SYLLABUS **BTTI/CURR/ACAD/F3**

DEPARTMENT: <u>BUSINESS & ICT</u> SECTION: <u>ICT</u> DATE: <u>JAN 8, 2014</u>

CLASS: <u>DICT</u> LEVEL: <u>ONE</u> LECTURER: <u>MR. NAMATSI</u>

SUBJECT: <u>STRUCUTURED PROGRAMMING</u>

1. Introduction to structured programming
    - Meaning of structured programming
    - Examples of structured programming languages.
    - History of structured programming languages: Machine, low-level, high-level, 4GLs and 5GLs.
    - Programming paradigms: Unstructured, structured, OOP, Visual and Internet-based Programming.
2. Program development and design
    - Meaning of program development and program design.
    - Describe program development cycle
    - Describe structured programming design concepts: top-down, bottom-up, modular, control flow and monolithic designs.
    - Describing program design tools: algorithms, flowcharts, pseudo codes, structured charts and decision tables
3. Program structure
    - Meaning of program structure
    - Describe the format of structured programming language
    - Describe common operators: operators and order of precedence
    - Describe data type: simple, structure and user-defined.
    - Describe identifiers, expressions and I/O instructions.
4. Program writing
    - Describing the contents of structured programming
    - Describing error handling
    - Write a program in structured language: coding, compiling, debugging, testing, execution, program deployment and error handling
5. Control structures
    - Meaning of control structures
    - Types of control structures: sequence, selection and looping/ Iteration.
6. Data structures
    - Meaning of data structures
    - Types of data structures: strings, lists, arrays, records, pointers, linked lists, queues, stack and trees
    - Implement data structures: strings, lists, arrays, records, and pointers
    - Sorting techniques: bubble, selection, quick, insertion and merge sort
    - Implement bubble sort
    - Searching techniques: sequential, binary and merge search
    - Implement sequential search
7. subprograms
    - Definition of subprograms
    - Types of subprograms
    - Scope of variables: local and global variables
    - Meaning of parameter and parameter passing.
8. File handling
    - Importance of file handling
    - Types of files
    - File organization techniques (sequential, random and indexed)
    - File design
    - File handling operations
9. Program documentation
    - Define program documentation
    - Importance of programming documentation
    - Types of program documentation
    - Writing program documentation
10. Emerging trends of structured programming
    - Identifying emerging trends in structured programming
    - Explaining challenges of emerging trends in structured programming
    - Coping with challenges of the emerging trends

References

1. C How to program,
2. Programming in Pascal
3. World Wide Web

INTRODUCTION TO STRUCTURED PROGRAMMING

DEFINITIONS

1. **Programming**: is the process of developing computer instructions used to solve a particular task using a programming language.
2. **Programming language**: is a set of symbols that can be translated into machine readable form by the computer when arranged in a given syntax.
3. **Syntax**: is a special sequence or order of writing set of characters (symbols) in programming language to perform a given task.
4. **Structured programming**: it is a style of programming that is based on use of control structures, subprograms and indentation to create computer instructions.
5. **Source program**: this is the program code entered by the programmer in programming language editor's window that is yet to be translated into machine readable form.
6. **Object code**: this is the program code that is machine readable form.
7. **Translators**: these are language processors that covert source code program into object code. Translators include: assemblers, interpreters and compilers
8. **Assembler**: this is a language processor that translate assembly language into machine language for a computer to understand and execute.
9. **Interpreter**: it is a language processor that translates the source code program line-by-line allowing the CPU to execute one line before translating the next line. This method of program translation was used by early computers that did not have enough memory to store the object code as a file.
10. **Compiler**: it translates the entire source program into object code file that can be made executable by Linking. The executable can be installed on other computers to perform the task programmed.

Examples of structured programming languages include; C, Pascal, Common Business-Oriented language (COBOL), Formula Translator (FORTRAN), Ada, Beginners' All-purpose Symbolic Instructional Code (BASIC) etc.

HISTORY OF PROGRAMMING LANGUAGES

There are two levels of programming languages, namely; low-level and high-level programming languages.

Low-level Programming Languages

These are languages that require less or no translation for a computer to understand, that is, they are already or almost in machine language. They are hardware-oriented hence not portable (a program written for one computer can not be used on another). They include:

1. *Machine language*: It is also called First Generation language. This is a programming language where program instructions are written using binary digits (bits). Given data and instructions are in binary form, many code lines are needed to accomplish a simple task.
2. *Assembly language*: Also known as Second Generation Languages (2GLs). These are programming languages whose instructions are written using mnemonics. Mnemonics are symbolic operation codes composed of two or three words. For instance

   ```
   MOV AX, 15        //Move 15 to register AX
   SUB AX, 10        //Subtract 10 from the value in AX
   ```

   Assembly languages overcame understanding and use of machine languages since they are readable however, require an assembler to translate them to machine language for a computer to understand.

High-Level Programming Languages

These are languages whose instructions are English-like, can be read and understood even by non-programmers. High-level languages are machine independent and therefore the programmer is more concerned with problem solving rather than how a machine operates. They include: -

1. *Third generation languages (3GLs):* these are also known as structured or procedural languages. They emphasize the breaking of program code into smaller units (modules), control structure and

indentations. These languages include Pascal, COBOL, BASIC, C, FORTRAN and Ada.

2. *Fourth Generation Languages (4GLs):* these are programming languages that present the programmer with graphical tools such as buttons, forms and other tools for easier interface designs. Unlike the 3GLs where the programmer has to come up with all code everything from scratch, 4GLs allow the programmer to select the graphical objects (controls) and use them as design on base form. They include Microsoft Visual Basic, Visual COBOL and Delphi Pascal.
3. *Fifth Generation Languages (5GLs):* these are languages that used in artificial intelligence that enable a computer to depict human like intelligence. These languages include PROLOG, Mercury, LISP and OCCAM.

PROGRAMMING PARADIGMS

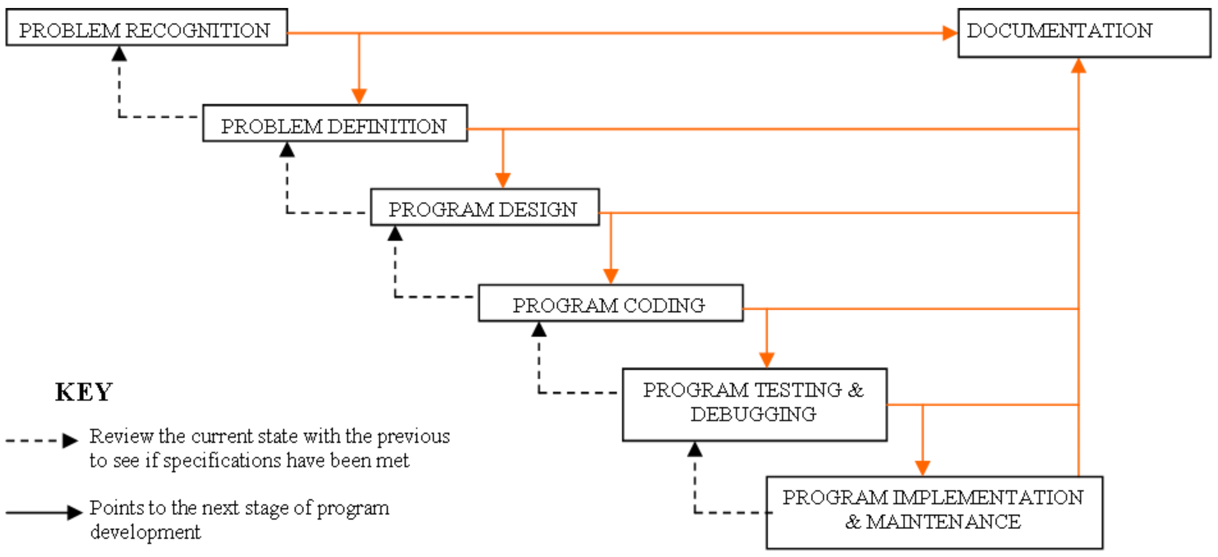
These are ways in which programming languages are classified. A given language is not limited to a given paradigm, for instance, Java supports elements of procedural, object-oriented and event-driven paradigms. The paradigms include:

1. *Unstructured/ monolithic programming:* its is a programming paradigm in which all the code of the program resides in a single large block. These languages do not support splitting of long program into either functions, subroutines or methods that perform a particular task.
2. *Structured programming:* it is a paradigm that emphasize the use of control structures, sub programs and indentation in writing program code. Structured programming does not allow the use of GOTO statement to change the course of program execution. The languages that support this paradigm include, Pascal, C, COBOL, BASIC, Ada etc.
3. *Object-oriented programming*: this is a programming paradigm in which the programmer designs both data structures and operations applied to the data structures. The pairing of data structures and the operations that can be performed on them is known as an object. A program therefore becomes a collection of cooperating objects rather than a list of instructions. OOP languages includes Java, C++, C#, Python etc
4. *Visual programming:* this is a programming paradigm where the programmer creates programs by manipulating program elements graphically rather than textually. Visual programming frees the programmer from having to write code but provide graphical objects to be used to create the interfaces instead. Languages that support this paradigm include: Microsoft Visual Basic, Visual C#, Visual COBOL etc.
5. *Internet-based programming*: this is a paradigm where the programmer creates web-based applications. Since HTML is the basic design languages, to enhance HTML and web applications to support the dynamic we requirements, server-side and client-side programming languages are used. The languages include: Javascript, Java, PHP, Perl, Python, ASP etc

PROGRAM DESIGN AND DEVELOPMENT

PROGRAM DEVELOPMENT LIFECYCLE

These are stages the programmer must follow to come up with a program that solves the indented problem or task. The stages are as illustrated below.


PROBLEM RECOGNITION
DOCUMENTATION
PROBLEM DEFINITION
PROGRAM DESIGN
PROGRAM CODING
PROGRAM TESTING & DEBUGGING
PROGRAM IMPLEMENTATION & MAINTENANCE
KEY
Review the current state with the previous to see if specifications have been met
Points to the next stage of program development

Note: Completion of one stage leads to the start of the next, current stage has to be continuously reviewed to ensure requirements are met. Documentation of each stage's important facts is required.

1. *Problem recognition*: This is the understanding and interpretation of a particular problem. To identify a problem look for keywords such as compute, evaluate, compare, calculate etc and rewrite the problem in a simplified way using the keyword. For instance, a mathematical problem of calculating the area of a circle, is rewritten as $A=\Pi r^2$
2. *Problem definition/Analysis*: This is when programmer defines the likely input, processing activities and expected outcome using keywords outlined at the problem recognition stage. This stage outlines a clear view of what the program should produce, methods that can be used to solve the problem and best alternative chosen. For instance, our problem $A=\Pi r^2$ has,

a.	Input	radius of the circle	r
b.	Process	calculate the area of the circle	Π*r*r
c.	Output	Area of the circle	A

3. *Program design*: This is the stage in which the programmer decides how the various goals of the program are to be achieved. It shows the logical steps a program follows in order to solve a problem. The programmer must analyze specifications and decide whether the program is monolithic or modular. Program design tools such as algorithm, pseudo code, flowchart, structured charts and decision tables.
4. *Program coding*: This is the process of converting program design into its equivalent program using a programming language. This stage results into source code program that can be translated into machine readable form. The program coded should solve the targeted problem.
5. *Program testing and debugging*: when a programmer write programs, it is rare that the code will be perfect. The program has to be tested to ensure it yields expected output, detect errors and debug (correct the error). There are two types of errors (bugs) encountered during testing of a program, namely: -
    a. Syntax errors: These are errors that emanate from improper use of programming language rule such as grammatical mistakes, misuse/ missing punctuation marks, improper naming of variables, misspelling of user-defined and reserved words. Syntax errors must be debugged before running the program.
    b. Logical errors: These errors are also known as runtime errors. These are errors that are not detected by translator but the program gives wrong output or halts during execution.

    Methods used to detect errors in a program include the following: -

a. Desk checking (Dry-run): This is going through a program while its still on paper before entering into programming language editor's window. This help to detect syntax and logical errors.
b. Using debugging utilities: This is the use of translators to detect syntax errors in a program already in programming language editor's window. Most of the programming languages have development IDEs with compiler that detect syntax errors, outline them and allow the programmer to correct them before execution.
c. Using test data: This is carrying out trial runs of new program by entering data variation and extremes (data with errors) to see whether the program halts. A good program should not crash due to wrong data input but inform the user about the anomaly and request for correct data.

6. *Implementation and maintenance*: implementation is the actual delivery and installation of the new program ready for use. Training of staff on use of the program is also done. That implemented program is then reviewed (incase specifications are not met) and maintained.

   Maintenance is an ongoing activity of evaluating implemented program to see whether it is performing as expected. Errors and shortcomings detected are corrected.

PROGRAM DOCUMENTATION

This is the writing of support information explaining how the program can be used, installed, maintained and modified. All stages of program development should be well documented. Documentation can be either: -

a. *Internal documentation*: These are non-executable lines, called comments, in a source program that enable a programmer to understand code statements. Internal documentation is part and parcel of source program code.
b. *External documentation*: These are reference material for users, operators and programmers printed as booklets. They specify user-manuals, installation manuals and programmer manuals.

There are three target groups for any documentation, namely: -

a. *User-oriented documentation*: These are reference material that enable user learn how to use a program with little help from program developer.
b. *Operator-oriented documentation*: These are reference material for technical staff, it provides outline on how to install and maintain the program.
c. *Programmer-oriented documentation*: This is documentation that provides technical information for the future medication of the program. All programs need to modification or review to improve their performance and suit changing environment.

The structure of Documentation

CHAPTER ONE

- Introduction
- Background
- Objectives
- Reasons for the program development
- Methods used to achieve the reasons
- Feasibility study
- Challenges experienced

CHAPTER TWO

- Information gathering methods
- New system requirements

CHAPTER THREE

- System Flowchart
- Dataflow diagram
- Data stores
- Entity Relationship Diagram
- Screen Inputs
- Screen Output

CHAPTER FOUR

- System implementation (screen inputs)
- Screen Output

CHAPTER FIVE

- Program requirements
- Installation
- User Manual
- Recommendations
- Conclusion

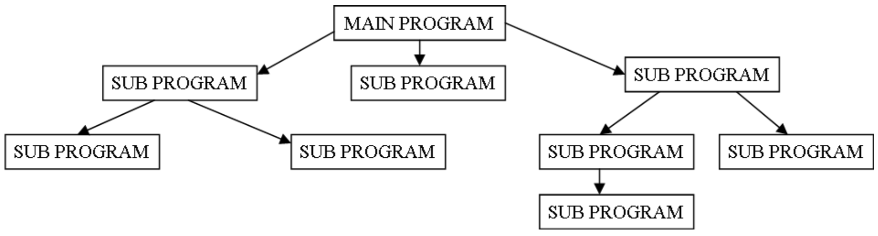
APPENDIX

- Abbreviations
- Sample documents
- Sample screen outputs (reports)
- Sample code

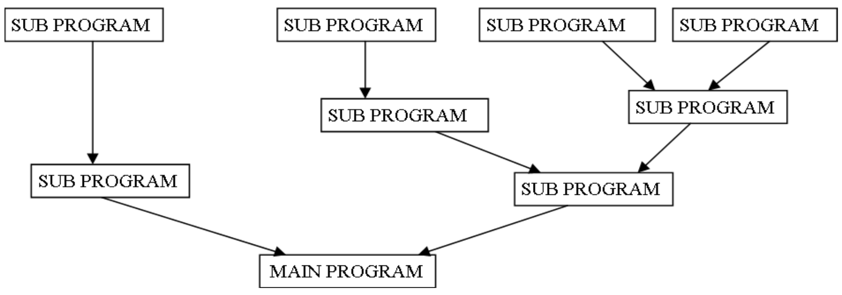
STRUCTURED PROGRAMMING DESIGN METHODS

These are ways of developing a program as series of independent sections for the purpose of performing a task. These ways or methods include: -

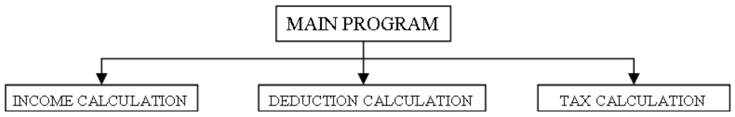
1. Top-Down Design: Also known as step-wise design. This is a programming design, mainly used in procedural languages, in which design begins by specifying complex pieces and then dividing them into successively smaller pieces. This technique emphasize the programmer to write a main procedure that names all major functions it will need. Later, the programming team look at requirements of each of those functions and the process is repeated. When all the various subprograms have been coded, the program is ready for testing.



2. Bottom-Up Design: This is a programming design where individual base elements of a system are first fully specified, coded and tested. These elements/ sub programs are then linked together to form larger subsystems, which in turn are linked until a complete top-level system is formed. OOP is a programming paradigm that uses objects to design applications and computer programs.



3. Modular Design: this is an extension of top-down and bottom-up design techniques that subdivides a system into smaller parts/ modules that are independently written and tested. This approach enables faster and easier debugging and maintenance of programs.



4. Monolithic Design: it is unstructured method of program design where the entire program is planned together as a block. It is wield to program like this though it is common among amateur programmers.

PROGRAM DESIGN TOOLS

Program design shows the logical steps a program follows in order to solve a problem. A poorly design leads to development of a program that does not meet the specifications or indented tasks. Tools used in program design include:

a. Algorithms,

b. Flowcharts,

c. Pseudo codes,
d. Structured charts and
e. Decision tables

ALGORITHM

It is a set of instructions which when correctly executed produces a solution to given problem. For instance, an algorithm to find a word in a dictionary or changing a punctured tyre.

There will always be several algorithms (methods) for solving a problem. It is upon the programmer to make the best choice of the algorithm to use with reasons. The algorithm developed must be easily converted into computer instructions and therefore must have three categories of instructions: -

a. Input instructions: - Supply data to a program
b. Processing instructions: - Manipulates data into information
c. Output instructions: - Gets information out and present it to the user

For instance, an algorithm to compute the area of a circle (A), the algorithm will be,

```
Ask user to enter radius of the circle (r)
Store the value in r
Calculate the area of circle $Πr^2$
Store the value in A
Print the value of A
Stop
```

Once an algorithm is developed, we must check it to ensure that it does the task correctly by hand using appropriate data. This is known as dry running or desk checking. Dry running aims at pin-pointing errors in logic execution before the computer program is written.

PSEUDO CODES

This is a set of statements written in English-like form that express the processing logic of a program. Some words may be drawn from programming languages and mixed with English to form structured statements. Pseudo codes are not executable by a computer.

A good pseudo code must have the following features

a. The statement must be short, clear and readable
b. That statements must not have more than one meaning (not ambiguous)
c. Then statement lines should be clearly outlined and indented.
d. Pseudo code show clearly show START and STOP of executable statements and control structures
e. The input, output and processing statements should clearly be stated using keywords such as PRINT, INPUT, READ etc.

For example, write a pseudo code to computer the area of a circle.

```
START
    PRINT "Enter the radius of a circle"
    READ r
    $A=Πr^2$
    PRINT A
STOP
```

Example 2, write a pseudo code that prompts the user to enter two numbers, calculate their sum and average. Display the two numbers, sum and average on the screen.

```
START
    PRINT "Enter the two numbers"
    INPUT no1, no2
    Sum=no1+no2
```

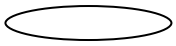
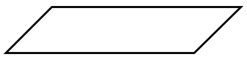
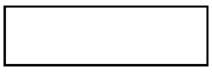
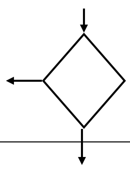
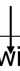
```
        Average=Sum/2
        PRINT no1, no2
        PRINT Sum, Average
STOP.
```

Example 3, write a pseudo code for a program that can be used to classify people according to age. If a person is 18years and above, the program outputs "Adult, Oyee" otherwise it outputs "Young, Uuh".

```
START
        PRINT "Enter your age in years"
        READ age
        IF age≥18 THEN
                Comment="Adult, Oyee"
        ELSE
                Comment="Young, Uuh"
        PRINT comment
STOP
```

FLOWCHART

This is the diagrammatic representation of a program's logical flow of execution using special symbols. The symbols used in program flowchart include;

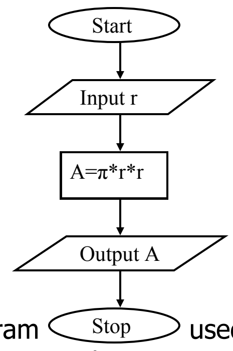
Symbol	Name	Description
	Terminal	It denotes the START or END of program algorithm
	Input/ output	It denotes either INPUT or OUTPUT operations such as read, write, rewrite, delete, print and communication with VDU
	Process	Shows data manipulation or calculations such as move, add, subtract, divide, accumulate, restore, replace, store, sort, merge and manual preparations.
	Decision	Specifies a condition that is evaluated to a Boolean value (True or False) for a program to execute the next instructions basing on the evaluated value.
	On-page Connector	Used to connect/ interface arrows coming from different directions
	arrow	Used to indicate the direction of flow of program logic

When drawing a flowchart;

i. There should be one starting and exit point of program algorithm
ii. Use correct symbol at each stage in program flowchart
iii. The logical flow of program should be clearly shown using arrows.

