a tree of degree d, level 0 then contains 1 node (namely, the root), level 1 contains its d descendants, level 2 contains the d2 descendants of the d nodes at level 2, etc.

This yields

$$N_d(h) = \mathbf{S}i: 0 \leq i < h: d^i$$

as the maximum number of nodes for a tree with height h and degree d. For d = 2, we obtain

$$N_2(h) = 2^h - 1$$

Of particular importance are the ordered trees of degree 2. They are called *binary* trees. We define an ordered binary tree as a finite set of elements (nodes) which either is empty or consists of a root (node) with two disjoint binary trees called the *left* and the *right subtree* of the root. In the following sections we shall exclusively deal with binary trees, and we therefore shall use the word tree to mean *ordered binary tree*. Trees with degree greater than 2 are called *multiway trees.*

Familiar examples of binary trees are the family tree (pedigree) with a person's father and mother as descendants , the history of a tennis tournament with each game being a node denoted by its winner and the two previous games of the combatants as its descendants, or an arithmetic expression with dyadic operators, with each operator denoting a branch node with its operands as subtrees shown in firure 18.2.4.

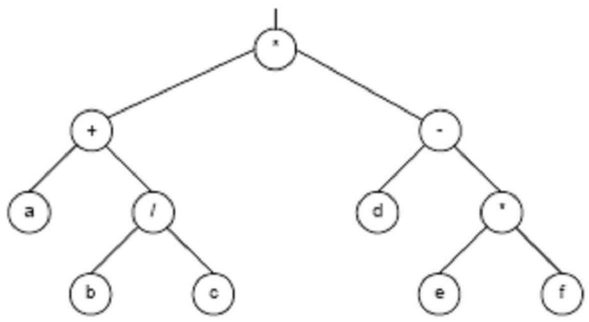
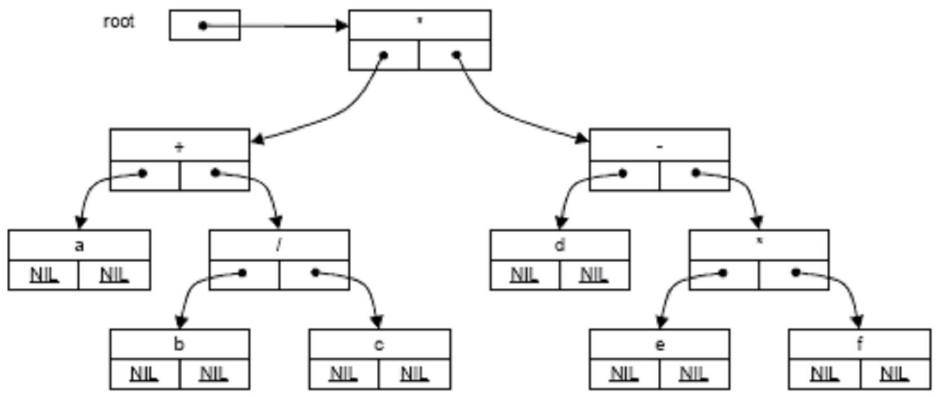


figure 18.2.4 Tree representation of expression (a + b/c) * (d – e*f)

18.3 Linked list representation of trees

We now turn to the problem of representation of trees. It is plain that the illustration of such recursive structures in terms of branching structures immediately suggests the use of our pointer facility. There is evidently no use in declaring variables with a fixed tree structure; instead, we define the nodes as variables with a fixed structure, i.e., of a fixed type, in which the degree of the tree determines the number of pointer components referring to the node's subtrees. Evidently, the reference to the empty tree is denoted by NIL. Hence, the tree of Figure 18.2.4 consists of components of a type defined as follows and may then be constructed as shown in Figure 18.3.1.

```
TYPE Node = POINTER TO NodeDesc;
TYPE NodeDesc = RECORD op: CHAR; left, right: Node END
```



18.3.1 Linked list representation of tree structure

18.4 Implementation of trees

Before investigating how trees might be used advantageously and how to perform operations on trees, we give an example of how a tree may be constructed by a program. Assume that a tree is to be generated containing nodes with the values of the nodes being n numbers read from an input file. In order to make the problem more challenging, let the task be the construction of a tree with n nodes and minimal height. In order to obtain a minimal height for a given number of nodes, one has to allocate the maximum possible number of nodes of all levels except the lowest one. This can clearly be achieved by distributing incoming nodes equally to the left and right at each node. This implies that we structure the tree for given n as shown in Figure 18.4.1., for n = 1, ... , 7.