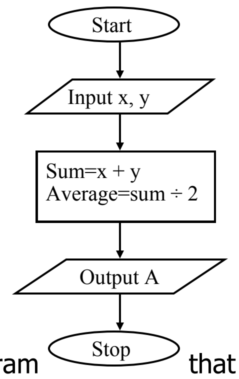
Example 1

Draw a flowchart for a program that calculates the area of a circle whose radius is entered by the user.



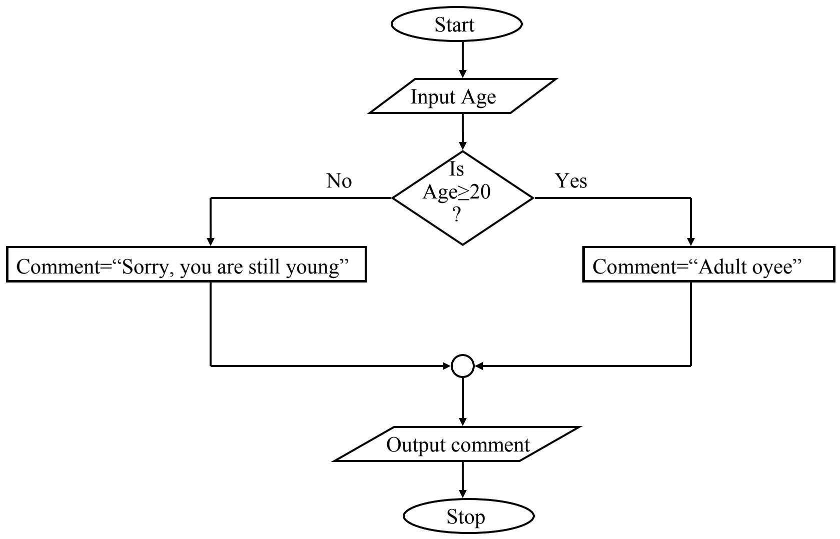
Example 2

Draw a flowchart for a program used to prompt the user to enter two numbers. The program should find and display the sum and average of the two numbers on the screen.



Example 3

Design a flowchart for a program that can be used to classify people according to their age. If a person is 18 years and above, the program outputs "Adult Oyee", otherwise outputs "Sorry, you are still young".

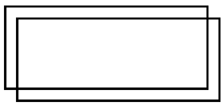
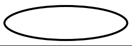
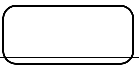
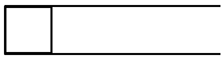
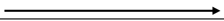


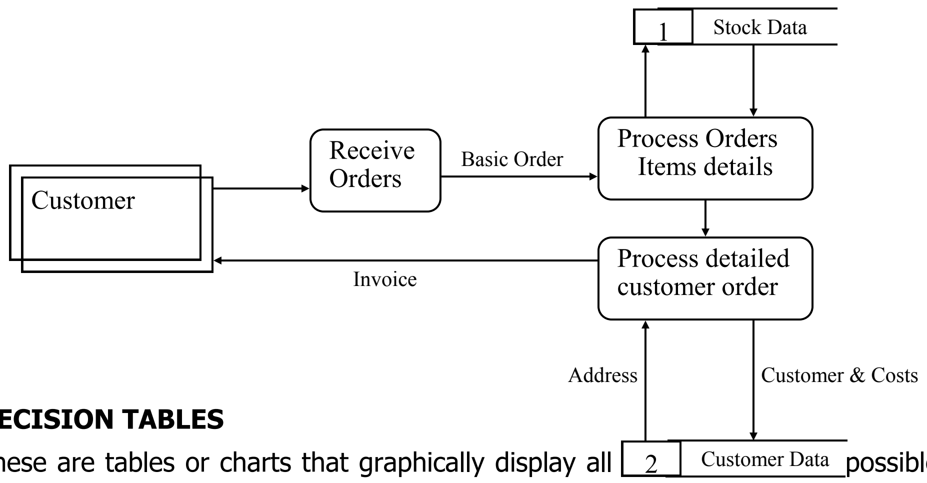
DATA FLOW DIAGRAMS

These are tools that provide an effective form in which to represent the movement of data through a system and associated transformations of data resulting from various processing actions.

The symbols used in drawing data flow diagrams include:

For example,

Symbol	**Description**
	Source/ destination of data which is external to the source/ destination system such as customer or supplier.
 **Or** 	Process or the flow of data is transformed. For instance, checking credit status
	Data flow. Any stored data with no reference to the physical method of storage
	Data flow direction



DECISION TABLES

These are tables or charts that graphically display all possible conditions (decisions) and related actions (outcomes) in a form of matrix. Their purpose is to provide a solution to a problem graphically. A decision table consists of four main parts, namely:

- Condition stub: It lists all possible conditions present in a system
- Condition entry: Shows all possible combinations of conditions after testing
- Action stub: Lists all actions that may be taken to meet specific condition
- Action entry: Lists all specific actions that should be followed for each combination of conditions.

PROGRAMMING IN PASCAL

When talking about computer languages, there are basically three major terms that will be used.

1. **Machine language** -- actual binary code that gives basic instructions to the computer's CPU. These are usually very simple commands like adding two numbers or moving data from one memory location to another.
2. **Assembly language** -- a way for humans to program computers directly without memorizing strings of binary numbers. There is a one-to-one correspondence with machine code. For example, in Intel x86 machine language, ADD and MOV are mnemonics for the addition and move operations.
3. **High-level language --** permits humans to write complex programs without going step-by step. High -level languages include Pascal, C, C++, FORTRAN, Java, BASIC, and many more. One command in a high-level language, like writing a string to a file, may translate to dozens or even hundreds of machine language instructions.

All computers can only run machine language programs directly. Assembly language programs are assembled, or translated into machine language. Likewise, programs written in high-level languages, like Pascal, must also be translated into machine language before they can be run. The technical terminology for this operation is compiling.
The program that accomplishes the translation is called a compiler. This program is rather complex since it not only creates machine language instructions from lines of code, but often also optimizes the code to run faster, adds error-correction code, and links the code with subroutines stored elsewhere. For example, when you tell the computer to print something to the screen, the compiler translates this as a call to a pre-written module. Your code must then be linked to the code that the compiler manufacturer provides before an executable program results.
With high-level languages, there are again three terms to remember:

1. **Source code** -- the code that you write. This typically has an extension that indicates the language used. For example, Pascal source code usually ends in ".PAS" and C++ code usually ends in ".CPP"
2. **Object code --** the result of compiling. Object code usually includes only one module of a program, and cannot be run yet since it is incomplete. On DOS/Windows systems, this usually has an extension of ".OBJ"
3. **Executable code** -- the end result. All the object code modules necessary for a program to function are linked together. On DOS/Windows systems, this usually has an extension of ".EXE"

Program Structure

The basic structure of a Pascal program is:

```
PROGRAM ProgramName (Input, Output);
  CONST
     (* Constant declarations *)
  TYPE
     (* Type declarations *)
  VAR
     (* Variable declarations *)
     (* Subprogram definitions *)
  BEGIN
     (* Executable statements *)
  END.
```

The elements of a program must be in the correct order, though some may be omitted if not needed. Here's a program that does nothing, but has all the REQUIRED elements:

```
program DoNothing;
begin
end.
```

Pascal comments start with a (* and end with a *). You can't nest comments:

```
(* (* *) *)
```

will yield an error because the compiler matches the first (* with the first *), ignoring everything in between. The second *) is left without its matching (*.
In Turbo Pascal, {Comment} is an alternative to (* Comment *). The opening brace signifies the beginning of a block of comments, and the ending brace signifies the end of a block of comments.
Commenting has two purposes: first, it makes your code easier to understand. If you write your code without comments, you may come back to it a year later and have a lot of difficulty figuring out what you've done or why you did it that way. Another use of commenting is to figure out errors in your program.

When you don't know what is causing an error in your code, you can comment out any suspect code segments. Remember the earlier restriction on nesting comments? It just so happens that braces {} supersede parenthesis-stars (* *). You will NOT get an error if you do this:

```
{ (* Comment *) }
```

So, if you have Turbo Pascal, I suggest using standard comments (* *), leaving the braces for debugging.

All spaces and end-of-lines are ignored by the Pascal compiler unless they are inside a string. However, to make your program readable by human beings, you should indent your statements and put separate statements on separate lines.

Identifiers

Identifiers are names that allow you to reference stored values, such as variables and constants. Also, every program must be identified (get it?) by an identifier.

Rules for identifiers:

- Must begin with a letter from the English alphabet.
- Can be followed by alphanumeric characters (alphabetic characters and numerals), or the underscore (_).
- May not contain special characters, such as:~ ! @ # $ % ^ & * ( ) _ + ` - = { } [ ] : " ; ' < > ? , . / | \

Several identifiers are reserved in Pascal -- you cannot use them as your own identifiers. They are:

and	array	begin	case	const	div	do	downto
else	end	file	for	forward	function	goto	if
in	label	mod	nil	not	of	or	packed
procedure	program	record	repeat	set	then	to	type
until	var	while	with				

Also, Pascal has several pre-defined identifiers. You can replace them with your own definitions, but then you'd be deleting part of the functionality of Pascal.

abs	arctan	boolean	char	cos	dispose	eof	eoln
exp	false	input	integer	ln	maxint	new	odd
ord	output	pack	page	pred	read	readln	real
reset	rewrite	round	sin	sqr	sqrt	succ	text
true	trunc	write	writeln				

Pascal is not case sensitive! MyProgram, MYPROGRAM, and mYpRoGrAm are equivalent. But for readability purposes, it is a good idea to use meaningful capitalization!

Identifiers can be any length, but many Pascal compilers will only look at the first 32 characters or so. That is,

```
ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFAlphaBeta
ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGammaDelta
```

may be equivalent to some Pascal compilers because the differences begin in the 33rd character.

To make your code compilable by all compilers, use a reasonable length for identifiers -- up to 15 characters.

Variables and Data Types

Variables are similar to constants, but their values can be changed as the program runs. Unlike in BASIC and other loosely-typed languages, variables must be declared in Pascal before they can be used:

```
var
     IdentifierList1 : DataType1;
     IdentifierList2 : DataType2;
     IdentifierList3 : DataType3;
  ...
```

IdentifierList is a series of identifiers, separated by commas (,). All identifiers in the list are declared as being of the same data type.

The main data types in Pascal are:

1. **integer** data type can contain integers from -32768 to 32767.
2. The **real** data type has a positive range from 3.4x10-38 to 3.4x1038. Real values can be written in either fixed-point notation or in scientific notation, with the character E separating the mantissa from the exponent. Thus, 452.13 is the same as 4.5213e2
3. The **char** data type holds characters. Be sure to enclose them in single quotes, or apostrophes: 'a', 'B', '+'. This data type can also hold system characters, such as the null character (ordinal value of 0) and the false-space character (ordinal value of 255).
4. The **Boolean** data type can have only two values: TRUE and FALSE
5. The String data type holds a group of characters enclosed in single quotation mark such as 'DICT', 'My names are' etc.

An example of declaring several variables is:

```
var
     age, year, grade : integer;
     circumference : real;
     LetterGrade : char;
     DidYouFail : Boolean;
```

Assignment and Operations

Once you have declared a variable, you can store values in it. This is called assignment. To assign a value to a variable, follow this syntax:

```
variable_name := expression;
```

Note that unlike other languages, whose assignment operator is simply an equals sign, Pascal uses a colon followed by an equals sign. The expression can either be a single value:

```
some_real := 385.385837;
```

or it can be an arithmetic sequence:

```
some_real := 37573.5 * 37593 + 385.8 / 367.1;
```

The arithmetic operators in Pascal are:

Operator	Operation	Operands	Result
+	Addition or unary positive	real or integer	real or integer
-	Subtraction or unary negative	real or integer	real or integer
*	Multiplication	real or integer	real or integer
/	Real division	real or integer	real
div	Integer division	integer	integer
mod	Modulus (remainder division)	integer	integer

div and **mod** only work on integers. / works on both real and integers but will always yield a real answer. The other operations work on both real and integers.

For operators that accept both real and integers, the resulting data type will be integer only if all the operands are integer. It will be real if any of the operands are real.

Therefore,

```
3857 + 68348 * 38 div 56834
```

will be integer, but

```
38573 div 34739 mod 372 + 35730 - 38834 + 1.1
```

will be real because 1.1 is a real value.

Each variable can only be assigned a value that is of the same data type. Thus, you cannot assign a real value to an integer variable. However, certain data types are compatible with others. In these cases, you can assign a value of a lower data type to a variable of a higher data type. This is most often done when assigning integer values to real variables. Suppose you had this variable declaration section:

```
var
     some_int : integer;
     some_real : real;
```

When the following block of statements executes,

```
some_int := 375;
some_real := some_int;
```

some_real will have a value of 375.0, or 3.75e2.

In Pascal, the minus sign can be used to make a value negative. The plus sign can also be used to make a value positive. This, however, is unnecessary because values default to being positive.
Do not attempt to use two operators side by side!

```
some_real := 37.5 * -2;
```

This may make perfect sense to you, since you're trying to multiply by negative-2. However, Pascal will be confused -- it won't know whether to multiply or subtract. You can avoid this by using parentheses:

```
some_real := 37.5 * (-2);
```

to make it clearer.
The computer follows an order of operations similar to the one that you follow when you do arithmetic:

```
* / div mod
+ -
```

The computer looks at each expression according to these rules:

1. Evaluate all expressions in parentheses, starting from the innermost set of parentheses and proceeding to the outermost.
2. Evaluate all multiplication and division from left to right.
3. Evaluate all addition and subtraction from left to right.

The value of

```
3.5 * (2 + 3)
```

will be 17.5.
Pascal cannot perform standard arithmetic operations on Booleans. There is a special set of Boolean operations. Also, you should not perform standard operations on characters because the results may vary from compiler to compiler.

Punctuation and Indentation

Since Pascal ignores end-of-lines and spaces, punctuation is needed to tell the compiler when a statement ends.
You MUST have a semicolon following:

- the program heading
- each constant definition
- each variable declaration
- each type definition (to be discussed later)
- almost all statements

The last statement in the program, the one immediately preceding the END, does not require a semicolon. However, it's harmless to add one, and it saves you from having to add a semicolon if suddenly you had to move the statement higher up.

Indenting is not required. However, it is of great use for the programmer, since it helps to make the program clearer. If you wanted to, you could have a program look like this:

```
program Stupid; const a=5; b=385.3; var alpha,beta:real; begin alpha := a + b; beta:= b / a end.
```

But it's much better for it to look like this:

```
program Stupid;
  const
     a = 5;
     b = 385.3;

  var
     alpha,
     beta : real;

  begin     (* main *)
     alpha := a + b;
     beta := b / a
  end.      (* main *)
```

In general, indent two to four spaces for each block. Skip a line between blocks (such as between the const and var blocks).
Most importantly, use comments liberally! If you ever return to a program that you wrote ten years ago, you probably wouldn't remember the logic unless you documented it. It is a good idea to comment the main executable part of the program, to distinguish it from subprograms.

Input

Input means to read data into memory, either from the keyboard, the mouse, or a file on disk.
We will not get into mouse input in detail, because that syntax differs from machine to machine, and may require a complicated interrupt call to the mouse driver. If you would like to see the source code for a mouse support unit for DOS, click here. This unit will not work under Windows, Macintosh, or X-Windows, because these operating systems handle all mouse input for you and do not let you interface with the mouse directly.
The basic format for reading in data is:

```
read (Variable_List);
```

Variable_List is a series of variable identifiers separated by commas.
read, however, does not go to the next line. This can be a problem with character input, because the end-of-line character is read as a space.
To read data and then go on to the next line, use

```
readln (Variable_List);
```

Suppose you had this input from the user, and a, b, c, and d were all integers.

```
45 97 3
1 2 3
```

This would be the result of various statements:

Statement(s)	a	b	c	d
`read (a);`	45	97		
`readln (a);`	45	1		
`read (a, b, c, d);`	45	97	3	1
`readln (a, b);`	45	97	1	2

You see, the read statement does not skip to the next line unless necessary, whereas the readln statement is just a read statement that skips to the next line at the end of reading.
When reading in integers, all spaces are skipped until a numeral is found. Then all subsequent numberals are read, until a non-numeric character is reached (including, but not limited to, a space).

```
8352.38
```

When an integer is read, its value becomes 8352. If, immediately afterwards, you read in a character, the value would be '.' Suppose you tried to read in two integers. That would not work, because when the computer looks for data to fill the second variable, it sees the '.' and stops, saying, "I couldn't find any data to read."
With real values, the computer also skips spaces and then reads as much as can be read. However, there is one restriction: a real that has no whole part must begin with 0.

```
.678
```

is invalid, and the computer can't read in a real, but

```
0.678
```

is fine.
Make sure that all identifiers in the argument list refer to variables! Constants cannot be assigned a value, and neither can literal values.

Output

For writing data to the screen, there are also two statements:

```
write (Argument_List);
writeln (Argument_List);
```

The writeln statement skips to the next line when done. You can use strings in the argument list, either constants or literal values. If you want to display an apostrophe within a string, use two consecutive apostrophes.
Formatting output is quite easy. For each identifier or literal value on the argument list, use:

```
Value : field_width
```

The output is right-justified in a field of the specified integer width. If the width is not long enough for the data, the width specification will be ignored and the data will be displayed in its entirety (except for real values -- see below).
Suppose we had:

```
   write ('Hi':10, 5:4, 5673:2);
```
The output (with a dash simulating the space) would be:
```
   --------Hi---55673
```
For real values, you can use the aforementioned syntax to display scientific notation in a specified field width, or you can convert to fixed notation with:
```
   Value : field_width : decimal_field_width
```
The field width is the total field width, including the decimal part. The whole number part is always displayed fully, so if you have not allocated enough space, it will be displayed anyway. However, if the number of decimal digits exceeds the specified decimal field width, the output will be rounded to the specified number of places (though the variable itself is not changed).
So
```
   write (573549.56792:20:2);
```
would look like:
```
      -----------573549.57
```

Boolean Expressions

Boolean expressions are used to compare two values. The simplest form of Boolean expression looks like this:
```
   value1 relational_operator value2
```
The following relational operators are used:

<	less than
>	greater than
=	equal to
<=	less than or equal to
>=	greater than or equal to
<>	not equal to

You can assign Boolean expressions to Boolean variables:
```
   some_bool := 3 < 5;
```
Of course, the value of some_bool becomes TRUE. Complex Boolean expressions are formed by using the Boolean operators:

not	negation (~)
and	conjunction (^)
or	disjunction (v)
xor	exclusive-OR

NOT is a unary operator - it is applied to only one value and inverts it:
```
   NOT true = false
   NOT false = true
```
AND yields TRUE only if both expressions are TRUE.
```
   TRUE and FALSE = FALSE      TRUE and TRUE = TRUE
```
OR yields TRUE if either expression is TRUE, or if both are. The following are TRUE:
```
   TRUE or TRUE
   TRUE or FALSE
   FALSE or TRUE
```
XOR yields TRUE if one expression is TRUE and the other is FALSE. Thus,
```
   TRUE or TRUE = FALSE
   TRUE or FALSE = TRUE
   FALSE or TRUE = TRUE
   FALSE or FALSE = FALSE
```
When combining two Boolean expressions using relational and Boolean operators, be careful to use parentheses.
```
      (3>5) or (650<1)
```

This is because the Boolean operators are higher on the order of operations than the relational operators:

```
   not
   * / div mod and
   + - or
   < > <= >= = <>
```

This way,

```
   3 > 5 or 650 < 1
```

becomes evaluated as

```
  3 > (5 or 650) < 1
```

which makes no sense, because the Boolean operator or only works on Boolean values, not on integers.
The Boolean operators (AND, OR, NOT, XOR) can be used on Boolean variables just as easily as they are used on Boolean expressions.
Whenever possible, don't compare two real values with the equals sign. Small round-off errors may cause two equivalent expressions to differ.

CONTROL STRUCTURES

A control structure is a block of programming that analyses variables and chooses a direction in which to go based on a given flow control. Flow control determines how a computer will respond when given certain conditions and parameters. Control structures are grouped into three, namely:

- Sequential control structures
- Selection control structures
- Repetition/ looping/ Iteration control structures

Sequential Control Structures

This is the default mode or programming. It is line by line execution of program code statements form start to the end.

Selection Control Structures

It is a group of control structures used for decisions and branching, that is, choosing between two or more alternative paths (or block of program statements) to be executed. They include:

- IF .. THEN
- IF .. THEN .. ELSE
- Nested IF
- CASE … OF

IF .. THEN

The IF statement is a simple control that tests whether a condition is true or false, if the condition is true, then an action occurs or given statements are executed otherwise, if the condition is false, nothing is done. The one-way branch format is:

```
   IF BooleanExpression THEN
        StatementIfTrue;
```

If the Boolean expression evaluates to true, the statement executes. Otherwise, it is skipped. The IF statement accepts only one statement. If you would like to branch to a compound statement, you must use a begin-end to enclose the statements:

```
   IF BooleanExpression THEN
        BEGIN
           Statement1;
           Statement2
        END;
```

For example, write a program to capture English, Kiswahili, Mathematics and Science marks scored by a student in an exam. The program comments “Excellent Work!!!” when the student’s average is 60 and above.

```
PROGRAM commenting(input, output);
VAR eng,kis,mat,sci:Integer;
       Avg:real;
BEGIN
       Writeln;writeln;
       Write(‘Enter English, Kiswahili, Mathematics and Science’);
       Readln(eng,kis,mat,sci);
       Avg:=(eng+kis+mat+sci)/3;
       If (Avg>=60) then
             Writeln(‘Excellent Work!!!’);
       Readln;
END;
```

IF .. THEN .. ELSE

This is a two way selection control structure, where a condition is tested and if true then an action occurs, and when false an alternate action is taken:

```
   IF BooleanExpression THEN
        StatementIfTrue
     ELSE
        StatementIfFalse;
```

If the Boolean expression evaluates to FALSE, the statement following the else will be performed. Note that you may NOT use a semicolon after the statement preceding the else. That causes the computer to treat it as a one-way selection, leaving it to wonder where the else came from.

For example, write a program to capture English, Kiswahili, Mathematics and Science marks scored by a student in an exam. The program comments "Excellent Work!!!" when the student's average is 60 and above otherwise it comments "Try again!!!'.

```
PROGRAM commenting(input, output);
VAR eng,kis,mat,sci:Integer;
       Avg:real;
BEGIN
       Writeln;writeln;
       Write('Enter English, Kiswahili, Mathematics and Science');
       Readln(eng,kis,mat,sci);
       Avg:=(eng+kis+mat+sci)/3;
       If (Avg>=60) then
              Writeln('Excellent Work!!!')
       Else
              Writeln('Try again!!!');
       Readln;
END;
```

NESTED IF

This is a multi-way selection, simply nest if statements, that combin ELSE's with other IF's allowing several tests to be made. In an IF -THEN-ELSEIF -THEN-ELSEIF -THEN structure, tests will stop as soon as a condition is true, that is, you'd probably want to put the most likely test first for efficiency :

```
   IF Condition1 THEN
        Statement1
   ELSEIF Condition2 THEN
        Statement2
   ELSE
        Statement3;
```

Or incase there are more than one statement in the construct then use:

```
    IF Condition1 THEN
   BEGIN
      Statement1;
      Statement2;
   END
   ELSEIF Condition2 THEN
   BEGIN
         Statement3;
         Statement4;
   END
   ELSE BEGIN
         Statement5;
         Statement6;
   END;
```

For example, write a program to capture English, Kiswahili, Mathematics and Science marks scored by a student in an exam. The program remarks as follows based on student's mean score: -

Mean range	Remarks
80-100	Excellent Work. Distinction
70-79	Very Good Work. Distinction
60-69	Good, Keep Up. Credit
40-59	Aim Higher. Pass
0-39	Try Again next exam. Fail

```
PROGRAM commenting(input, output);
VAR eng,kis,mat,sci:Integer;
       Avg:real;
       Comment:string;
BEGIN
       Writeln;writeln;
```

```
        Write('Enter English, Kiswahili, Mathematics and Science');
        Readln(eng,kis,mat,sci);
        Avg:=(eng+kis+mat+sci)/3;
        If ((Avg>=60) AND (AVG<=100)) then
               Comment:='Excellent Work. Distinction'
        ElseIf ((Avg>=70) AND (AVG<=79)) then
               Comment:='Very Good Work. Distinction'
        ElseIf ((Avg>=60) AND (AVG<=69)) then
               Comment:='Good, Keep Up. Credit'
        ElseIf ((Avg>=40) AND (AVG<=59)) then
               Comment:='Aim Higher. Pass'
        Else
               Comment:='Try Again next exam. Fail';
        Writeln;
        Writeln(comment);
        Readln;
END;
```

Exercise

Write a program that prompts the user to enter two numbers and operator of the process to be performed. Basing on the operator, the program process and outputs the answer on the screen. Otherwise, if the operator is invalid (not either /,+,- or *) the program dismisses with error message.

CASE .. OF

It is the alternative to nested IF control structure and preferred in some cases to reduce unnecessary program code. For instance, suppose you wanted to branch one way if b is 1, 7, 2037, or 5; and another way if otherwise. You could do it by:

```
  if (b = 1) or (b = 7) or (b = 2037) or (b = 5) then
       Statement1
  Elseif (b>1) and (b<7) then
        Statement2
  else
       Statement3;
```

But in this case, it would be simpler to list the numbers for which you want Statement1 to execute. You would do this with a case statement:

```
  case b of
       1,7,2037,5: Statement1;
       2..6: Statement2;
       otherwise   Statement3
    end;
```

The general form of the case statement is:

```
    case selector of
         List1:    Statement1;
         List2:    Statement2;
         ...
         Listn:    Statementn;
         otherwise Statement
      end;
```

Or for block of statements

```
        case selector of
         List1: BEGIN
               Statement1;
               Statement2;
         END;
         List2: BEGIN
               Statement3;
               Statement4;
         END;
         ...
         ...
         otherwise BEGIN
               Statementn;
               Statementn+1;
         END
        END;
```

The otherwise part is optional and can be replaced with else as shown below.

```
      case selector of
```

```
        List1:    Statement1;
        List2:    Statement2;
        ...
        Listn:    Statementn;
     else Statement
     end;
```

Example 1

Write a Pascal program that calculates and displays membership fee based on the age of the member as follows.

AGE IN YEARS	FEES
1-12	15
13-24	50
Above 25	150

```
Program Membership_Fee(input,output);
Uses crt;
Var age, fee: integer;
Begin
       Clrscr;
       Write('Enter your age in years: ');
       Readln(age);
       CASE age OF
              1..12: fee:=15;
              13..24: fee:=50;
       ELSE fee:=150;
       END;
       Writeln;
       Writeln('Your fees is ',fee);
       Readln;
END.
```

Example 2

Write a program that allows the user to enter a Module 1 examination score as an integer in the range 0-100. the program convert and display the score in grade suing the following ranges otherwise it outputs "Error, wrong marks".

Score	Grade
0-29	E. Failure
30-39	D. Referred
40-49	C. Pass
50-59	B. Credit
60-100	A. Distinction.

```
Program grading(input,output);
   Uses crt;
   Var score:integer;
       Grd:string;
   BEGIN
          Clrscr;
          Write('Enter module 1 examination score ');
          Readln(score);
          Case score OF
                 0..29:grd:='E. Failure';
                 30..39:grd:='D. Referred';
                 40..49:grd:='C. Pass';
50..59:grd:='B. Credit';
                 60..100:grd:='A. Distinction';
          Else grd:='Error, wrong marks';
          End;
          Writeln(grd);
          Readln;
   END.
```

Exercise

1. Write a program which allows the user to input a number in the range 1-10. if the user has entered a number in the range, output the number in words to the screen. If they enter a number out of the range, display an error message telling them off.

Goto Statement

A goto statement in Pascal provides an unconditional jump from the goto statement to a labeled statement in the same function.

NOTE: Use of goto statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and modify. Any program that uses a goto statement can be rewritten so that is doesn't need the goto.

The syntax is as follows:
```
      goto label;
            …..
            …..
      label: statement;
```
Here, label must be an unsigned integer label, whose value can be from 1 to 9999.
For example.
```
      Program exGoto(input, output);
      Label 1;
      Var
            a:integer;
      Begin
            a:=10;
            1:
            Repeat
                  If (a=15) then
                  begin
                        a:=a+1;
                        Goto 1;
                  End;
                  Writeln('Values of a: ', a);
                  a:=a+1;
            Until (a=20);
            Readln;
      End.
```
Please not that:

1. In Pascal all labels must be declared before constant and variable declarations
2. The if and goto statements may be used in the compound statement to transfer control out of the compound statement, but it is illegal to transfer control into a compound statement

Looping Control Structures

These are group of control structures that cause a block of program statements to be executed multiple times. A loop is a section of code that repeats itself. A loop index or loop counter is an integer variable that is used to keep track of how many times a loop has been executed. A loop limit is a variable or constant that is integer valued, which determines the number of times a loop will execute, or a maximum value of the loop index to be reached at the loop termination. They include:

1. FOR..DO
2. WHILE .. DO
3. REPEAT .. UNTIL

FOR .. DO
Looping means repeating a statement or compound statement over and over for specific number of times. In Pascal, the fixed repetition loop is the for loop. The general form is:
```
   FOR loop-index := initial-value to final-value DO
      statement;
```
Where `loop-index` is the counter to be incremented
`Initial-value` is the beginning value of the loop index
`Final-value` is the ending value of the loop index.

For example
Write a program that outputs number 1-20 on vertically on the screen.
```
      Program Numbers_1_20(output);
      Uses crt;
      Var counter:integer;
      Begin
            Clrscr;
            For counter:=1 to 20 do
                  Writeln(counter);
            Readln;
      End.
```
The index variable must be of an ordinal data type. You can use the index in calculations within the body of the loop, but you should not change the value of the index. An example of using the index is a program

that sums all whole numbers from 1 to 100:

```
        Program sum_1_100(output);
        Uses crt;
        Var count,sum:integer;
        Begin
                sum := 0;
                for count := 1 to 100 do
                        sum := sum + count;
                Writeln('The sum of whole numbers from 1 to 100 is ',sum);
                Readln;
        End.
```

In the for-to-do loop, the starting value MUST be lower than the ending value, or the loop will never execute! If you want to count down, you should use the for-downto-do loop:

```
    for index := StartingHigh downto EndingLow do
      statement;
```

For example, a program that outputs numbers from 1 to 20 in descending order

```
        Program Numbers_20_1(output);
        Uses crt;
        Var counter:integer;
        Begin
                Clrscr;
                For counter:=20 downto 1 do
                        Writeln(counter);
                Readln;
        End.
```

Exercise

Write a program that display even numbers between 1 and 30 horizontally on the screen.

WHILE..DO

It is a conditional loop that is initiated and program statement(s) continues to be executed until a certain condition is met/ satisfied. The statements may or never be executed at all basing on the condition set. The pretest loop has the following format:

```
        loop_initialization;
        while BooleanExpression do
          statement;
```

The loop continues to execute until the `Booleanexpression` becomes FALSE. In the body of the loop, you must somehow affect the Boolean expression by changing one of the variables used in it. Otherwise, an infinite loop will result:

```
        a := 5;
        while a < 6 do
          writeln (a);
```

Remedy this situation by changing the variable's value:

```
     a := 5;
      while a < 6 do
         begin
            writeln (a);
            a := a + 1
         end;
```

The WHILE ... DO lop is called a pretest loop because the condition is tested before the body of the loop executes. So if the condition starts out as FALSE, the body of the while loop never executes.

For example a program that displays number from 10 to 30 on horizontally on the screen.

```
        Program nos_10_30(output);
        CONST MAX=30;
        VAR index:integer;
        BEGIN
                Index:=10; {Initialize first number to be displayed}
                WHILE (index<=30) DO BEGIN
                        Write(index:5);
                        Index:=index+1; {increment index to next value in the series}
                End;
                Readln;
        END.
```

REPEAT..UNTIL

It is a posttest loop, that is, it is similar to WHILE .. DO loop except that a condition is tested at the end of block of statements, therefore statements are executed at least once. It has the following format:

```
loopinitilization;
repeat
              statement1;
              statement2
until BooleanExpression;
```

In a repeat loop, compound statements are built-in -- you don't need to use BEGIN-END. Also, the loop continues until the Boolean expression is TRUE, whereas the while loop continues until the Boolean expression is FALSE.

This loop is called a posttest loop because the condition is tested AFTER the body of the loop executes. The REPEAT loop is useful when you want the loop to execute at least once, no matter what the starting value of the Boolean expression is.

For example a program that displays number from 10 to 30 on horizontally on the screen.

```
Program nos_10_30(output);
CONST MAX=30;
VAR index:integer;
BEGIN
       Index:=10; {initialize first number to be displayed}
       REPEAT
              Write(index:5);
              Index:=index+1; {increment index to next value in the series}
       UNTIL (index>30); {repeat until the condition is true}
       Readln;
END.
```

CONSTANTS

These are values that which do not change during programs execution. An example would be the value of PI, 3.141592654. In a program required to calculate the circumference of several circles, it would be simpler to write the word PI, instead of its value 3.141592654.

To declare constants, the keyword CONST is used, followed by the name of the constant, an equals sign, the constants value, and semi-colon, eg,

```
CONST PI=3.141592654;
```

For example, a program to compute the area of a circle would be,

```
Program CIRCUMFERENCE (Input, Output);
Const PI=3.141592654;
Var Circumfer, Diameter: Real;
Begin
      Writeln('Enter the diameter of the circle');
      Readln(Diameter);
      Circumfer:=PI*Diameter;
      Writeln('The circumference of the circle is ',Circumf:6:2);
End.
```

Exercise, write a program which prompts the user to input ordinary time and overtime worked, calculating the gross pay. The rate is 4.20 per hour, and overtime is two times and half the rate.

SUB-PROGRAMS

Pascal is a structured programming language, this means that a problem is divided into sub-problems and then coded in a structure. These sub-problems are known as subprograms. A subprogram is a program unit/module that performs a particular task. These subprograms are combined to form larger programs. A subprogram can be invoked by a subprogram/ program, which is called the calling program.

Pascal provides two kinds of subprograms:

- Functions: these subprograms return a single value.
- Procedures: these subprograms do not return a value directly.

PROCEDURES

These are subprograms that, instead of returning a single value, allow to obtain a group of results. They help the programmer to avoid repetitions. A procedure start off with a **begin** and ends up with an **end**; . Procedure definition is as shown below:

```
procedure name(argument(s): type1, argument(s): type 2, ... );
   < local declarations >
begin
   < procedure body >
end;
```

A *procedure definition* in Pascal consists of a header , local declarations and a body of the procedure. The procedure header consists of the keyword procedure and a name given to the procedure. Here are all the parts of a procedure:

- **Arguments**: The argument(s) establish the linkage between the calling program and the procedure identifiers and also called the formal parameters. Rules for arguments in procedures are same as that for the functions.
- **Local declarations**: local declarations refer to the declarations for labels, constants, variables, functions and procedures, which are application to the body of the procedure only.

- **Procedure Body**: The procedure body contains a collection of statements that define what the procedure does. It should always be enclosed between the reserved words begin and end. It is the part of a procedure where all computations are done.

Procedure Declarations:

A procedure declaration tells the compiler about a procedure name and how to call the procedure. The actual body of the procedure can be defined separately. A procedure declaration has the following syntax:

```
procedure name(argument(s): type1, argument(s): type 2, ... );
```

Please note that the name of the procedure is not associated with any type. For the above defined procedure findMin(), following is the declaration:

```
procedure findMin(x, y, z: integer; var m: integer);
```

To have an exact know the importance of a procedure, you should compare a program which includes a repeated section with another program avoiding the repeated sections by using a procedure, which is called several times:

```
Program Lesson7_Program(input,output);
Uses crt;
Var Counter : Integer;
Begin
 textcolor(green);
 GotoXy(10,5);
 For Counter := 1 to 10 do
 Begin
       write(chr(196));
 End;
 GotoXy(10,6);
 For Counter := 1 to 10 do
  Begin
   write(chr(196));
  End;
 GotoXy(10,7);
 For Counter := 1 to 10 do
  Begin
   write(chr(196));
  End;
 GotoXy(10,10);
 For Counter := 1 to 10 do
  Begin
   write(chr(196));
  End;
 Readkey;
End.
```

Now have a look at the next program which uses a procedure:

```
Program Lesson7_Program(input,output);
Uses crt;
Procedure DrawLine;
Var Counter : Integer;
Begin
 textcolor(green);
 For Counter := 1 to 10 do
  Begin
   write(chr(196));
  End;
End;
Begin
 GotoXy(10,5);
 DrawLine;
 GotoXy(10,6);
 DrawLine;
 GotoXy(10,7);
 DrawLine;
 GotoXy(10,10);
 DrawLine;
 Readkey;
End.
```

There are some differences between these two programs which are very important to note. These are :

1. **Size of the program.** It is very important for a program to be small in size. The first program, say, its size is 1900 bytes, but the second one holds about 1350 bytes!
2. **Neatness.** Adopting a neat style of writing for a program helps the programmer (and other future debuggers) to cater with future bugs. I think that the first program is cumbersome, whilst the other is not! What do you think??!
3. **Repetitions.** Repetitions in a program can cause a hard time for a programmer. So procedures are an essential way to avoid repetitions in a program. They also enlarge the size of a program!
4. **Debugging Efficiency.** When you are required to debug the program, bugs could be much more easier to find out as the program is sliced into smaller chunks. You may run the program and notice a mistake at a certain point and which is located in a particular procedure/function. It would be much more difficult to find a mistake in a program if it would be one whole piece of code. Do slice your program into smaller chunks, and this needs design of the whole problem in hand prior to coding.

<u>Calling a Procedure:</u>

Procedures are placed ABOVE the main program's BEGIN … END. When a program calls a procedure, program control is transferred to the called procedure. A called procedure performs the defined task and when its last end statement is reached, it returns the control back to the calling program.

To call a procedure you simply need to pass the required parameters along with the procedure name as shown below:

Example 1

Write a program that uses a procedure to display the message "Welcome to Pascal Procedures" on the standard output device.

```
Program using_procudure(output);
Procedure hello;
Begin
      Writeln("Welcome to Pascal Procedures");
End;
BEGIN
      Hello; {calling the HELLO procedure to display the message}
End.
```

Example 2

Design a program that uses a procedure to display the messages "Hello World" and "Good Bye" using a procedure.

```
Program simple_procedure(output);
      Procedure say(message:string); {defining a procedure called say
      Begin
            Writeln(message);
      End;
Var m:string;
Begin
      m:='Hello world';
      Say(m); { procedure call to display the message}
      m:='Good bye';
      Say(m); {procedure call to display the second message}
      Readln;
End.
```

Example 3

Write a program that prompts the user to enter three numbers, the program compares the numbers and display the largest. Use Procedure to compare numbers.

```
program exProcedure(Input,Output);
var
   a, b, c,  min: integer;
procedure findMin(x, y, z: integer; var m: integer);
(* Finds the minimum of the 3 values, variable m is passed by reference *)
begin
   if (x < y AND x < z ) then m:= x
   elseif (y < x AND y < z ) then  m:= y
   else m:= z;
end; { end of procedure findMin }
begin
   writeln(' Enter three numbers: ');
   readln( a, b, c);
   findMin(a, b, c, min); (* Procedure call *)
   writeln(' Minimum: ', min);
end.
```

Example 4

Using a procedure, write a program that computers and displays the square of a number entered by the user.

```
Program VAR_PARAM_EXAMPLE(Input, Output);
  Procedure Square(Index : Integer; Var Result : Integer);
  Begin
    Result := Index * Index;
  End;
Var
  Res : Integer;
Begin
 Writeln('The square of 5 is: ');
 Square(5, Res);
 Writeln(Res);
End.
```

VARIABLE SCOPE

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable cannot be accessed. There are three places where variables can be declared in Pascal programming language:

- Inside a subprogram or a block which is called **local** variables
- Outside of all subprograms which is called **global** variable
- In the definition of subprogram parameters which is called **formal** parameters

Local Variables

Variables that are declared inside a subprogram or block are called local variables. They can be used only by statements that are inside that subprogram or block of code. Local variables are not known to subpro-grams outside their own. Following is the example using local variables. Here all the variablesa, b and c are local to program named exLocal.

```
program exLocal(Input, Output);
var
   a, b, c: integer;
begin
  (* actual initialization *)
   a := 10;
   b := 20;
   c := a + b;
   writeln('value of a = ', a , ' b =  ',  b, ' and c = ', c);
end.
```

Now, let us extend the program little more, let us create a procedure named display which will have its own set of variables a, b and c and display their values, right from the program exLocal.

```
program exLocal(Input, Output);
var
   a, b, c: integer;
procedure display;
var
   a, b, c: integer;
begin
    (* local variables *)
   a := 10;
   b := 20;
   c := a + b;
   writeln('Winthin the procedure display');
   writeln('value of a = ', a , ' b =  ',  b, ' and c = ', c);
end;
begin
   a:= 100;
   b:= 200;
   c:= a + b;
   writeln('Within the program exlocal');
   writeln('value of a = ', a , ' b =  ',  b, ' and c = ', c);
   display();
end.
```

Global Variables

Global variables are defined outside of a function, usually on top of the program. The global variables will hold their value throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. Following is an example using global and local variables:

```
program exGlobal(Input, Output);
var
   a, b, c: integer;
procedure display;
var
   x, y, z: integer;
begin
    (* local variables *)
   x := 10;
   y := 20;
   z := x + y;
  (*global variables *)
   a := 30;
   b:= 40;
   c:= a + b;
   writeln('Winthin the procedure display');
   writeln(' Displaying the global variables a, b, and c');
   writeln('value of a = ', a , ' b =  ',  b, ' and c = ', c);
   writeln('Displaying the local variables x, y, and z');
   writeln('value of x = ', x , ' y =  ',  y, ' and z = ', z);
end;
begin
   a:= 100;
   b:= 200;
   c:= 300;
   writeln('Winthin the program exlocal');
   writeln('value of a = ', a , ' b =  ',  b, ' and c = ', c);
   display();
end.
```

Please note that the procedure display has access to the variables a, b and c, which are global variables with respect to display as well as its own local variables. A program can have same name for local and global variables but value of local variable inside a function will take preference.