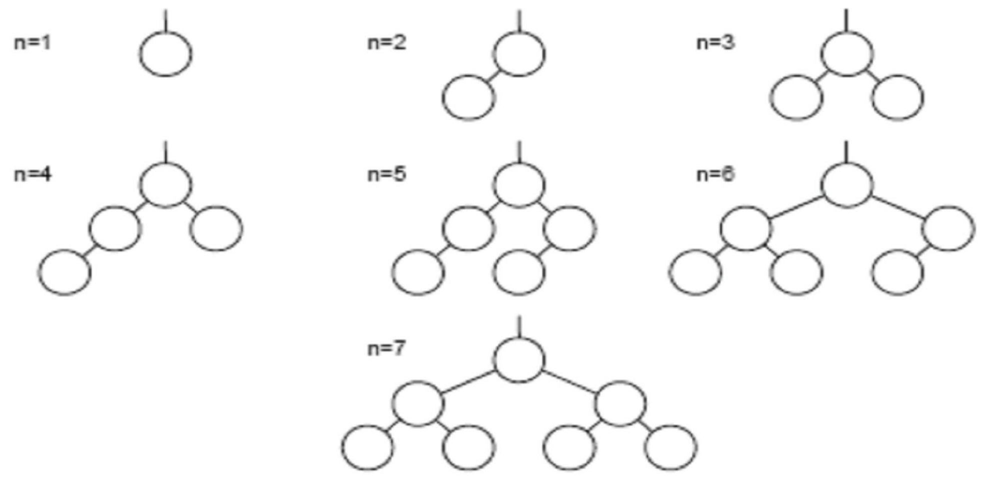


Figure 18.4.1. Perfectly balanced trees

The rule of equal distribution under a known number n of nodes is best formulated recursively:

1. Use one node for the root.
2. Generate the left subtree with nl = n DIV 2 nodes in this way.
3. Generate the right subtree with nr = n - nl - 1 nodes in this way.

The rule is expressed as a recursive procedure which reads the input file and constructs the perfectly balanced tree. We note the following definition: A tree is perfectly balanced, if for each node the numbers of nodes in its left and right subtrees differ by at most 1.

Example 18.1

```
TYPE Node = POINTER TO RECORD
        key: INTEGER; left, right: Node
      END ;

VAR R: Texts.Reader; W: Texts.Writer; root: Node;

PROCEDURE tree(n: INTEGER): Node;
    VAR new: Node;
        x, nl, nr: INTEGER;
BEGIN (*construct perfectly balanced tree with n nodes*)
    IF n = 0 THEN new := NIL
```

```
      ELSE nl := n DIV 2; nr := n-nl-1;
        NEW(new); Texts.ReadInt(R, new.key);
        new.key := x; new.left := tree(nl); new.right := tree(nr)
      END ;
      RETURN new
END tree;

PROCEDURE PrintTree(t: Node; h: INTEGER);
    VAR i: INTEGER;
BEGIN (*print tree t with indentation h*)
    IF t # NIL THEN
          PrintTree(t.left, h+1);
          FOR i := 1 TO h DO Texts.Write(W, TAB) END ;
          Texts.WriteInt(W, key, 6); Texts.WriteLn(W);
          PrintTree(t.right, h+1)
    END
END PrintTree;
```

Assume, for example, the following input data for a tree with 21 nodes:

8 9 11 15 19 20 21 7 3 2 1 5 6 4 13 14 10 12 17 16 18

The call *root := tree(21)* reads the input dara while constructing the perfectly balanced tree shown in Figure 18.4.1 . Note the simplicity and transparency of this program that is obtained through the use of recursive procedures. It is obvious that recursive algorithms are particularly suitable when a program is to manipulate information whose structure is itself defined recursively. This is again manifested in the procedure which prints the resulting tree: The empty tree results in no printing, the subtree at level L in first printing its own left subtree, then the node, properly indented by preceding it with L tabs, and finally in printing its right subtree.

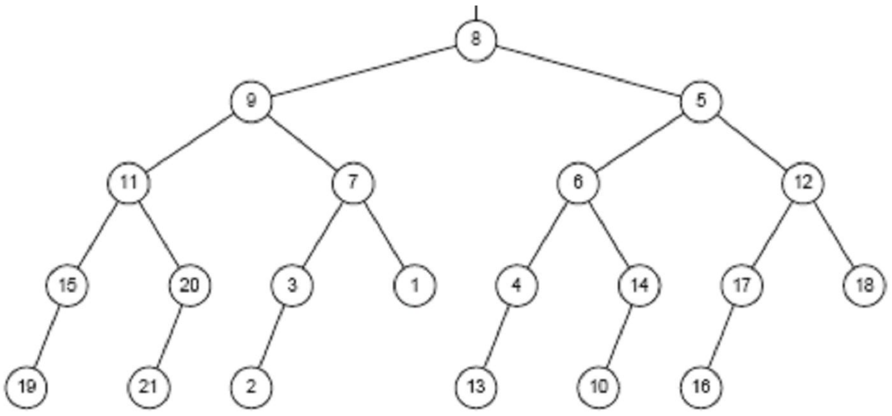


Figure 18.4.1 :Tree generated by preceding program

18.5 Traversal

Definition: A technique for processing the *nodes* of a *tree* in some order.