Let us change the previous example a little, now the local variables for the procedure display has same names as a, b, c:

```
program exGlobal(Input, Output);
var
   a, b, c: integer;
procedure display;
var
   a, b, c: integer;
begin
   (* local variables *)
   a := 10;
   b := 20;
   c := a + b;
   writeln('Winthin the procedure display');
   writeln(' Displaying the global variables a, b, and c');
   writeln('value of a = ', a , ' b =  ',  b, ' and c = ', c);
   writeln('Displaying the local variables a, b, and c');
   writeln('value of a = ', a , ' b =  ',  b, ' and c = ', c);
end;
begin
   a:= 100;
   b:= 200;
   c:= 300;
   writeln('Winthin the program exlocal');
   writeln('value of a = ', a , ' b =  ',  b, ' and c = ', c);
   display();
end.
```

Call Type	Description
Call by value	This method copies the actual value of an argument into the formal parameter of the subprogram. In this case, changes made to the parameter inside the subprogram have no effect on the argument.
Call by reference	This method copies the address of an argument into the formal parameter. Inside the subprogram, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

FUNCTIONS

This is a sub program that return a value at the end of execution. A function start and end in a similar way to that of a procedure. If more than one value is required to be returned by a module, you should make use of the variable parameter. A function can have parameters too. If you don't need to return any values, a procedure is more best. However, if a value should be returned after the module is executed, function should be used instead.

There are two types of function in Pascal, namely: -

1. In-built functions: -Pascal standard library provides numerous built-in functions that your program can call.
2. User-defined functions: They are defined and called within a program by the programmer.

Pascal In-Built Functions

Conversion Functions

The following functions convert one type of data to another.

1. round(x). This converts a real expression x into an integer value by rounding it up or down to the nearest integer. Therefore, round(3.2) has the value 3 while round(3.7) has the value 4. round(-3.2) has the value -3 while round(-3.7) has the value -4.

2. trunc(x). This converts a real expression x into an integer value by truncating the fractional part. Therefore the expressions trunc(3.2) and trunc(3.7) both have the value 3. trunc(-3.2) and trunc(-3.7) both have the value -3.
3. chr(i). This converts an integer expression i into a char value. The integer is converted to whatever character has the same ASCII representation. This means that the integer must lie inside the values for the ASCII character set, in this case 0 to 255.
4. Frac(x). This returns a real number whose value is the fractional part of the real value x. For example, n := Frac(7.3); will cause n to have the value 0.3.
5. Int(x). This returns a real number whose value is the integer part of the real value x. For example, n := Int(8.3); will cause n to have the value 8.0.

Mathematical Functions

Pascal supports most of the basic mathematical functions.

1. abs(x). returns the absolute value of x. x may be real or integer, and the result will have the same type as the argument.
2. odd(x). returns true if x is an odd integer, and false if x is even. x must be an integer.
3. sqr(x). returns the square of x. x may be real or integer, and the result will have the same type as the argument.
4. sqrt(x). returns the square root of x. x may be real or integer. The result is always real.
5. sin(x). returns the sine of x. x may be real or integer, and gives an angle in radians. The result is always real.
6. cos(x). returns the cosine of x. x may be real or integer, and gives an angle in radians. The result is always real.
7. arctan(x). returns the arctangent of x. x may be real or integer. The result is always real, and gives an angle in radians.
8. ln(x). returns the natural logarithm of x (base e). x may be real or integer. The result is always real.
9. exp(x). returns the number e to the power x. x may be real or integer. The result is always real.

The functions abs and sqr are non-standard, in that they may return either an integer or real number. For this reason, they may not be passed as function arguments to subprograms.

Order Functions

Order functions can be applied to any scalar type. The ``successor'' function succ has the form

next := succ(value);

where value is any scalar value. The result returned by succ is the next value in the scalar type. For example, if value is an integer, succ returns the next integer, so

succ(3) = 4

If value is an enumerated type, succ returns the next value in the enumerated list. Thus if you define

```
type days = (sun,mon,tue,wed,thu,fri,sat);
```

you would have

```
succ(sun) = mon
succ(mon) = tue
```

If you try

```
succ(sat)
```

you will be given an error message. sat is the last element in the enumerated type and has no successor.

The “predecessor” function pred is the converse of succ. It has the form

```
previous := pred(value);
```

pred returns whatever comes before the given value. For example,

```
succ(3) = 2
succ(tue) = mon
```

If value has no predecessor, Alice will give you an error.

String Manipulation Routines

String manipulation routines manipulate string constants and variables with string types. Our descriptions of these routines will give examples of source code using each routine, followed by output from this source code.

1. Delete(str, pos, num); This procedure deletes a substring of length num starting at position pos in the string variable str. The remaining characters (if any) in the string are shifted left. This routine is similar to the Watcom StrDelete routine.
2. Insert(src, dest, pos); This procedure inserts the string src into the string dest at position pos. The variables src and dest are both of type string; pos is of type integer. An error message will be issued if the pos is past the end of the string, or the string becomes too long. This is stricter error checking than that provided by the same routine in the Turbo Pascal compiler. This routine is similar to the Watcom StrInsert routine.
3. Copy(str, pos, num); This function returns a substring of the string variable str of length num starting at position pos. It returns the null string (i.e. a string of length zero) if pos is greater than the length of the string.
4. Length(str). This function returns the length of the string argument in characters. For a string constant, this is the number of characters in the constant. For a string variable, it is the number of characters currently stored in the variable. Length gets its value from the special length byte kept at the zeroth index of every string. It is similar to the Watcom Strlen function.
5. Pos(srch, str); This function returns the position in the string str at which the string srch is found. It returns 0 if srch was not found in str. For example,

   i := Pos('exec', 'autoexec.bat');

   will cause i to have the value 5.
6. Str(val, str). This procedure puts the value of the given variable val (which may be integer or real) into the given string variable str as an ASCII representation of the number. The variable val can actually have format specifiers of the same type found in write procedure calls. One can specify a field width for integers and reals. Reals may also have a precision field, just as with write. In general

   ```
   Str(i:n, dest);
   writeln(dest);
   is the same as:
   writeln(i : n);
   ```

   1 if dest is a string large enough to hold the written number. Note that one places a space in front of integer operands if no field width is provided. A field width of zero eliminates this. This space is not output when Borland deviations (+b) are enabled. An error occurs if the string is not large enough to hold the number.
7. Val(str, variable, code); This procedure is the inverse of the str procedure; it examines the string contained in str and converts it to a number which it stores in var. The type of the number is either real or

integer, depending on the type of var. Leading and trailing spaces are not permitted; the valid formats for the number are the same as those for constants in Alice Pascal. In addition, real numbers with nothing before the decimal point (e.g. ".1") are acceptable.

If the conversion is successful, the integer variable code is set to zero; otherwise, it contains an index into str of the character on which the conversion failed. If this is the case, the value of var is undefined. For example,

```
Val('7.8', n, stat);
```

will set n to 7.8 and stat to zero, while

```
Val('7y2', n, stat);
```

will result in stat being 2 (the index of the letter y in the string) and the value of n will be undefined.

8. UpCase(c); This function returns a character which is the upper-case equivalent of the character c. For example,

```
ch := UpCase('b');
```

will result in ch having the value 'B'. This routine is handy if you want to accept user input in either upper or lower case.

File and Directory Handling Routines

1. ChDir(path); This routine changes the DOS current directory to the specified path. If the string includes a leading drive specifier, the current drive is changed as well. For example,

```
ChDir('c:\data');
```

will set the current drive to be C:, and the current directory to be the \data directory on that drive.

If you change directories, any relative pathnames (i.e. pathnames that are not completely specified from the root directory down) will no longer be valid. In particular, this applies to the save file name. This routine is contained in TURBOLIB.AP.

2. Erase(filename); This procedure erases the specified file from the disk. It is important that you do not erase a file that is currently active (that is, opened using reset or rewrite or append). This routine is contained in TURBOLIB.AP.

```
GetDir(drive, stringvar);
```

This procedure stores into the specified string variable the current directory for the specified drive. The drive is an integer, 1 for A:, 2 for B: and so on. If drive is 0, the current drive is assumed. For example,

```
GetDir(3, currdir);
```

will store into the string variable currdir the current directory for drive C:. A typical directory string would be '\data\first'. If the current directory for the specified drive is the root directory, the string variable will contain the null string (i.e. StrLen(currdir) will be zero).

This routine is contained in TURBOLIB.AP.

```
MkDir(path);
```

This procedure creates a new directory on the disk, whose name is contained in the string argument path. This new directory is always created on the current drive. For more information on directories, see the DOS manual.

3. RmDir(path); This procedure removes (i.e., erases) a directory from the disk. You cannot remove a directory that contains files or other directories. Refer to the DOS manuals for more information about directory handling.

```
Rename(oldpath, newname);
```

This routine changes the name of a file or directory on disk. oldpath specifies the name (and optional path) of the file or directory to be renamed, and newname contains the new name.

User-Defined Functions (UDF)

Are sub programs defined by a programmer that is called to return a value when called. A function is defined using the function keyword. The general form of a function definition is as follows:

```
function name(argument(s): type1; argument(s): type2; ...): function_type;
local declarations;
begin
   ...
   < statements >
   ...
   name:= expression;
end;
```

A function definition in Pascal consists of a function header, local declarations and a function body. The function header consists of the keyword function and a name given to the function. Here are all the parts of a function:

- **Arguments**: The argument(s) establish the linkage between the calling program and the function identifiers and also called the formal parameters. A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Use of such formal parameters is optional. These parameters may have standard data type, user-defined data type or sub range data type.
- The formal **parameters** list appearing in the function statement could be simple or subscripted variables, arrays or structured variables, or subprograms.
- **Return Type**: All functions must return a value, so all functions must be assigned a type. The function -type is the data type of the value the function returns. It may be standard, user-defined scalar or sub range type but it cannot be structured type.
- **Local declarations**: local declarations refer to the declarations for labels, constants, variables, functions and procedures, which are application to the body of function only.
- **Function Body**: The function body contains a collection of statements that define what the function does. It should always be enclosed between the reserved words begin and end. It is the part of a function where all computations are done. There must be an assignment statement of the type - name := expression; in the function body that assigns a value to the function name. This value is returned as and when the function is executed. The last statement in the body must be an end statement.

Following is an example showing how to define a function in Pascal:

```
(* function returning the max between two numbers *)
function max(num1, num2: integer): integer;
var
   (* local variable declaration *)
   result: integer;
begin
   if (num1 > num2) then
      result := num1
   else
      result := num2;
   max := result;
end;
```

Calling a Function:

While creating a function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, program control is transferred to the called function. A called function performs defined task and when its return statement is executed or when it last end statement is reached, it returns program control back to the main program.

To call a function you simply need to pass the required parameters along with function name and if function returns a value then you can store returned value. Following is a simple example to show the usage:

```
program exFunction(Input, Output);
var
   a, b, ret : integer;
(*function definition *)
function max(num1, num2: integer): integer;
var
   (* local variable declaration *)
   result: integer;
begin
   if (num1 > num2) then
      result := num1
   else
      result := num2;
   max := result;
end;
begin
   a := 100;
   b := 200;
  (* calling a function to get max value *)
   ret := max(a, b);
   writeln( 'Max value is : ', ret );
end.
```

Recursive Subprograms

We have seen that a program or subprogram may call another subprogram. When a subprogram calls itself, it is referred to as a recursive call and the process is known as recursion.

To illustrate the concept, let us calculate the factorial of a number. Factorial of a number n is defined as:

```
n! = n*(n-1)!
   = n*(n-1)*(n-2)!
      ...
   = n*(n-1)*(n-2)*(n-3)... 1
```

The following program calculates the factorial of a given number by calling itself recursively.

```
program exRecursion(Input, Output);
var
   num, f: integer;
function fact(x: integer): integer; (* calculates factorial of x - x! *)
begin
   if x=0 then
      fact := 1
   else
      fact := x * fact(x-1); (* recursive call *)
end; { end of function fact}
begin
   writeln(' Enter a number: ');
   readln(num);
   f := fact(num);
   writeln(' Factorial ', num, ' is: ' , f);
end.
```

Example 2: Following is another example which generates the Fibonacci Series for a given number using a recursive function:

```
program recursiveFibonacci(Input, Output);
var
   i: integer;
function fibonacci(n: integer): integer;
begin
   if n=1 then
      fibonacci := 0
   else if n=2 then
      fibonacci := 1
   else
      fibonacci := fibonacci(n-1) + fibonacci(n-2);
end;
begin
   for i:= 1 to 10 do
   write(fibonacci (i), '  ');
end.
```

Arguments of a Subprogram:

If a subprogram (function or procedure) is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the subprogram.

The formal parameters behave like other local variables inside the subprogram and are created upon entry into the subprogram and destroyed upon exit.

While calling a subprogram, there are two ways that arguments can be passed to the subprogram:

By default, Pascal uses call by value to pass arguments. In general, this means that code within a subprogram cannot alter the arguments used to call the subprogram. The example program we used in the chapter 'Pascal - Functions' called the function named max() using call by value.

DATA STRUCTURES

A data structure is a scheme (format) for storing and organizing related data in a computer so that it can be used efficiently. They provide a means to manage large amounts of data efficiently.

The basic types of data structures include the following:

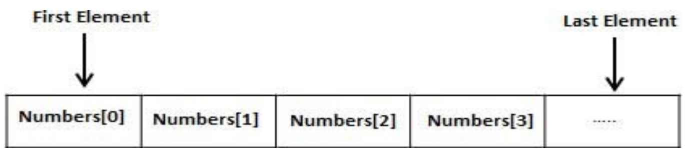
a) Strings,
b) Arrays,
c) Records,
d) Files
e) Lists,
f) Pointers,
g) Linked lists,
h) Queues,
i) Stack and
j) Trees

Arrays

An array is a data structure which can store a fixed-size sequential collection of elements of the same type. It is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number1, number2, ..., and number100, you declare one array variable such as numbers and use numbers[1], numbers[2], and ..., numbers[100] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element. Please note that if you want a C style array starting from index 0, you just need to start the index from 0, instead of 1.



Declaring Arrays

To declare an array in Pascal, a programmer may either declare the type and then create variables of that array or directly declare the array variable.

The general form of type declaration of one dimensional array is:

```
type
   array-identifier = array[index-type] of element-type;
```

Where,

- array-identifier indicates the name of the array type.
- index-type specifies the subscript of the array; it can be any scalar data type except real
- element-type specifies the types of values that are going to be stored

For example,

```
type
   vector = array [ 1..25] of real;
var
   velocity: vector;
```

Now velocity is a variable array of vector type, which is sufficient to hold up to 25 real numbers. To start the array from 0 index, the declaration would be:

```
type
   vector = array [ 0..24] of real;
var
   velocity: vector;
```

Types of Array Subscript

In Pascal, an array subscript could be of any scalar type like, integer, Boolean, enumerated or subrange, except real. Array subscripts could have negative values too.

For example,

```
type
   temperature = array [-10 .. 50] of real;
var
   day_temp, night_temp: temperature;
```

Let us take up another example where the subscript is of character type:

```
type
   ch_array = array[char] of 1..26;
var
   alphabet: ch_array;
```

Subscript could be of enumerated type:

```
type
   color = ( red, black, blue, silver, beige);
   car_color = array of [color] of boolean;
var
   car_body: car_color;
```

Initializing Arrays

In Pascal, arrays are initialized through assignment, either by specifying a particular subscript or using a for -do loop.

For example:

```
type
   ch_array = array[char] of 1..26;
var
   alphabet: ch_array;
   c: char;
begin
   ...
   for c:= 'A' to 'Z' do
   alphabet[c] := ord[m];
   (* the ord() function returns the ordinal values *)
```

Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:

```
a: integer;
a: = alphabet['A'];
```

The above statement will take the first element from the array named alphabet and assign the value to the variable a.

Following is an example which will use all the above mentioned three concepts viz. declaration, assignment and accessing arrays:

```
program exArrays;
var
   n: array [1..10] of integer;   (* n is an array of 10 integers *)
   i, j: integer;
begin
   (* initialize elements of array n to 0 *)
   for i := 1 to 10 do
       n[ i ] := i + 100;   (* set element at location i to i + 100 *)
    (* output each array element's value *)
   for j:= 1 to 10 do
      writeln('Element[', j, '] = ', n[j] );
end.
```

Records

This is a type of variable that allows the programmer to combine data items of different data types (kinds). Records consist of different fields. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book:

- Title
- Author
- Subject
- Book ID

Defining a Record

To define a record type, you may use the type declaration statement. The record type is defined as:

```
Type record-name = record
   field-1: field-type1;
   field-2: field-type2;
   ...
   field-n: field-typen;
end;
```

Here is the way you would declare the Book record:

```
type Books = record
   title: packed array [1..50] of char;
   author: packed array [1..50] of char;
   subject: packed array [1..100] of char;
   book_id: integer;
end;
```

The record variables are defined in the usual way as

```
var  r1, r2, ... : record-name;
```

Alternatively, you can directly define a record type variable as:

```
Var Books : record
   title: packed array [1..50] of char;
   author: packed array [1..50] of char;
   subject: packed array [1..100] of char;
   book_id: integer;
end;
```

Accessing Fields of a Record

To access any field of a record, we use the member access operator (.). The member access operator is coded as a period between the record variable name and the field that we wish to access. Following is the example to explain usage of structure:

```
program exRecords;
Type Books = record
   title: packed array [1..50] of char;
   author: packed array [1..50] of char;
   subject: packed array [1..100] of char;
   book_id: longint;
end;
var
   Book1, Book2: Books; (* Declare Book1 and Book2 of type Books *)
begin
   (* book 1 specification *)
   Book1.title  := 'C Programming';
   Book1.author := 'Nuha Ali ';
   Book1.subject := 'C Programming Tutorial';
   Book1.book_id := 6495407;

   (* book 2 specification *)
   Book2.title := 'Telecom Billing';
   Book2.author := 'Zara Ali';
   Book2.subject := 'Telecom Billing Tutorial';
   Book2.book_id := 6495700;

   (* print Book1 info *)
   writeln ('Book 1 title : ', Book1.title);
   writeln('Book 1 author : ', Book1.author);
   writeln( 'Book 1 subject : ', Book1.subject);
   writeln( 'Book 1 book_id : ', Book1.book_id);
   writeln;

   (* print Book2 info *)
   writeln ('Book 2 title : ', Book2.title);
   writeln('Book 2 author : ', Book2.author);
   writeln( 'Book 2 subject : ', Book2.subject);
   writeln( 'Book 2 book_id : ', Book2.book_id);
end.
```

Records as Subprogram Arguments

You can pass a record as a subprogram argument in very similar way as you pass any other variable or pointer. You would access the record fields in the similar way as you have accessed in the above example:

```
program exRecords;
Type Books = record
   title: packed array [1..50] of char;
   author: packed array [1..50] of char;
   subject: packed array [1..100] of char;
   book_id: longint;
end;
var
   Book1, Book2: Books; (* Declare Book1 and Book2 of type Books *)
(* procedure declaration *)
procedure printBook( var book: Books );
begin
   (* print Book info *)
   writeln ('Book  title : ', book.title);
   writeln('Book  author : ', book.author);
   writeln( 'Book  subject : ', book.subject);
   writeln( 'Book book_id : ', book.book_id);
end;
begin
   (* book 1 specification *)
   Book1.title  := 'C Programming';
   Book1.author := 'Nuha Ali ';
   Book1.subject := 'C Programming Tutorial';
   Book1.book_id := 6495407;
  (* book 2 specification *)
   Book2.title := 'Telecom Billing';
   Book2.author := 'Zara Ali';
   Book2.subject := 'Telecom Billing Tutorial';
   Book2.book_id := 6495700;
  (* print Book1 info *)
   printbook(Book1);
   writeln;
   (* print Book2 info *)
   printbook(Book2);
end.
```

The With Statement

We have discussed that the members of a record can be accessed using the member access operator (.). This way the name of the record variable has to be written every time. The With statement provides an alternative way to do that.

Look at the following code snippet taken from our first example:

```
(* book 1 specification *)
   Book1.title  := 'C Programming';
   Book1.author := 'Nuha Ali ';
   Book1.subject := 'C Programming Tutorial';
   Book1.book_id := 6495407;
```

The same assignment could be written using the With statement as:

```
(* book 1 specification *)
With Book1 do
begin
   title  := 'C Programming';
   author := 'Nuha Ali ';
   subject := 'C Programming Tutorial';
   book_id := 6495407;
end;
```

FILES

A file contains data which is saved in the hard disk. File operations includes open, read, write, and close. In order to understand file structures, it helps to think of a disk drive in terms of a drawer in a filing cabinet.
In each drawer, there are many files, which are usually contained in manila folders. In order to view, create, or modify the contents of a folder, one must first ***retrieve*** then ***open*** the folder. This is similar to ***initializing*** and ***opening*** a computer disk file.
If one wants to view the file contents, then one must ***read*** the file, which holds for either physical or computer files. Similarly, creating new file contents or modifying existing file information is accomplished by ***writing*** to the file.
After one has completed operations on a given file, it is returned to the file cabinet, to keep the work area neat (this helps one find the file when it is next needed). A similar situation holds for computer disk files, where one ***closes*** the file in order to deallocate file pointers assigned by the file I/O library and runtime module.

PASCAL File I/O Commands.

The PASCAL language provides constructs for allocating or initializing, opening, reading, writing, and closing files. File addresses or references are expressed in terms of symbolic file handles, which are represented in PASCAL as names assigned to a given file. The following commands pertain:

ASSIGN: The ASSIGN statement provides a mechanism for linking a disk file to a symbolic name or file handle. Its syntax:
`ASSIGN( file-handle , file-pathname ) ;` , where `file-handle` is a name declared as type text in a VAR statement and `file-pathname` is a DOS pathname of the file to be referenced by file-handle.

Example:
```
VAR handle : text ;
      :
ASSIGN(handle,'A:/PROJECTS/PROJ-2.DAT');
```

Notes: In the preceding example, the string constant 'A:/PROJECTS/PROJ-2.DAT' may be replaced by a string variable that contains the pathname. Also, the file referenced by this path must be a text file, as noted in the VAR statement that precedes the ASSIGN statement.

There are two types of file opening statements, one of which opens the file for reading (input), the other for writing (output).

RESET: The RESET statement opens a file for reading. Its syntax is
`RESET (file-handle) ;` , where `file-handle` was associated with a disk file by the ASSIGN statement.

Example:
```
RESET (handle);
```

REWRITE: The REWRITE statement opens a file for writing. Its syntax is
`REWRITE (file-handle) ;` , where `file-handle` was associated with a disk file by the ASSIGN statement.

Example:
REWRITE (handle);
PASCAL has two ways of reading or writing to a file. The READLN (or WRITELN) statement reads a sequence of characters terminated by a newline character or a carriage return, whereas the READ (WRITE) statement reads the file as a stream of characters. Since we have already covered the READLN and WRITELN statements in class, which we used to obtain input from (send output to) the computer keyboard (monitor), we herein discuss the READ and WRITE statements only. The syntax of READLN and WRITELN is symmetric to that of the READ and WRITE statements.
READ: The READ statement inputs data from the keyboard or a file as a stream of characters. Its syntax is
`READ ([file-handle], [I/O-list]) ;` , where `file-handle` was associated with a

disk file by the ASSIGN statement and `I/O-list` is a list of variables with optional format specifiers that reference data contained in the file that is itself referenced by file-handle.

Example:

```
READ (handle, a[1], letter, x, y);
```

Notes: If the file handle is omitted, then the Turbo PASCAL runtime module understands that the input is being taken from the computer keyboard.

WRITE: The WRITE statement inputs data from the keyboard or a file as a stream of characters. Its syntax is,
`WRITE ([file-handle], [I/O-list]) ;` , where `file-handle` was associated with a disk file by the ASSIGN statement and `I/O-list` is a list of variables with optional format specifiers that reference data contained in the file that is itself referenced by file-handle.

Example:

```
WRITE (handle, a[1], letter, x, y);
```

Notes: If the file handle is omitted, then the Turbo PASCAL runtime module understands that the output is to be directed toward the computer monitor.

CLOSE: The CLOSE statement deallocates the file handle that was activated with the ASSIGN statement, and closes the file on disk. The syntax is,
`CLOSE ([file-handle]) ;` , where `file-handle` was associated with a disk file by the ASSIGN statement

Example:

```
CLOSE (handle);
```

Read from a File (file input)

Reading a file in Pascal is very easy. Note that there are no reserved words in Pascal that are used to to read or write to a file. We used the 'usual words': `readln()` and `writeln();` Here's the technique of how to read a **text** file (only):

```
Program Open_file(input,output);
Var UserFile : Text;
    FileName, TFile : String;
Begin
        Writeln('Enter the file name (with its full path) of the text file:');
        readln(FileName);
        Assign(UserFile, FileName + '.txt'); {Assign the .txt file to a text variable}
        Reset(UserFile); {'Reset(x)' - means open the file x}
        Repeat
                Readln(UserFile,TFile);
                Writeln(TFile);
        Until Eof(UserFile);
        Close(UserFile);
        Readln;
End.
```

It is worth to take a look at various important lines of the program.

1. A new variable of type: '`Text`' is new to you, and this should be used whenever you are going to edit a **text** file! The variable 'FileName' is required to link to the file indicated by the user.
2. The '`assign`' statement is used to declare 'FileName' + '.txt' to a text file, so that the file could be opened, using - '`Reset`()'. To read from the first line to the very last line of the file, you should use the repeat-until loop, ending the loop with : '`Until Eof(textfile);`', which means: 'Until the **E**nd **O**f **F**ile [eof]'.

Create and Write to a File (file output)

The following program is an example of how to create-and-write or overwrite a file:

```
Program Create_file(input,output);
Var   Txt : String[20];
```

```
            UserFile:Text;
    Begin
             Assign(UserFile,'C:\fpc\TextFile.txt'); {assign a text file}
             Rewrite(UserFile); {open the file 'fname' for writing}
             Writeln(UserFile,'PASCAL PROGRAMMING');
             Writeln(UserFile,'If you did not understand something,');
             Writeln(UserFile,'Please send me an email to:');
             Writeln(UserFile,'shanams81@gmail.com');
             Writeln('Write some text to the file:');
             Readln(Txt);
             Writeln(UserFile,'');
             Writeln(UserFile,'The user entered this text:');
             Writeln(UserFile,Txt);
             Close(UserFile);
    End.
```

In the above program, I am using the 'writeln()' statement so that I write to the file I have previously assigned to. Note that, since I am using writeln(), there is **no** output to the screen, it goes to the file I initialized.

To check exactly what has just been written to this file, go to C:\, and see if there is a file named: **Textfile.txt**. Open it and see what does it contain!!.

Append text to an existing File

Writing to an existing file, means, open a file and add extra data, but not overwrite the file. Some beginner programmers do not actually understand how to, not overwrite a file with data they would like to input. This is the common problem, to open an existing file and add contents:

```
    Program Append_file(input, output);
    Var     UFile : Text;
            Dat:string[30];
    Begin
            Assign(UFile,'C:\FPC\ADDTEXT.TXT');
            Append(UFile);
            Writeln(UFile,'How many sentences, are present in this file?');
            Write('Enter what else to be added to the file: ');
            Readln(Dat);
            Writeln(UFile,Dat); {Appends what user entered to end of file}
            Close(UFile);
    End.
```

The reserved word is `'append(f)'` is used to open a file for addition of extra data, where `f` is a variable of type text. This can be done by simply change the `'Rewrite(UFile)'` to `'append(UFile)'`, and the text file is not overwritten, but appended!.

Delete Files

In Pascal, the reserved word used to delete files from the hard this is the 'Erase(f)' where f is a variable of any data type. This means that 'f' could be both file and text variable type. Note, that the file you are about to delete is not taken in the recycle bin, but it is directly kicked off the hard disk!!

Unlike any other file functions, the *`erase()`* function does not open the file to delete it, so you don't need to apply a `'close(...)'` after `erase()`.

Example Program:

```
    Program Delete_file;
    Var UFile : Text; { or it could be of 'file' type}
    Begin
            Assign(UFile,'C:\FPC\ADDTEXT.TXT');
            Erase (UFile);
    End.
```

Binary Files

There are two kinds of binary files :
1. Typed files
2. Untyped files

Typed file means that the file has a uniform format throughout its contents. This kind of file includes databases, because all of them contains the data records. Simply said, file of records.
Untyped file means that the file contains no uniform data. Although you may have seen records stored in this kind of file, that file contains some additional information that may be different record structure. Simply said, file with no distinct records.
First, we discuss typed files. Suppose you define a record like this :

```
Type  Temployee = record
                  name    : string[20];
                  address : string[40];
                  phone   : string[15];
                  age     : byte;
                  salary  : longint;
end;
```

Typed file of THAT record is defined like this :

```
Var F : file of Temployee;
```

The steps of using typed file is just the same as using text file.
1. You associate it with file name using `assign`.
2. Open it, using `reset`, OR Create it, using `rewrite`.
3. Use it. `Writeln` in text file MUST BE CHANGED into `Write` and `Readln` with `Read` respectively.
4. Close it using `close`.

All error handling and `IOResult` use is all the same, so that I don't have to re-mention it all over again.
The difference is : If you open typed file with `reset` it doesn't mean that you can only read it (just in the text files), but you may write on it and modify it. The command `rewrite` is still the same, create a new one, discarding the file previous contents. Then, look at this example :

```
PROGRAM binary_file_write(input,output);
uses crt;
Type Temployee = record
           name    : string[20];
           address : string[40];
           phone   : string[15];
           age     : byte;
           salary  : longint;
end;
var
   F : file of Temployee;
   c : char;
   r : Temployee;
   s : string;
begin
   clrscr;
   write('Input file name to record databases : '); readln(s);
   assign(F,s);            { Associate it }
   rewrite(F);             { Create it    }
   repeat
      clrscr;
      write('Name    = '); readln(r.name);     { Input data }
      write('Address = '); readln(r.address);
      write('Phone   = '); readln(r.phone);
      write('Age     = '); readln(r.age);
      write('Salary  = '); readln(r.salary);
      write(F,r);                 { Write data to file }
      write('Input data again (Y/N) ?');
      repeat
         c:=upcase(readkey);      { Ask user : Input again or not }
      until c in ['Y','N'];
      writeln(c);
   until c='N';
   close(F);
end.
```

Easy, right ? After creating database, display it. Modify the above program to read the file contents. This is

the hint :
1. Change `rewrite` to `reset`.
2. After the second clrscr (inside repeat..until block), add : read(F,r);
3. Remove the line "write(F,r)"

As shown below:
```
PROGRAM binary_file_read(input,output);
uses crt;
Type Temployee = record
           name    : string[20];
           address : string[40];
           phone   : string[15];
           age     : byte;
           salary  : longint;
 end;
var
   F : file of Temployee;
   c : char;
   r : Temployee;
   s : string;
begin
   clrscr;
   write('Input file name to read databases : '); readln(s);
   assign(F,s);           { Associate it }
   reset(F);            { open it    }
   repeat
      clrscr;
      Read(F,r)
      write('Name    = ',r.name);
      write('Address = ',r.address);
      write('Phone   = ',r.phone);
      write('Age     = ',r.age);
      write('Salary  = ',r.salary);

   until eof(F);
   close(F);
end.
```

That's all. You may alter the displayed message to the appropriate one. Run it and see how it's done. Now, it's time to understand file pointer. In order to know the current position of the file, Pascal use a file pointer. It's simply points to the next record or byte to read. To move the file pointer, use `seek` :
```
seek(F,recordno);
```

The `recordno` simply said the record number. If you want to read the tenth record of the file at any instant, use this :
```
seek(F,9);         { Data record number started from 0 }
read(F,r);         { r is the record variable }
```

You may conclude that it is easy to access the records. Say the record number, seek it, and read it. Any record number in range could be accessed. In range means not exceeding the maximum number of record inside that file. Therefore, it is called Random File Access.
In the other hand, text files could not behave like that. So that it requires to be handled sequentially. Therefore, there comes the jargon Sequential File Access.
Append DOES NOT work in typed files or untyped files. It is specially designed for text files. Then how can we append data to typed files ? Easy. Follow these steps :
1. Open the file with `reset`.
2. Move the file pointer after the last record using seek.

`Reset` causes file opened but the file pointer points to the first record. How can we know the number of records that is stored inside a file ? Number of records can be calculated as follows :
```
totalrecord := filesize(f);
```

Here is an example of a 'crude' database. It creates a new database if it is not exist, otherwise it appends data.

```
{ A crude database recording }
```

```
uses crt;
Type Temployee = record
     name    : string[20];
     address : string[40];
     phone   : string[15];
     age     : byte;
     salary  : longint;
 end;
var
   F : file of Temployee;
   c : char;
   r : Temployee;
   s : string;
   n : integer;
begin
   clrscr;
   write('Input file name to record databases : '); readln(s);
   assign(F,s);           { Associate it }
   {$I-}
      reset(F);           { First, open it }
   {$I+}
   n:=IOResult;
   if n<>0 then           { If it's doesn't exist then }
   begin
      {$I-}
         rewrite(F);      { Create it    }
      {$I+}
      n:=IOResult;
      if n<>0 then
      begin
         writeln('Error creating file !'); halt;
      end;
   end
   else                    { If it exists then }
      seek(F,filesize(F)); { Move file pointer to the last record }

   repeat
     :
     :
     :
     { All remains the same }
     :
     :
```

Now, how can we delete a data ? The only routine that Pascal provides is Truncate. It deletes all data starting from where file pointer points to the end of file. You may wonder how to delete a single data record. This is how : Suppose the record number you want to delete is stored in n.

```
for i:=n to totalrecord-1 do
   begin
      seek(f,i);
      read(f,r);
      seek(f,i-1);
      write(f,r);
   end;
   seek(f,totalrecord-1);
   truncate(f);
   dec(totalrecord);
```

Yes, you move the next record to the deleted record. The second next to the next and so on until the end of data. After that, you can safely truncate the last record, since the last record is already stored in record numbertotalrecord-1 and the last record would be a mere duplicate. Last step you must make is that you must adjust the totalrecord to comply with present situation (after deletion).
Easy, right ? Oh, yes ! I forgot to mention : Flush cannot be applied to binary files. It's just for text files.
It is unpractical to always having file pointer tracked. You can obtain the file pointer position by using filepos :

```
n:=filepos(F);
```

N will hold the current file position (the record number).
That's all about typed files. You may want to see this program for better details.

```
 { A crude database recording }
 uses crt;
 type Temployee = record
            name    : string[20];
            address : string[40];
            phone   : string[15];
            age     : byte;
            salary  : longint;
end;
 var
    F : file of Temployee;
    c : char;
    r : Temployee;
    s : string;
    n : integer;
 begin
    clrscr;
    write('Input file name to record databases : '); readln(s);
    assign(F,s);           { Associate it }
    {$I-}
       reset(F);           { First, open it }
    {$I+}

    n:=IOResult;
    if n<>0 then           { If it's doesn't exist then }
    begin
       {$I-}
          rewrite(F);      { Create it    }
       {$I+}
       n:=IOResult;
       if n<>0 then
       begin
          writeln('Error creating file !'); halt;
       end;
    end
    else
    begin                  { If it exists then }
       n:=filesize(F);     { Calculate total record }
       seek(F,n);          { Move file pointer PAST the last record }
    end;
    repeat
       clrscr;
       writeln('File position : ',filepos(f));
       write('Name    = '); readln(r.name);     { Input data }
       write('Address = '); readln(r.address);
       write('Phone   = '); readln(r.phone);
       write('Age     = '); readln(r.age);
       write('Salary  = '); readln(r.salary);
       write(F,r);                 { Write data to file }
       write('Input data again (Y/N) ?');
       repeat
          c:=upcase(readkey);      { Ask user : Input again or not }
       until c in ['Y','N'];
       writeln(c);
    until c='N';
    close(F);
 end.
```

Text files are usually used for INI files or setting files. Or, if your game needs special setup, you can use this skill to modify AUTOEXEC.BAT, CONFIG.SYS or even *.INI in WINDOWS directory. Typed files are usually done for recording high scores of your game, while untyped ones are for reading your game data : pictures, sounds, etc. Serious applications make an extensive use of file. Databases usually use typed files. Text files are used for making memos. Untyped ones is for reading pictures and sounds, for perhaps, you want to make presentations or just displaying the company logo.

Untyped Files

Now, we're going to discuss the untyped files. The basics is all the same. Imagine you have a typed file, but the record is one byte long. The declaration is a bit different from typed files :

```
Var  F : file;
```

This declare F as untyped files. Assign, Reset, Rewrite, and Close are still the same. But write and read is not apply in this case. Use blockwrite and blockread instead. Here is the syntax :

```
blockread (f,buffer,count,actual);
blockwrite(f,buffer,count,actual);
```

f is the file variable, **buffer** is your own buffer, not Pascal's, **count** is the number of bytes you want to read/write and **actual** is the number of bytes that has been read/written.

In untyped files, you must prepare a buffer. A buffer can be records, arrays, or even pointers. Usually, programmers use array instead of records. But, usually, if the file has certain structure, like graphic formats, programmers use records. Let's first concern about array as the buffer. Suppose I declared buffer as array of bytes :

```
var
  buffer        : array[1..2048] of byte;
  count, actual : word;
  f             : file;
```

Reading is done by this :

```
count:=sizeof(buffer);
blockread(f,buffer,count,actual);
```

Variable actual holds the number of bytes that is actually read from the disk. Likewise, writing to disk is done through blockwrite :

```
count:=sizeof(buffer);
blockwrite(f,buffer,count,actual);
```

You can even specify the number of bytes you want to read. Suppose you want to read just 512 bytes from a file :

```
blockread(f,buffer,512,actual);
```

Writing 512 bytes is just similar to reading. Now, how if the buffer is a record ? Suppose I declare the record :

```
type
  THeader = record
                 tag          : string[4];
                 width, depth : word;
                 bitperpixel  : byte;
             end;

var
  hdr : THeader;
```

That kind of header is one example of reading picture file header. Usually, after reset, programmer has to read the header to check validity. Reading the header can be done by blockread :

```
blockread(f,hdr,sizeof(hdr),actual);
```

The operator sizeof returns the number of bytes occupied by the operand or parameter automatically (so that you don't have to count it manually). If the file is good, the header is fully read. That can be checked by this :

```
if actual=sizeof(header) then    { The file has a good header }
      :
      :
```

But .... wait ! I saw somebody using untyped file with write and read. Well, that kind of person treating typed file as untyped one. That causes a LOT of pain. That's why I'm not going to teach it. But, if you insist, you can write me.
That's all about untyped files.

File Commands

Now, we're going to discuss other file commands :

1. Rename
2. Erase
3. Getenv
4. FSearch, FExpand and FSplit
5. FindFirst and FindNext
6. UnpackTime and PackTime
7. GetFTime and SetFTime
8. GetFAttr and SetFAttr
9. DiskFree and DiskSize

Number 3 through 9 need DOS unit

Rename, just like its name is to rename files. You must assign the old file name to a file variable, (the file is not necessarily be opened) then use rename :

```
   assign(f,oldname);
     rename(f,newname);
```

Erase, is to erase files. You assign the file you want to erase, then erase it. You may NOT open the file. If you've already opened it, close it first before erasing !

```
   assign(f,filename);
     erase(f);
```

You have used the crt unit so long and nothing else. Now, it's time to corporate DOS unit, so you'll probably do this :
uses crt, dos;

You need no files to add as crt and dos are both in SYSTEM.TPL. SYSTEM.TPL is always loaded when Pascal starts. Why do we need DOS unit ? Well many of file routines (that has been mentioned as number 3 thru 9) is in DOS unit. Also, interrupt handling, system time and other handy things reside in it. Let's cover the file-handling routines.

Getenv

Getenv fetches the environment string of DOS. Go to command prompt of DOS, then type SET then press Enter. DOS displays all the environment string, such as PATH, PROMPT, etc. This is example of how to get thePATH contents (s is a string) :

```
  s:=Getenv('PATH');
```

Getting PROMPT is similar : s:=Getenv('PROMPT');

FSearch

FSearch do searching files in a specified directory. Suppose you want to search for FORMAT.COM in DOS and WINDOWS directory :

```
      uses dos;
      var
        s : string;

      begin
        s:=FSearch('FORMAT.COM','C:\DOS;C:\WINDOWS');
        if s='' then
           writeln('FORMAT.COM not found')
        else
           writeln('FORMAT.COM found in ',s);
      end.
```

When found, s returns complete path and filename, otherwise empty. You may extend the directory list (the second parameter of FSearch) using semicolon such as :

```
  ... FSearch( ... , 'C:\DOS;C:\WINDOWS;C:\SCAN;C:\TOOL');
```

You may wonder that you can even search a file in the PATH environment variable. Yes, you COULD ! Do it like this :

```
 ... FSearch( ... , getenv('PATH'));
```

FExpand
FExpand expands a simple file name into a full name (drive, full directory, and the file name itself). It is especially useful when user inputs a relative directory, like this (s is a string) :
```
 s:=FExpand('..\README');
```

S will be like this (for example) : 'C:\PASCAL\LESSON.1\README'
FSplit
It is just contrary to FExpand, splits a fully qualified name into directory, file name, and extension. Example :
```
  var
      s : string;
      d : dirstr;
      n : namestr;
      e : extstr;

      :
      :
      :
    s:='C:\WINDOWS\WIN.INI';
    fsplit(s,d,n,e);    { d = 'C:\WINDOWS\', n = 'WIN', e = '.INI' }
```
Look at this examples for better details
```
      uses dos;
      var
        s : string;
        d : dirstr;
        n : namestr;
        e : extstr;

      begin
        s:=FSearch('FORMAT.COM',getenv('PATH'));
        if s='' then
        begin
           writeln('FORMAT.COM not found');
           halt;
        end;

        writeln('FORMAT.COM found in ',s);
        fsplit(s,d,n,e);
        writeln(d,' ',n,' ',e);
      end.
```

STRINGS

The string in Pascal is actually a sequence of characters with an optional size specification. The characters could be numeric, letters, blank, special characters or a combination of all. Extended Pascal provides numerous types of string objects depending upon the system and implementation. We will discuss more commonly types of strings used in programs.