Pre-Order

Here the root of the subtree is processed first before going into the left then right subtree.

In-Order

After the complete processing of the left subtree the root is processed followed by the processing of the complete right subtree.

Post-Order

The root is processed only after the complete processing of the left and right subtree.

Level-Order

Using a queue the tee is processed by levels. So first all nodes on level i are processed from left to right before the first node of level i+1 is visited.

18.5.1 Preorder traversal

Definition: Process all *nodes* of a *tree* by processing the *root*, then *recursively* processing all sub trees.

```
preorder(tree)
 begin
    if tree is null, return;

    print(tree.root);
    preorder(tree.left_subtree);
    preorder(tree.right_subtree);
 end
```

18.5.2 In-order traversal

Definition: Process all *nodes* of a *tree* by *recursively* processing the left sub tree, then processing the *root*, and finally the right sub tree.

```
inorder(tree)
 begin
    if tree is null, return;
    inorder(tree.left_subtree);
    print(tree.root);
    inorder(tree.right_subtree);
 end
```

18.5.3 Post order traversal

Definition: Process all *nodes* of a *tree* by *recursively* processing all sub trees, then finally processing the *root*.

```
postorder(tree)
 begin
    if tree is null, return;

    postorder(tree.left_subtree);
    postorder(tree.right_subtree);
    print(tree.root);
 end
```

18.5.4 Level-order traversal

Definition: Process all *nodes* of a *tree* by *depth*: first the *root*, then the *children* of the root, etc. Equivalent to a *breadth-first search* from the root.

```
levelorderAux(tree, level)
 begin
    if tree is null, return;

    if level is 1, then
       print(tree.root);
    else if level greater than 1, then
       levelorderAux(tree.left_subtree, level-1);
       levelorderAux(tree.right_subtree, level-1);
    endif
 end

 levelorder(tree)
 begin
    for d = 1 to height(tree)
       levelorderAux(tree, d);
    endfor
 end
```

18.6 HEAP

18.6.1 Definition

The data structure of the heapsort algorithm is a heap. The data sequence to be sorted is stored as the labels of the binary tree. As shown later, in the implementation no pointer structures are necessary to represent the tree, since an almost complete binary tree can be efficently stored in an array.

The heap data structure is an array object which can be easily visualized as a complete binary tree.There is a one to one correspondence between elements of the array and nodes of the tree.The tree is completely filled on all levels except possibly the lowest,which is filled from the left upto a point.All nodes of heap also satisfy the relation that the key value at each node is at least as large as the value at its children.

18.6.2 Procedure get an element from heap structures.

The following procedure will helps you to access an element from the heap structure.

Step I: The user inputs the size of the heap(within a specified limit).The program generates a corresponding binary tree with nodes having randomly generated key Values.

Step II: Build Heap Operation: Let n be the number of nodes in the tree and i be the key of a tree. For this, the program uses operation Heapify.when Heapify is called both the left and right subtree of the i are Heaps.The function of Heapify is to let i settle down to a position(by swapping itself with the larger of its children,whenever the heap property is not satisfied) till the heap property is satisfied in the tree which was rooted at (i).This operationcalls.

Step III: Remove maximum element:The program removes the largest element of the heap(the root) by swapping it with the last element.
Step IV: The program executes Heapify(new root) so that the resulting tree satisfies the heap property.

Step V: Goto step III till heap is empty.

Example:

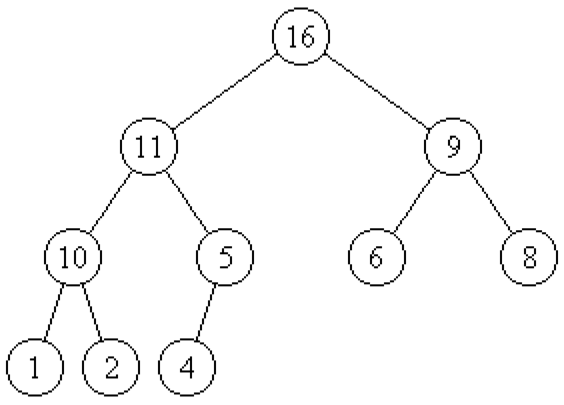


Figure 18.1: *Heap with n* = 10 *vertices*

Observe that each leaf automatically has the heap property regardless of its label, since it has no descendants.

The following description of heapsort refers to Figure 2 (a) - (e).