You can define a string in many ways:

- **Character arrays**: This is a character string which is a sequence of zero or more byte-sized characters enclosed in single quotes.
- **String variables**: The variable of String type, as defined in Turbo Pascal.
- **Short strings**: The variable of String type with size specification.
- **Null terminated strings**: The variable of pchar type.
- **AnsiStrings**: Ansistrings are strings that have no length limit.

Pascal provides only one string operator . string **concatenation** operator (+).

Examples

The following program prints first four kinds of strings. We will use AnsiStrings in the next example.

```
program exString;
var
   greetings: string;
   name: packed array [1..10] of char;
   organisation: string[10];
   message: pchar;
begin
   greetings := 'Hello ';
   message := 'Good Day!';
   writeln('Please Enter your Name');
   readln(name);
   writeln('Please Enter the name of your Organisation');
   readln(organisation);
   writeln(greetings, name, ' from ', organisation);
  writeln(message);
end.
```

Following example makes use of few more functions, let's see:

```
program exString;
uses sysutils;
var
   str1, str2, str3 : ansistring;
   str4: string;
   len: integer;
begin
   str1 := 'Hello ';
   str2 := 'There!';
  (* copy str1 into str3 *)
   str3 := str1;
   writeln('appendstr( str3, str1) :  ', str3 );
  (* concatenates str1 and str2 *)
   appendstr( str1, str2);
   writeln( 'appendstr( str1, str2) ' , str1 );
   str4 := str1 + str2;
   writeln('Now str4 is: ', str4);

  (* total lenghth of str4 after concatenation  *)
   len := byte(str4[0]);
   writeln('Length of the final string str4: ', len);
end.
```

Functions used in string manipulation include:

1. **Delete(str, pos, num);** This procedure deletes a substring of length num starting at position pos in the string variable str. The remaining characters (if any) in the string are shifted left. This routine is similar to the Watcom StrDelete routine.
2. **Insert(src, dest, pos);** This procedure inserts the string src into the string dest at position pos. The variables src and dest are both of type string; pos is of type integer. An error message will be issued if the pos is past the end of the string, or the string becomes too long. This is stricter error checking than that provided by the same routine in the Turbo Pascal compiler. This routine is similar to the Watcom StrInsert routine.
3. **Copy(str, pos, num);** This function returns a substring of the string variable str of length num starting at position pos. It returns the null string (i.e. a string of length zero) if pos is greater than the length of the string.
4. **Length(str).** This function returns the length of the string argument in characters. For a string constant, this is the number of characters in the constant. For a string variable, it is the number of characters currently stored in the variable. Length gets its value from the special length byte kept at the zeroth index of every string. It is similar to the Watcom Strlen function.
5. **Pos(srch, str);** This function returns the position in the string str at which the string srch is found. It returns 0 if srch was not found in str. For example,

   i := Pos('exec', 'autoexec.bat');

   will cause i to have the value 5.
6. **Str(val, str).** This procedure puts the value of the given variable val (which may be integer or real) into the given string variable str as an ASCII representation of the number. The variable val can actually have format specifiers of the same type found in write procedure calls. One can specify a field width for integers and reals. Reals may also have a precision field, just as with write. In general

   ```
   Str(i:n, dest);
   6.writeln(dest);
   1is the same as:
   2writeln(i : n);
   ```

   3if dest is a string large enough to hold the written number. Note that one places a space in front of integer operands if no field width is provided. A field width of zero eliminates this. This space is not output when Borland deviations (+b) are enabled. An error occurs if the string is not large enough to hold the number.
7. **Val(str, variable, code);** This procedure is the inverse of the str procedure; it examines the string contained in str and converts it to a number which it stores in var. The type of the number is either real or integer, depending on the type of var. Leading and trailing spaces are not permitted; the valid formats for the number are the same as those for constants in Alice Pascal. In addition, real numbers with nothing before the decimal point (e.g. ".1") are acceptable.

   If the conversion is successful, the integer variable code is set to zero; otherwise, it contains an index into str of the character on which the conversion failed. If this is the case, the value of var is undefined. For example,

   Val('7.8', n, stat);

   will set n to 7.8 and stat to zero, while

   Val('7y2', n, stat);

   will result in stat being 2 (the index of the letter y in the string) and the value of n will be undefined.

8. **UpCase(c);** This function returns a character which is the upper-case equivalent of the character c. For example,

   ```
   ch := UpCase('b');
   ```

   will result in ch having the value 'B'. This routine is handy if you want to accept user input in either upper or lower case.

DYNAMIC DATA STRUCTURES

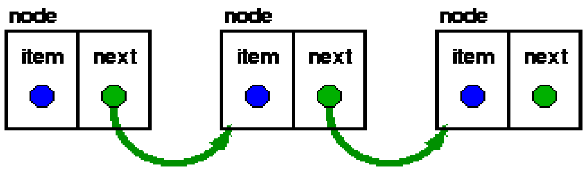
Structures which grow or shrink as the data they hold changes. Linked Lists, queues, stacks and trees are all dynamic structures.

LINKED LISTS

A linked list is a linearly arranged collection of elements that allows insertion and deletion at any place in the sequence.

A linked list is a collection of objects linked to one another like a string of pearls. As a minimum, a linked list knows about its first element, and each element knows about its successor. A doubly linked list is an extension of a singly linked list whose elements also know about their predecessor. Both provide a way insert a new element at any location. Another advantage of linked lists is that they occupy only the amount of space required to hold their elements whereas ordered collections may occupy more space because they are normally only partially filled. However, elements of linked lists (their nodes) must be packaged in more complex objects.

The linked list is a very flexible dynamic data structure: items may be added to it or deleted from it at will. A programmer need not worry about how many items a program will have to accommodate: this allows us to write robust programs which require much less maintenance. A very common source of problems in program maintenance is the need to increase the capacity of a program to handle larger collections: even the most generous allowance for growth tends to prove inadequate over time!



In a linked list, each item is allocated space as it is added to the list. A link is kept with each item to the next item in the list.

Each node of the list has two elements

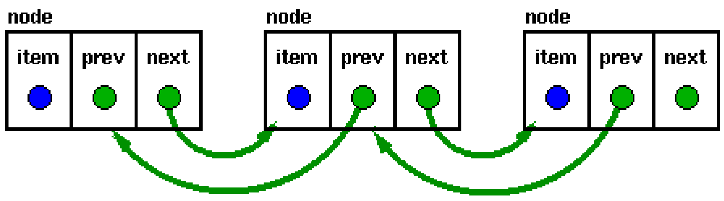
- the item being stored in the list and
- a pointer to the next item in the list

The last node in the list contains a NULL pointer to indicate that it is the end or tail of the list.

As items are added to a list, memory for a node is dynamically allocated. Thus the number of items that may be added to a list is limited only by the amount of memory available.

Circularly Linked Lists

By ensuring that the tail of the list is always pointing to the head, we can build a circularly linked list. If the external pointer (the one in struct t_node in our implementation), points to the current "tail" of the list, then the "head" is found trivially via tail->next, permitting us to have either LIFO or FIFO lists with only one external pointer. In modern processors, the few bytes of memory saved in this way would probably not be regarded as significant. A circularly linked list would more likely be used in an application which required "round-robin" scheduling or processing.



Doubly linked lists have a pointer to the preceding item as well as one to the next.

They permit scanning or searching of the list in both directions. (To go backwards in a simple list, it is necessary to go back to the start and scan forwards.) Many applications require searching backwards and forwards through sections of a list: for example, searching for a common name like "Kim" in a Korean telephone directory would probably need much scanning backwards and forwards through a small region of the whole list, so the backward links become very useful. In this case, the node structure is altered to have two links:

```
struct t_node {
     void *item;
     struct t_node *previous;
     struct t_node *next;
}node;
```

Stacks

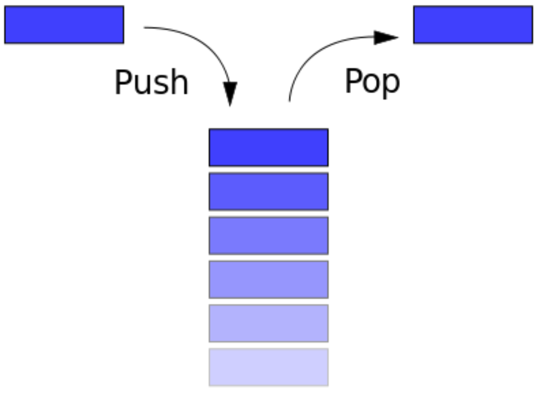
A stack is a collection whose elements can be accessed only at one end called the top of the stack. The operation adding an element on the top of the stack is called **push**, the operation removing the top element from the stack is called **pop**.

A stack is a basic data structure that can be logically thought as linear structure represented by a real physical stack or pile, a structure where insertion and deletion of items takes place at one end called top of the stack.

The basic concept can be illustrated by thinking of your data set as a stack of plates or books where you can only take the top item off the stack in order to remove things from it. This structure is used all throughout programming.

The basic implementation of a stack is also called a LIFO (Last In First Out) to demonstrate the way it accesses data, since as we will see there are various variations of stack implementations.

There are basically three operations that can be performed on stacks . They are



1) inserting an item into a stack (push).
2) deleting an item from the stack (pop).
3) displaying the contents of the stack (pip).

Stack<item-type> Operations

- ♦ *push(new-item:item-type):* Adds an item onto the stack.
- ♦ *top():item-type*: Returns the last item pushed onto the stack.
- ♦ *pop():* Removes the most-recently-pushed item from the stack.
- ♦ *is-empty():*Boolean: True if no more items can be popped and there is no top item.
- ♦ *is-full():Boolean*: True if no more items can be pushed.
- ♦ *get-size():Integer*: Returns the number of elements on the stack.

A formal specification of a stack class would look like:

```
typedef struct t_stack *stack;
stack ConsStack( int max_items, int item_size );
/* Construct a new stack
   Pre-condition: (max_items > 0) && (item_size > 0)
   Post-condition: returns a pointer to an empty stack
*/
void Push( stack s, void *item );
/* Push an item onto a stack
   Pre-condition: (s is a stack created by a call to ConsStack) &&
                  (existing item count < max_items) &&
                  (item != NULL)
   Post-condition: item has been added to the top of s
*/
void *Pop( stack s );
/* Pop an item of a stack
   Pre-condition: (s is a stack created by a call to
                  ConsStack) &&
                  (existing item count >= 1)
```

```
     Post-condition: top item has been removed from s
  */
```

Points to note:

1. A stack is simply another collection of data items and thus it would be possible to use exactly the same specification as the one used for our general collection. However, collections with the LIFO semantics of stacks are so important in computer science that it is appropriate to set up a limited specification appropriate to stacks only.
2. Although a linked list implementation of a stack is possible (adding and deleting from the head of a linked list produces exactly the LIFO semantics of a stack), the most common applications for stacks have a space restraint so that using an array implementation is a natural and efficient one (In most operating systems, allocation and de-allocation of memory is a relatively expensive operation, there is a penalty for the flexibility of linked list implementations.).

Queues

A queue is a collection in which elements are added at one end and retrieved at the other. As in a queue in a bank, the first item entering the queue is also the first to be retrieved and removed from the queue and this is why a queue is also called a first-in-first-out (FIFO) structure.

In spite of its simplicity, the queue is a very important concept with many applications in simulation of real life events such as lines of customers at a cash register or cars waiting at an intersection, and in programming (such as printer jobs waiting to be processed.

A queue is also called a FIFO (First In First Out) to demonstrate the way it accesses data. Its operations include:

- *Enqueue (new-item:item-type):* Adds an item onto the end of the queue.
- *front():item-type*: Returns the item at the front of the queue.
- *dequeue():* Removes the item from the front of the queue.
- *is-empty():Boolean*: True if no more items can be dequeued and there is no front item.
- *is-full():Boolean*: True if no more items can be enqueued.
- *get-size():Integer*: Returns the number of elements in the queue.

When you want to enqueue something, you simply add it to the back of the item pointed to by the tail pointer. So the previous tail is considered next compared to the item being added and the tail pointer points to the new item. If the list was empty, this doesn't work, since the tail iterator doesn't refer to anything

```
method enqueue(new_item:item_type)
  if is-empty()
    list.prepend(new_item)
    tail := list.get-begin()
  else
    list.insert_after(new_item, tail)
    tail.move-next()
  end if
 end method
```

The front item on the queue is just the one referred to by the linked list's head pointer

```
method front():item_type
   return list.get-begin().get-value()
 end method
```

When you want to dequeue something off the list, simply point the head pointer to the previous from head item. The old head item is the one you removed of the list. If the list is now empty, we have to fix the tail iterator.

```
     method dequeue()
       list.remove-first()
       if is-empty()
         tail := list.get-begin()
       end if
      end method
```

A check for emptiness is easy. Just check if the list is empty.

```
     method is-empty():Boolean
       return list.is-empty()
      end method
```

A check for full is simple. Linked lists are considered to be limitless in size.

```
     method is-full():Boolean
       return False
      end method
```

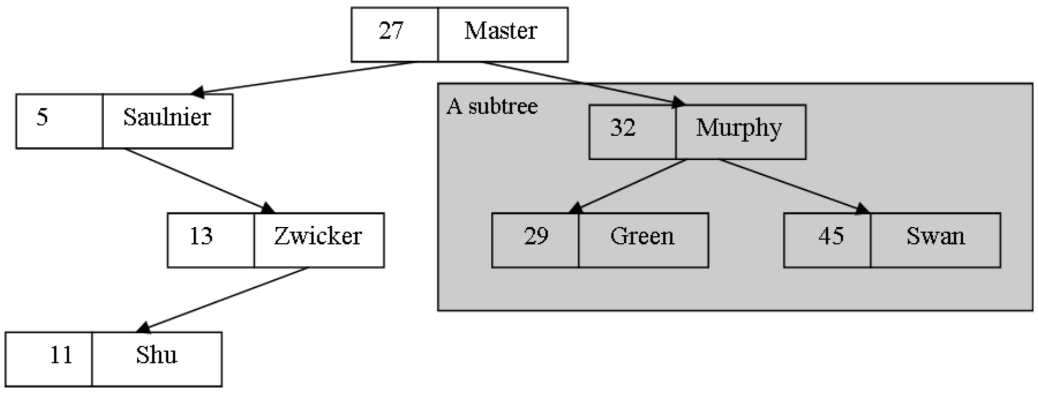
A check for the size is again passed through to the list.

```
     method get-size():Integer
       return list.get-size()
      end method
    end type
```

TREE

A tree is a structure whose graphical representation looks like a family tree: It starts with a root at the top, and branches downward. Typical uses of trees are the representation of the class hierarchy, storing data for fast access, and translation of program code.

A tree is a two-dimensional collection of objects called nodes. Three examples are a class hierarchy tree, a family tree, and a tree of student records as in figure below.



The nodes of a tree can be classified as follows:

- The node at the top of the tree is called the **root**. This is the node through which the tree can be accessed. The node with ID 27 in our example is the root of the tree. A tree has exactly one root.
- A node that is not a root and has at least one child is an **internal node**. The records with IDs 13 and 32 in our example are internal nodes. Every internal node has exactly one parent node but it may have any number of children, unless the definition of the tree specifies otherwise.
- A node that does not have any children is called a **leaf**. The node with ID 11 in our example is a leaf node. Like an internal node, a leaf has exactly one parent node.

An interesting and useful property of trees is that any node in the tree can be treated as the root of a new tree consisting of all the underlying nodes. Such a tree is called a **subtree** of the original tree. As an example, the node with ID 32 in figure above can be used as the root of a tree with children 29 and 45. Leaves, such as 29, can be thought of as degenerate trees that consist of a root and nothing else.

Trees are very important in computer applications and a variety of trees have been devised to provide the best possible performance for different uses. The differences between different kinds of trees are in the number of children that a node may have (Each node with the exception of the root always has exactly one parent), and the way in which the tree is managed (how node are added and deleted). The subject of trees is not trivial and we will restrict our presentation to the example of a simple binary tree. In a binary tree, each node may have at most two children as in figure above.

S O R T I N G A N D S E A R C H I N G

S O R T I N G

Consider sorting the values in an array A of size N. Most sorting algorithms involve what are called comparison sorts; i.e., they work by comparing values. Comparison sorts can never have a worst-case running time less than O(N log N). Simple comparison sorts are usually O(N2); the more clever ones are O(N log N).

Three interesting issues to consider when thinking about different sorting algorithms are:

1. Does an algorithm always take its worst-case time?
2. What happens on an already-sorted array?
3. How much space (other than the space for the array itself) is required?

We will discuss four comparison-sort algorithms:

1. Selection sort
2. Insertion sort
3. Bubble sort
4. Merge sort

Selection Sort

This algorithm selects the smallest (or the largest if the array is to be sorted in descending order) element of the array and place it at the head of the array. Then the process is repeated for the remainder of the array. It is slightly faster than bubble sort because it looks at progressively smaller part of the array each time since the front part is already sorted.

The idea behind selection sort is:

♦ Find the smallest value in A; put it in A[0].

♦ Find the second smallest value in A; put it in A[1] etc.

Here's the algorithm for selection sort:

```
SELECTION_SORT (A)
for i ← 1 to n-1 do
    x←i
    Min ← A[i]
    for j ← i + 1 to n do
        If min > A[j] then  x←j;
    End
    If i is not equal to x then swap(A[i], A[x]);
End
```

Insertion Sort

Inserts each element of an array into its proper position, leaving progressively larger stretches of the array sorted. This means that the sort iterates down an array and the part of the array already covered is in order. The idea behind insertion sort is:

♦ Put the first 2 items in correct relative order.

♦ Insert the 3rd item in the correct place relative to the first 2.

♦ Insert the 4th item in the correct place relative to the first 3 etc.

For instance, if the array to be sorted is	5	2	3	1	4
1. First 2 is replaced by 5 resulting in	2	5	3	1	4
2. 3 is inserted in the correct position relative to 2, 5 Hence	2	3	5	1	4
3. 1 is placed in the right position relative to 2,3,5 Hence	1	2	3	5	4
4. 4 is last placed in the right position in 1,2,3,5 series	1	2	3	4	5

BUBBLE/ SINK SORT

Bubble sort is an algorithm that sorts by iterating down an array to be sorted from the first element to the last comparing each pair of elements and switching/ swapping them if necessary. This process is repeated as many times as possible until the array is sorted.

Its called bubble sort because the smallest element gradually move upward like a bubble. It called sink sort because the largest element gradually sinks

For example

Suppose A is an array of 5 integers and the initial value in A are:

```
        67      33      21      84      4
Pass 1:
        33      67      21      84      4
        33      21      67      84      4
        33      21      67      84      4
        33      21      67      4       84
Pass 2:
        21      33      67      4       84
        21      33      67      4       84
        21      33      4       67      84
        21      33      4       67      84
Pass 3:
        21      33      4       67      84
        21      4       33      67      84
        21      4       33      67      84
        21      4       33      67      84
Pass 4:
        21      4       33      67      84
        21      4       33      67      84
        21      4       33      67      84
        21      4       33      67      84
Pass 5:
        21      4       33      67      84
        21      4       33      67      84
        21      4       33      67      84
        21      4       33      67      84
```

The process continues up to the 5th pass when the array will be sorted. The algorithm is as follows:

```
SEQUENTIAL BUBBLESORT (A)
for i ← 1 to length [A] do
    for j ← 1 To length [A]-1 do
        If A[j] < A[j+1] then
            Exchange A[j] ↔ A[j+1]
```

Implementation of bubble sort on a 10 number array in Pascal

```
Program bubble_sort(input, output);
USES crt;
TYPE    nos=ARRAY[1..10] of INTEGER;
VAR     I:Integer;
        Ten_nos:nos;
PROCEDURE bubblesort();
VAR j, temp: Integer;
BEGIN
        FOR I := 1 TO 10 DO BEGIN
                FOR J := 1 TO (10-1) DO BEGIN
                        IF (Ten_nos[J] > Ten_nos[J+1]) THEN BEGIN
                                Temp:=Ten_nos[J];
                                Ten_nos[J]:=Ten_nos[J+1];
                                Ten_nos[J+1]:=Temp;
                        End
                End
        End
END;
BEGIN
        Clrscr();
        Writeln('THE TEN NUMBERS IN THE ARRAY'); writeln;
```

```
          FOR i:=1 To 10 Do Begin
                 Write('Enter number ', I, ' In the array: ');
                 Readln(Ten_nos[i]);
          END;
          Writeln;writeln('THE ARRAY BEFORE SORTING');writeln;
          FOR i:=1 TO 10 DO write(Ten_nos[i]:5);
          Bubblesort(); //call procedure
          Writeln; writeln('THE ARRAY AFTER SORTING'); writeln;
          FOR i:=1 TO 10 DO write(Ten_nos[i]:5);
          Readln;
   END.
```

Merge Sort

Merge sort is based on the divide-and-conquer paradigm. Its worst-case running time has a lower order of growth than insertion sort. Since we are dealing with sub problems, we state each sub problem as sorting a sub array A[p .. r]. Initially, p = 1 and r = n, but these values change as we recurse through sub problems.

To sort A[p .. r]:

1. Divide Step. If a given array A has zero or one element, simply return; it is already sorted. Otherwise, split A[p .. r] into two sub arrays A[p .. q] and A[q + 1 .. r], each containing about half of the elements of A[p .. r]. That is, q is the halfway point of A[p .. r].
2. Conquer Step. Conquer by recursively sorting the two sub arrays A[p .. q] and A[q + 1 .. r].
3. Combine Step. Combine the elements back in A[p .. r] by merging the two sorted sub arrays A[p .. q] and A[q + 1 .. r] into a sorted sequence. To accomplish this step, we will define a procedure MERGE (A, p, q, r).

Note that the recursion bottoms out when the sub array has just one element, so that it is trivially sorted.

Algorithm: Merge Sort

To sort the entire sequence A[1 .. n], make the initial call to the procedure MERGE-SORT (A, 1, n).

```
MERGE-SORT (A, p, r)
    IF p < r                                         // Check for base case
        THEN q = FLOOR[(p + r)/2]                    // Divide step
             MERGE (A, p, q)                         // Conquer step.
             MERGE (A, q + 1, r)                     // Conquer step.
        MERGE (A, p, q, r)                           // Conquer step.
```

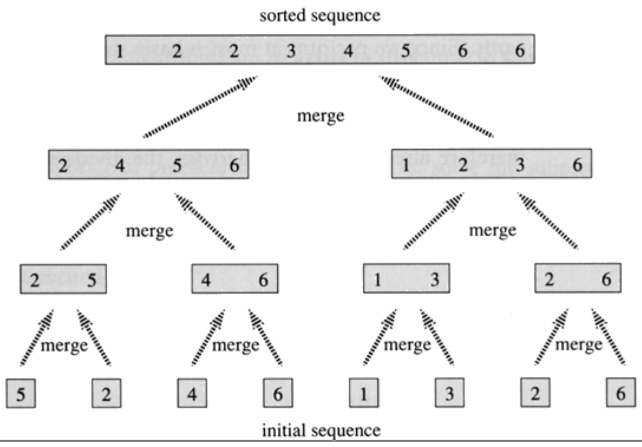
Example: Bottom-up view of the above procedure for n = 8.

SEARCHING ALGORITHM

These are algorithms used for finding an item or data from an array or record structures. These algorithms include the following: -

1. Sequential Search
2. Binary Search

Sequential Searching



This is an algorithm that finds a particular value in a list or array by checking everyone of its elements, on at a time from the beginning until the desired one is found. It can search from both ordered and unordered lists.

It is practical when the list has few elements or when performing a single search in an unordered list. The algorithm is as follows:

```
Input n, x(i), i=1,…,n; key
found = false;
i=1
while i ≤ n and not(found)
{
        if key = x(i) then found = true
        else i=i+1
}
If found then
        print “key is found”
else
        print ”key is not found”
```

Pascal Program implementation of sequential search

```
Program Sequential_Search(input, output);
uses crt;
Var     fivenos:array[0..4] of Integer;
        Key,position,i:Integer;
Function  Sequential(no:Integer):integer;
Var     found:boolean;
Begin
        found:=false;  i:=0;
        Repeat
                If (fivenos[i]=no) then  Found:=true
                Else i:=i+1;
        Until  ((i>4) or (found=true));
        If (found=true) then  sequential:=i+1
        Else sequential:=-1;
End;
Begin
        clrscr();
        For i:=0 to 4 do Begin
                Write('Enter number ',(i+1),' in the array: ');
                Readln(fivenos[i]);
        End;
        Writeln; write('Enter the number/ key to be found:  ');
        readln(key);
        writeln; writeln;
        position:=sequential(key);
        if position=-1 then writeln(key, ' is not found in the array')
        else writeln(key, ' is found at position ',position,' in the array');
        writeln;
        Readln();
End.
```

Binary searching

A binary search finds the position of a specified value within a sorted array. At each stage, the algorithm compares the input value with the key value of the middle element of the array. If the key match, the matching element has been found so its index (position) is returned. Otherwise, if the sort sort key is less than the middle element’s key, then the algorithm repeats its action on the sub array to the left of the middle element or if the input key is greater then the search is done on the sub array to the right.

If the remaining array to be searched is reduced to zero, then the key can not be found in the array. Can search from ordered list only.

```
binary search(x,L)
{
   let n = length of L, i=n/2.
   if (n = 0) return no match
   else if (L[i] matches x) return L[i]
   else if (L[i] > x) binary search(x,L[1..i-1])
   else binary search(x,L[i+1..n])
 }
```

Recursion is not really necessary:

```
Binary search(x,L)
{
        let n = length of L
        let a = 1, b = n
        while (L[i = (a+b)/2] doesn't match)
            if (L[i] > x) b = i-1
            else a = i+1
            if a>b return no match
        return L[i]
}
```

STRUCTURED PROGRAMMING IN C

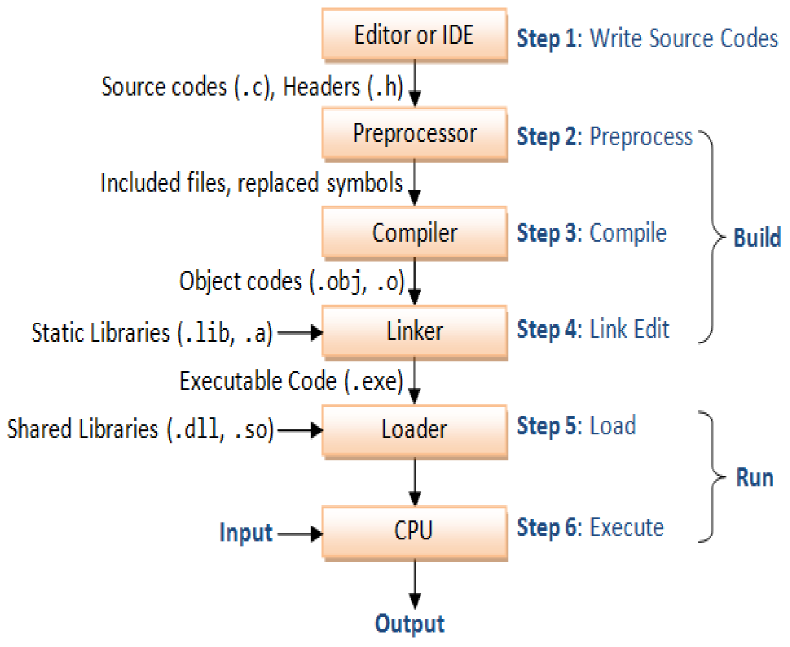
C language was created by Dennis Ritchie at Bell Telephone Laboratories in 1972. it was designed to design Unix operating systems. The C language is so named because post cedes B language developed by Ken Thompson in the same place (Bell Labs).

Why use C language in programming?

1. It is popular and preferred by many programmers
2. It is powerful and flexible language
3. It is a portable language
4. It contains a handful of keywords
5. It is modular—enables programmer to break large programs into small units/ functions.

PROGRAM DEVELOPMENT IN C

- Step 1: Write the source codes (.c) and header files (.h).
- Step 2: Pre-process the source codes according to the preprocessor directives. The preprocessor directives begin with a hash sign (#), such as #include and #define. They indicate that certain manipulations (such as including another file or replacement of symbols) are to be performed BEFORE compilation.
- Step 3: Compile the pre-processed source codes into object codes (.obj, .o).
- Step 4: Link the compiled object codes with other object codes and the library object codes (.lib, .a) to produce the executable code (.exe).
- Step 5: Load the executable code into computer memory.
- Step 6: Run the executable code.



Sample program

```
#include<stdio.h>
int main()
{
        printf("Hello, welcome to programming in C\n");
        return 0;
}
```

A function is an independent section of a program code that performs a certain task, has been assigned a name and may return a value after execution.

Main() function consists of the name of the function, *main*, its return data type (that is *int*), empty parenthesis, *()*, and a pair of braces *{}*. Within the braces are statements that make up the main body of the program.

The "#**include**" is called a *preprocessor directive*. A preprocessor directive begins with a # sign, and is processed before compilation. The directive **"#include <stdio.h>"** tells the preprocessor to include the "stdio.h" header file to support input/output operations. This line shall be present in all our programs.

Comments are NOT executable and are ignored by the compiler. But they provide useful explanation and documentation to your readers (and to yourself later). There are two kinds of comments:

1. Multi-line Comment: begins with /* and ends with */. It may span more than one lines
2. End-of-line Comment: begins with // and lasts until the end of the current line.

The **printf** function is the standard C way of displaying output on the screen. The quotes tell the compiler that you want to output the literal string as-is. The '\n' sequence is actually treated as a single character that stands for a newline.

Finally, at the end of the program, we **return a value** from main to the operating system by using the return statement. This return value is important as it can be used to tell the operating system whether our program succeeded or not. A return value of 0 means success.

C Terminology and Syntax

1. **Statement**: A programming statement performs a piece of programming action. It must be terminated by a semi-colon (;)
2. **Preprocessor Directive**: The #include (Line 4) is a preprocessor directive and NOT a programming statement. A preprocessor directive begins with hash sign (#). It is processed before compiling the program. A preprocessor directive is NOT terminated by a semicolon - Take note of this rule.
3. **Block**: A block is a group of programming statements enclosed by braces { }. This group of statements is treated as one single unit. There is one block in this program, which contains the body of the main() function. There is no need to put a semi-colon after the closing brace.
4. **Comments**: A multi-line comment begins with /* and ends with */. An end-of-line comment begins with // and lasts till the end of the line. Comments are NOT executable statements and are ignored by the compiler. But they provide useful explanation and documentation. Use comments liberally.
5. **Whitespaces**: Blank, tab, and newline are collectively called whitespaces. Extra whitespaces are ignored, i.e., only one whitespace is needed to separate the tokens. But they could help you and your readers better understand your program. Use extra whitespaces liberally.
6. **Case Sensitivity**: C is case sensitive - a ROSE is NOT a Rose, and is NOT a rose.

Basic Arithmetic Operations

Operator	Meaning	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus (Remainder)	x % y
++	Increment by 1 (Unary)	++x or x++
--	Decrement by 1 (Unary)	--x or x--

Comparison Operators

Take note that the comparison operator for equality is a double-equal sign (==); whereas a single-equal sign (=) is the assignment operator.

Operator	Meaning	Example
==	Equal to	`x == y`
!=	Not equal to	`x != y`
>	Greater than	`x > y`
>=	Greater than or equal to	`x >= y`
<	Less than	`x < y`
<=	Less than or equal to	`x <= y`

Logical operators

Suppose that you want to check whether a number x is between 1 and 100 (inclusive), i.e., 1 <= x <= 100. There are two simple conditions here, (x >= 1) AND (x <= 100). In programming, you cannot write 1 <= x <= 100, but need to write (x >= 1) && (x <= 100), where "&&" denotes the "AND" operator. Similarly, suppose that you want to check whether a number x is divisible by 2 OR by 3, you have to write (x % 2 == 0) || (x % 3 == 0) where "||" denotes the "OR" operator.

There are three so-called logical operators that operate on the boolean conditions:

Operator	Meaning	Example
&&	Logical AND	`(x >= 1) && (x <= 100)`
||	Logical OR	`(x < 1) || (x > 100)`
!	Logical NOT	`!(x == 8)`

For examples:

```
// Return true if x is between 0 and 100 (inclusive)
(x >= 0) && (x <= 100)  // AND (&&)
// Incorrect to use 0 <= x <= 100
// Return true if x is outside 0 and 100 (inclusive)
(x < 0) || (x > 100)       // OR (||)
!((x >= 0) && (x <= 100))  // NOT (!), AND (&&)
// Return true if "year" is a leap year
// A year is a leap year if it is divisible by 4 but not by 100, or it is
divisible by 400.
((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0)
```

DATA TYPES

Data types refer to an extensive system used for declaring variables or functions of different types. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.

The types in C can be classified as follows:

S.N.	Types and Description
1	**Basic Types:** They are arithmetic types and consists of the two types: (a) integer types and (b) floating-point types.
2	**Enumerated types:** They are again arithmetic types and they are used to define variables that can only be assigned certain discrete integer values throughout the program.
3	**The type void:** The type specifier *void* indicates that no value is available.
4	**Derived types:** They include (a) Pointer types, (b) Array types, (c) Structure types, (d) Union types and (e) Function types.

The array types and structure types are referred to collectively as the aggregate types. The type of a function specifies the type of the function's return value.

Integer Types

Following table gives you details about standard integer types with its storage sizes and value ranges:

Type	**Storage size**	**Value range**
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

To get the exact size of a type or a variable on a particular platform, you can use the sizeof operator. The expressions **sizeof(type)** yields the storage size of the object or type in bytes. Following is an example to get the size of int type on any machine:

```
#include <stdio.h>
#include <limits.h>
int main()
{
        printf("Storage size for int : %d \n", sizeof(int));
        return 0;

}
```

Floating-Point Types

Following table gives you details about standard floating-point types with storage sizes and value ranges and their precision:

Type	**Storage size**	**Value range**	**Precision**
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

The header file float.h defines macros that allow you to use these values and other details about the binary representation of real numbers in your programs. Following example will print storage space taken by a float type and its range values:

```
#include <stdio.h>
#include <float.h>
int main()
{
   printf("Storage size for float : %d \n", sizeof(float));
   printf("Minimum float positive value: %E\n", FLT_MIN );
   printf("Maximum float positive value: %E\n", FLT_MAX );
   printf("Precision value: %d\n", FLT_DIG );
   return 0;

}
```

<u>Variables</u>

- A variable is a named storage location, where data may be stored and later changed.
- An identifier is a more general term for a named location, which may contain either data or code.
    1. Identifiers must begin with a letter or an underscore, preferable letters for user programs.

2. The remaining characters must be either alphanumeric or underscores.
3. Identifiers may be of any length, but only the first 31 characters are examined in most implementations.
4. Identifiers are case sensitive, so "NUMBER", "number", and "Number" are three different identifiers.
5. By convention ordinary variables begin with a lower case letter, globals with a Single Capital, and constants in ALL CAPS.
6. Multi-word variables may use either underscores or "camel case", such as "new_value" or "newValue".
7. Integers are usually assigned variable names beginning with the letters I, J, K, L, M, or N, and floating point variables are usually assigned names beginning with other letters.
8. Identifiers may not be the same as reserved words.

- All variables must be declared before they can be used. To declare a variable, the syntax is:

  ```
  type variable_list;
  ```

  For instance,

  ```
  int    i, j, k;
  char   c, ch;
  float  f, salary;
  double d;
  ```

- Variables may be given an initial value at the time they are declared. This is called "initialization", or "initializing the variables". The syntax is:

  ```
  type variable_name=value;
  ```

  For instance,

  ```
  extern int d = 3, f = 5;    // declaration of d and f.
  int d = 3, f = 5;           // definition and initializing d and f.
  byte z = 22;                // definition and initializes z.
  char x = 'x';               // the variable x has the value 'x'.
  ```

- Variables may be declared "const", meaning that their values cannot be changed.
  1. **const** variables MUST be initialized at the time they are declared.
  2. By convention, const variables are named using ALL CAPS.
  3. Examples:

     const double PI = 3.14159;

     const int MAXROWS = 100;

Note: K&R C did not have the const keyword, and so the #define pre-processor macro was used to define constant values. The const qualifier is a better approach when it is available, because it allows the compiler to perform type checking among other reasons. For CS 107 we will defer the discussion of #define until we get to the chapter on the pre-processor.

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

Escape Sequences

There are certain characters in C when they are preceded by a backslash they will have special meaning and they are used to represent like newline (\n) or tab (\t). Here, you have a list of some of such escape sequence codes:

Escape sequence	Meaning
\\	\ character
\'	' character
\"	" character
\?	? character
\a	Alert or bell
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\ooo	Octal number of one to three digits
\xhh . . .	Hexadecimal number of one or more digits

For instance,

```
#include <stdio.h>

int main()
{
   printf("Hello\tWorld\n\n");

   return 0;
}
```

CONSTANTS

There are two simple ways in C to define constants:

1. Using #define preprocessor.
2. Using const keyword.

The #define Preprocessor

Following is the form to use #define preprocessor to define a constant:

```
#define identifier value
```

For example,

```
#include <stdio.h>
#define LENGTH 10
#define WIDTH  5
#define NEWLINE '\n'
int main()
{
   int area;
   area = LENGTH * WIDTH;
   printf("value of area : %d", area);
   printf("%c", NEWLINE);
   return 0;
}
```

The const Keyword

You can use const prefix to declare constants with a specific type as follows:

```
const type variable = value;
```

Following example explains it in detail:

```
#include <stdio.h>
int main()
{
   const int  LENGTH = 10;
   const int  WIDTH  = 5;
   const char NEWLINE = '\n';
   int area = LENGTH * WIDTH;
   printf("value of area : %d", area);
   printf("%c", NEWLINE);
   return 0;
}
```

Placeholders

These placeholders are called format specifiers. Many other format specifiers work with printf. All format specifiers are:

Data type	printf conversion specification	scanf conversion specification
long double	%Lf	%Lf
double	%f	%lf
float	%f	%f
unsigned long int	%lu	%lu
long int	%ld	%ld
unsigned int	%u	%u
int	%d	%d
unsigned short	%hu	%hu
short	%hd	%hd
char	%c	%c
string	%s	%s

<u>Input using scanf()</u>

The scanf() function is the input method equivalent to the printf() output function. In its simplest invocation, the scanf format string holds a single placeholder representing the type of value that will be entered by the user. These placeholders are as shown above.

The scanf() function requires the memory address of the variable to which you want to save the input value. Instead, the simple technique is to use the address-of operator, &.

For instance,

```
#include<stdio.h>
int main(void)
{
   int a;
   printf("Please input an integer value: ");
   scanf("%d", &a);
   printf("You entered: %d\n", a);
   return 0;
}
```

CONTROL STRUCTURES

These are statement that transfer control of a program execution. The control structures are classified into two categories, namely:

1. Selection control structures. These are statements used to choose among alternative courses of action. They include `if`, `if … else` and `switch` selection statements
2. Repetition control structures such as `while`, `do .. while` and `for` repetition structures.

The `if` selection statement

It is a selection control structure that allows a statement or group of statements to be executed only when a given condition is evaluated to `true`. The general syntax is

```
if (<condition>)
      statement;
```

Or for multiple statements in the construct

```
if (<condition>){
      Statement1;
      Statement2;
}
```

For instance, a program that displays the message "Passed, promoted to next class" when the score of a student entered by the user is 60 and above.

```
#include<stdio.h>
int main()
{
      int marks=0;
      printf("Enter students' score in the exams ");
      scanf("%d",&marks)
      if (marks>=60) printf("\n\nPassed, promoted to next class");
      return 0;
}
```

The `if` statement is a single selection single-entry/ single-exit structure that allows the enclosed statements to be executed only when the condition is evaluated to `true` otherwise no action is taken.

The `if … else` selection statement

It is a selection structure that allows you as a programmer to specify that different actions are to be performed when the condition is `true` and when the condition is `false`. The general syntax is

```
if (<condition>)
      statement;
else
      Statement2;
```

Or for multiple statements in the construct

```
if (<condition>){
      Statement1;
      Statement2;
}else {
      Statement3;
      Statement4;
}
```

For instance, a program that displays the message "Passed, promoted to next class" when the score of a student entered by the user is 60 and above otherwise it displays "Pull up next time".

```
#include<stdio.h>
int main()
{
      int marks=0;
      printf("Enter students' score in the exams ");
      scanf("%d",&marks)
      if (marks>=60) printf("\n\nPassed, promoted to next class");
```

```
            else printf("\n\nPull up next time");
            return 0;
    }
```

Nested `if … else` statement tests multiple cases by placing `if … else` statements inside `if … else` statements. For example, a program that grades exam scores as A for scores greater or equal to 90, B for scores between 80 and 89, C for scores greater or equal to 70, D for scores between 50 and 69, and F for all other scores.

```
    #include<stdio.h>
    int main()
    {
            int score=0;
            printf("Enter students' score in the exams ");
            scanf("%d",&score)
            if (score>=90) printf("\n\nA");
            else if (score>=80) printf("\n\nB");
            else if (score>=70) printf("\n\nC");
            else if (score>=50) printf("\n\nD");
            else printf("\n\nF");
            return 0;
    }
```

The `switch` multiple-selection statement

`Switch` case statements are a substitute for long `if` statements that compare a variable to several "integral" values ("integral" values are simply values that can be expressed as an integer, such as the value of a char). The value of the variable given into switch is compared to the value following each of the cases, and when one value matches the value of the variable, the computer continues executing the program from that point until `break` statement is met.