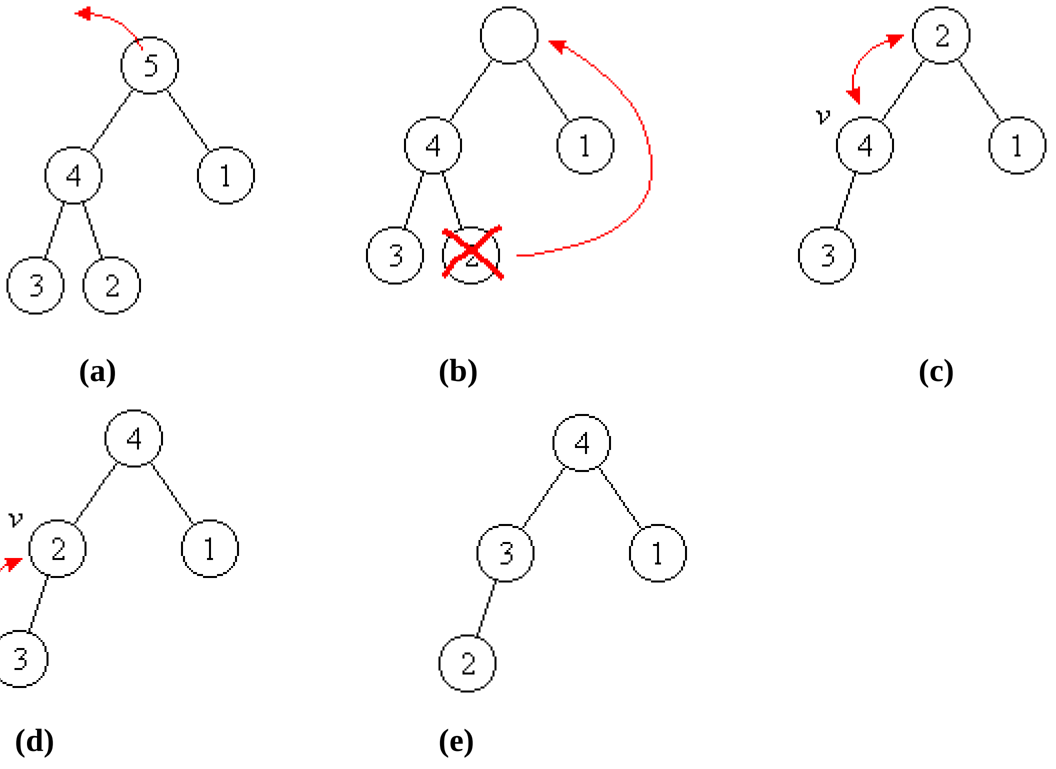


Figure 18.2: *Retrieving the maximum element and restoring the heap*

If the sequence to be sorted is arranged as a heap, the greatest element of the sequence can be retrieved immediately from the root (a). In order to get the next-greatest element, the rest of the elements have to be rearranged as a heap.

18.7 Let us sum up

In this lesson, we discussed about fundamental information of tree structure, tree traversal, representation and implementation of trees. We also discussed about heap structure, rules for accessing an element from heap, representation of heap structures.

18.8 Points for discussion

1. How to define tree structure?

2. What is ment by balanced tree?

3. Define the terms terminal node and depth of the tree

4. what is meny by heap?

18.9 Check your progress

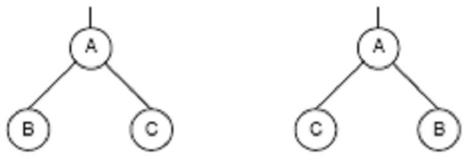
1. Define the term tree.

A tree structure with base type T is either

1. The empty structure.
2. A node of type T with a finite number of associated disjoint tree structures of base type T, called subtrees.

2. Define the terms ordered tree, ancestor and descendant.

An ordered tree is a tree in which the branches of each node are ordered. Hence the two ordered trees in following figure are distinct, different objects. A node y that is directly below node x is called a (direct) *descendant* of x; if x is at level i, then y is said to be at level i+1. Conversely, node x is said to be the (direct) *ancestor* of y.



18.10 Lesson-end Activities

1. How tree structure will differ from other structures?
2. Where we can use this tree structure?
3. How to identify leaf node?

18.11 References

Ellis HoroWitz and Sartaj Sahni: Fundamentals of Data structure, Galgotia book source.
Ashok N Kamthane, Programming and Data structures, Pearson Education
M.Tanenbaum, Data structure using C, PHI pub.
http://www.programmersheaven.com/download/
http://en.literateprograms.org/
http://www.cs.utk.edu/~plank/plank/classes/
http://www.go4expert.com/forums/
http://www.cs.bu.edu/teaching/c/