The syntax is

```
  switch ( <variable> ) {
    case this-value:
      Code to execute if <variable> == this-value
      break;
    case that-value:
      Code to execute if <variable> == that-value
      break;
    ...
    default:
      Code to execute if <variable> does not equal the value following any of the cases
      break;
  }
```

The condition of a `switch` statement is a value. The `case` says that if it has the value of whatever is after that case then do whatever follows the colon. The `break` is used to break out of the case statements. `Break` is a keyword that breaks out of the code block, usually surrounded by braces, as commonly used to exit from a `switch` construct and loops. In this case, `break` prevents the program from falling through and executing the code in all the other case statements.

Example 1, A program that computes and display the area of either circle, rectangle, trapezium or square as chosen by the user.

```
    #include<stdio.h>
    int main()
    {
      int choice;
      float area;
      printf("\n\nChoose the shape to compute the area\n1. Circle, \n2. Rectangle");
      printf("\n3. Trapezium \n4. Square\n");
      scanf("%d",&choice);
      switch (choice){
            case 1:
                    float r;
                    printf("\nArea of a circle\nEnter the radius of the circle: ");
                    scanf("%f",&r);
                    area=(22/7)*r*r;
                    break;
            case 2:
                    int l,w;
                    printf(\nArea of a rectangle\nEnter the length and width of a ");
```

```
              printf("rectangle");
                     scanf("%d%d",&l,&w);
                     area=l*w;
                     break;
              Case 3:
                     int a,b,h;
                     printf("\nArea of trapezium\nEnter the side a, b and h length: ");
                     scanf("%d%d%d",&a,&b,&h);
                     area=0.5*(a+b)*h;
                     break;
              Case 4:
                     int l;
                     printf("\n\nArea of a square\nEnter the length of the square");
                     scanf("%d",&l)
                     area=l*l;
                     break;
              Default:
                     area=0;
                     printf("\nWrong shape chosen";
                     break;
              }
         printf("\n\nThe area of the shape if %.2f",area);
         return 0;
       }
```

Example 2, a program that prompts the user to enter two numbers and operation to be executed, that is, either +, -, /, * or % and display the answer on the screen.

```
       #include<stdio.h>
       int main()
       {
         float no1,no2,ans;
         char operation;
         printf("\n\nEnter the two numbers: ");
         scanf("%f%f",&no1,&no2);
         printf("\nEnter the operation to be performed (either +, -, /, * or %):");
         scanf("%c",&operation);
         switch (operation){
                case '+':
                        ans=no1+no2;
                        break;
                case '-':
                        ans=no1-no2;
                        break;
                case '/':
                        ans=no1/no2;
                        break;
                case '*':
                        ans=no1*no2;
                        break;
                case '%':
                        ans=no1%no2;
                        break;
                default:
                        printf("\n\nWrong operation entered");
                        ans=0;
                        break;
         }
         printf("\n\n%10.2f %c %10.2f = %10.2f",no1,operation,no2,ans);
         return 0;
       }
```

Example 3, Design a program that uses switch structure to capture an alphabetic letter from the user. The program displays the letter and a word that begin with that alphabet.

```
       #include<stdio.h>
       int main()
       {
         char letter;
         printf("\nEnter the alphabetic letter to view the word that begins with it: ");
         scanf("%c",&letter);
         switch (letter){
                case 'a':
                case 'A':
```

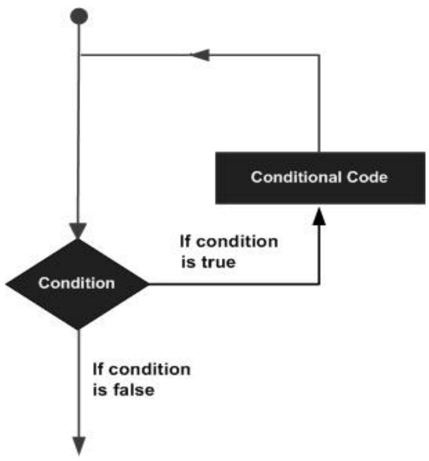
```
                printf("\n\nA for Apple";
                break;
        case 'b':
        case 'B':
                printf("\n\nB for Boy";
                break;
        case 'c':
        case 'C':
                printf("\n\nC for Cat";
                break;
        case 'd':
        case 'D':
                printf("\n\nD for Dog";
                break;
        case 'e':
        case 'E':
                printf("\n\nE for Elephant";
                break;
        case 'f':
        case 'F':
                printf("\n\nF for Fox";
                break;
        case 'g':
        case 'G':
                printf("\n\nG for God";
                break;
        case 'h':
        case 'H':
                printf("\n\nH for Hen";
                break;
        case 'i':
        case 'I':
                printf("\n\nI for Insect";
                break;
        case 'j':
        case 'J':
                printf("\n\nJ for Joy";
                break;
        case 'k':
        case 'K':
                printf("\n\nK for Kettle";
                break;
        case 'l':
        case 'L':
                printf("\n\nL for Lion";
                break;
        case 'n':
        case 'N':
                printf("\n\nN for Niece";
                break;
        case 'm':
        case 'M':
                printf("\n\nM for Mother";
                break;
        case 'o':
        case 'O':
                printf("\n\nO for Orphan";
                break;
        case 'p':
        case 'P':
                printf("\n\nP for Person";
                break;
        case 'Q':
        case 'q':
                printf("\n\nQ for Queen";
                break;
        case 'r':
        case 'R':
                printf("\n\nR for Rat";
                break;
```

```
            case ‘s’:
            case ‘S’:
                    printf(“\n\nS for Sister”;
                    break;
            case ‘t’:
            case ‘T’:
                    printf(“\n\nT for Table”;
                    break;
            case ‘u’:
            case ‘U’:
                    printf(“\n\nU for Uncle”;
                    break;
            case ‘v’:
            case ‘V’:
                    printf(“\n\nV for Vehicle”;
                    break;
            case ‘w’:
            case ‘W’:
                    printf(“\n\nW for Whale”;
                    break;
            case ‘x’:
            case ‘X’:
                    printf(“\n\nX for Xenon”;
                    break;
            case ‘y’:
            case ‘Y’:
                    printf(“\n\nY for Yam”;
                    break;
            case ‘z’:
            case ‘Z’:
                    printf(“\n\nZ for Zebra”;
                    break;
            default:
                    printf(“\n\nWrong entry.”);
                    break;
    }
    return 0;
  }
```

Note: When using `switch` statement, remember that each individual `case` can test only a **`constant integral`** expression. A character constant is represented as a specific character in single quotes, such as, ’A’. Integer constant are simply integer values.

LOOPING STRUCTURES

A loop is a group of instructions the computer executes repeatedly while loop-continuation condition remains true, the flowchart alongside is the general form of a loop statement in most of the programming languages.



There may be a situation, when you need to execute a block of code several number of times. Programming languages provide various control structures that allow for more complicated execution paths.

There are two means of repetition, namely: -

1. Counter-controlled repetition
2. Sentinel-controlled repetition

Counter-controlled repetition (also called definite repletion) are loops that we know how may times the block of statements are to be executed. A control variable is used to count the number of repetitions, it is incremented each time the group of instructions is performed. When the value of the control variable indicates that the correct number of repetitions has been performed, the loop terminates.

Sentinel-controlled repetitions that use sentinel values to control repetition when the number of repetitions is not known in advance. The sentinel value indicates "end of data." The sentinel is entered after all regular data items have been supplied to the program. Sentinels must be distinct from regular data items.

C programming language provides the following types of loop to handle looping requirements.

Loop Type	**Description**
`while`	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
`for`	Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
`do...while`	Like a while statement, except that it tests the condition at the end of the loop body
nested loops	You can use one or more loop inside any another while, for or do..while loop.

Loop Control Statements:

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C supports the following control statements.

Control Statement	**Description**
break	Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch.
continue	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
goto	Transfers control to the labeled statement. Though it is not advised to use `goto` statement in your program.

The Infinite Loop. A loop becomes infinite loop if a condition never becomes false and therefore continues the repetition without exiting. The for loop is traditionally used for this purpose. Since none of the three expressions that form the for loop are required, you can make an endless loop by leaving the conditional expression empty.

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

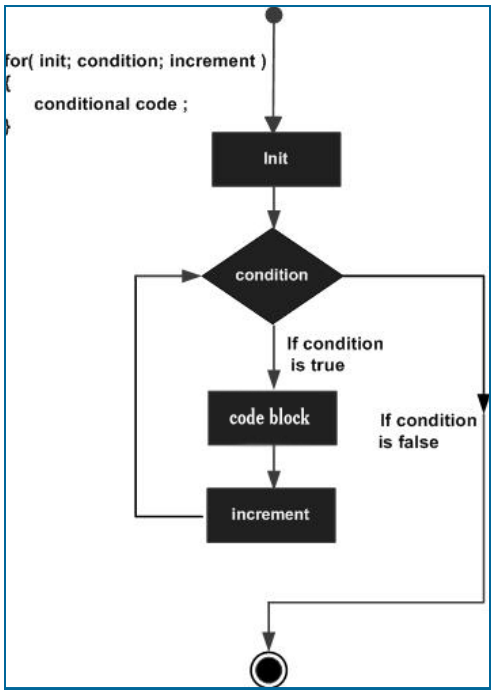
```
        {
           for( ; ; )
           {
              printf("This loop will run forever.\n");
           }
           return 0;
        }
```

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C programmers more commonly use the for(;;) construct to signify an infinite loop.

For loop

A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times. It has the following general syntax

```
        for ( init; condition; increment )
        {
           statement(s);
        }
```



Here is the flow of control in a for loop:

1. The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
2. Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.
3. After the body of the for loop executes, the flow of control jumps back up to the increment statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
4. The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates.

Example 1, Write a program that displays whole numbers from 10-20.

```
        #include <stdio.h>
        int main ()
        {
           int a = 10;
           for(int a=10; a < 20; a++ )
           {
              printf("value of a: %d\n", a);
           }
           return 0;
        }
```

Example 2, Using for loop structure write a c program that display odd number between 11 and 30 in descending order.

```
        #include<stdio.h>
        int main(void)
        {
                printf("\n\nODD NUMBERS BETWEEN 11 -30 IN DESCENDING ORDER\n");
                for (int i=30; i>10; i--){
                        if ((i%2)==1) printf ("\n%d");
                        else continue;
                }
                return 0;
        }
```

Example 3, Write a program that sum up all even numbers from 2 to 100. the program displays the even numbers and their sum.

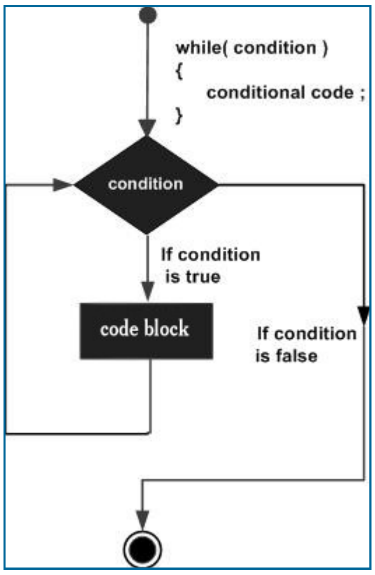
```
        #include<stdio.h>
        int main(void)
        {
                printf("\n\nEVEN NUMBERS BETWEEN 2 AND 100\n");
                int sum=0;
                for (int i=2; i<101; i+=2){
                        printf("%3d",i);
                        sum+=i;
                }
                printf("\n\nThe sum of the even numbers is %4d\n\n",sum);
                return 0;
        }
```

The above program can also be written as follows

```
        #include<stdio.h>
        int main(void)
        {
                printf("\n\nEVEN NUMBERS BETWEEN 2 AND 100\n");
                for (int i=2,sum=0; i<101; i+=2,sum+=i){
                        printf("%3d",i);                  }
                }
                printf("\n\nThe sum of the even numbers is %4d\n\n",sum);
                return 0;
        }
```

While loop

A while loop statement in C programming language repeatedly executes a target statement as long as a given condition is true.



The syntax of a while loop in C programming language is:

```
      Loop_counter_initialization;
      while(condition)
      {
             statement(s);
             Loop_counter_change;
      }
```

Here, **loop_counter_initialization** is done before the loop, **statement (s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any nonzero value. The loop iterates while the condition is true. When the condition becomes false, program control passes to the line immediately following the loop.

Here, key point of the while loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example 1, Write a program that displays whole numbers from 10-20.

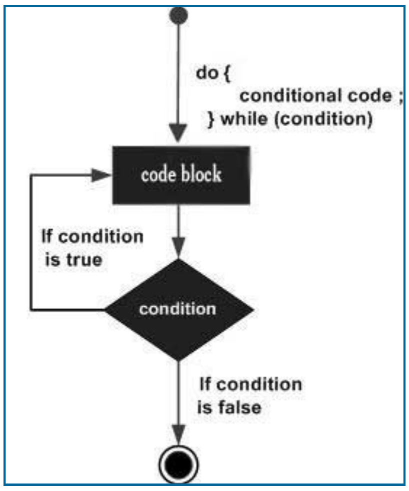
```
      #include <stdio.h>
      int main ()
      {
         /* local variable definition */
         int a = 10;
         /* while loop execution */
         while( a < 20 )
         {
            printf("value of a: %d\n", a);
            a++;
         }
         return 0;
      }
```

Example 2, Write a program that sum up all even numbers from 2 to 100. the program displays the even numbers and their sum.

```
#include<stdio.h>
int main(void)
{
        printf("\n\nEVEN NUMBERS BETWEEN 2 AND 100\n";
        int sum=0, i=2;
        while (i<101){
                printf("%4d",i);
                sum+=i;
                i+=2;
        }
        printf("\n\nThe sum of the even numbers is %d", sum);
        return 0;
}
```

Do … while loop

Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop in C programming language checks its condition at the bottom of the loop.



A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

The syntax of a do...while loop in C programming language is:

```
Loop_counter_initialization;
do
{
      statement(s);
      Loop_counter_change;
}while( condition );
```

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop execute once before the condition is tested.

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes false.

Example 1, write a program that uses do...while to display whole numbers from 10 to 20

```
#include <stdio.h>
int main ()
{
   /* local variable definition */
   int a = 10;
   /* do loop execution */
   do
   {
       printf("value of a: %d\n", a);
       a = a + 1;
   }while( a < 20 );
   return 0;
}
```

Example 2, Using do...while loop structure write a c program that display odd number between 11 and 30 in descending order.

```
#include<stdio.h>
int main(void)
{
        printf("\n\nODD NUMBERS BETWEEN 11 -30 IN DESCENDING ORDER\n");
        int i=30;
        do {
                if ((i%2)==1) printf ("\n%d");
                i--;
        }while(i>10);
        return 0;
}
```

FUNCTIONS IN C

A function is an independent, self-contained section of code that is written to perform a certain task and optionally returns a value to the calling program. Function are invoked by a function call, which specify function name and arguments. There are two types of functions, namely:

1. C standard library functions
2. User-Defined functions (UDF) or programmer-defined function

STANDARD LIBRARY FUNCTIONS

This is a rich collection of prepackaged functions that enable the programmer to perform common mathematical calculations, string manipulations, character manipulations, input/ output and many more useful operations.

The input/ output functions include printf() and scanf() we've been using and are defined in the header file stdio.h. To use the two functions, you MUST include this header file in the program.

The math library functions allow you to perform common mathematical calculations. Math functions are prepackaged in math.h header file. The functions include:

Function	Description	Example
`sqrt(x)`	Square root of x	sqrt(900.00) is 30.0
`exp(x)`	Exponential function e^x	exp(1.0) is 2.718282 exp(2.0) is 7.389056
`log(x)`	Natural logarithm of x (base e)	log(2.718282) is 1.0
`log10(x)`	Logarithm of x (base 10)	log10(1.0) is 0.0 log10(10.0) is 1.0
`fabs(x)`	Absolute value of x	fabs(13.5) is 13.5 fabs(-13.5) is 13.5
`ceil(x)`	Rounds x to smallest integer not less than x	ceil(9.2) is 10.0 ceil(-9.8) is -9.0
`floor(x)`	Rounds x to the largest integer not greater than x	floor(9.2) is 9.0 floor(-9.8) is –10.0
`pow(x, y)`	x raised to the power of y (x^y)	pow(2,7) is 128.0 pow(9, .5) is 3.0
`fmod(x, y)`	Remainder of x/f as a floating point number	fmod(13.657, 2.333) is 1.992
`sin(x)`	Trigonometric sine of x (x in radians)	sin(0.0) is 0.0
`cos(x)`	Trigonometric cosine of x (x in radians)	cos(0.0) is 1.0
`tan(x)`	Trigonometric tangent of x (x in radians)	tan(0.0) is 0.0

The string and character manipulation function to be discussed in the topic strings.

USER-DEFINED FUNCTIONS

These are functions coded by the programmer to modularize/ structure a program. All variable defined in a function definition are called local variables.

Most functions have a list of parameters that provide means for communicating information between functions. A function's parameters are also local variables. Why use functions?

1. Enable programmer to divide-and-conquer large programmers by making them more manageable
2. Functions enhance software reusability. Existing functions are used as building blocks to create new programs.
3. To avoid repeating codes in a program. Packaging a code as a function allows the code to be executed

from several locations in a program simply by calling the function.

A function declaration also called function prototype tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts:

```
return_type function_name( parameter list );
```

The basic syntax of function definition is,

```
return_type function_name( parameter list )
{
   body of the function
}
```

Note: The function header is similar to function prototype except it has no semi colon. A function prototype provides the name and arguments of the function. A function definition in C programming language consists of a function header and a function body. Here are all the parts of a function:

- **Return Type**: A function may return a value. The **`return_type`** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the **`return_type`** is the keyword **`void`**.
- **Function Name**: This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters**: A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body**: The function body contains a collection of statements that define what the function does.

Example 1: Following is the source code for a function called max(). This function takes two parameters num1 and num2 and returns the maximum between the two:

```
#include <stdio.h>
int max(int num1, int num2); /* function declaration */
int main ()
{
   /* local variable definition */
   int a = 100;
   int b = 200;
   int ret = max(a, b); /* calling a function to get max value */
   printf( "Max value is : %d\n", ret );
   return 0;
}
/* function returning the max between two numbers */
int max(int num1, int num2)
{
   /* local variable declaration */
   int result;
   if (num1 > num2) result = num1;
   else result = num2;
   return result;
}
```

Example 2: a program that uses a function square to calculate and print squares of the integers from 1 to 10.

```
#include <stdio.h>
int square(int y); /* function prototype */
int main () /*main function begins program execution*/
{
   /* local variable definition */
```

```
        int x; /* loop counter*/
        For (x=1; x<=10; x++){
           Printf("\n%d squared is %d",x,square(x));
        }
        printf( "\n");
        return 0;
     }
     /* function returning the square of a given whole number */
     int square(int y)
     {
        return (y*y);
     }
```

Function Arguments:

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function:

Call Type	Description
Call by value	This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
Call by reference	This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

By default, C uses **call by value** to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max() function used the same method.

The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

To pass the value by reference, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function swap (), which exchanges the values of the two integer variables pointed to by its arguments.

```
#include <stdio.h>

/* function declaration */
void swap(int *x, int *y);

int main ()
{
   /* local variable definition */
   int a = 100;
   int b = 200;

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

   /* calling a function to swap the values.
    * &a indicates pointer to a ie. address of variable a and
    * &b indicates pointer to b ie. address of variable b.
   */
   swap(&a, &b);
   printf("After swap, value of a : %d\n", a );
   printf("After swap, value of b : %d\n", b );

   return 0;
}
/* function definition to swap the values */
void swap(int *x, int *y)
{
   int temp;
   temp = *x;     /* save the value at address x */
   *x = *y;       /* put y into x */
   *y = temp;     /* put temp into y */
 }
```

Scope Rules

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable can not be accessed. There are three places where variables can be declared in C programming language:

1. Inside a function or a block which is called **local** variables,
2. Outside of all functions which is called **global** variables.
3. In the definition of function parameters which is called **formal** parameters.

Local Variables

Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. Following is the example using local variables. Here all the variables a, b and c are local to main () function.

```
#include <stdio.h>

int main ()
{
  /* local variable declaration */
  int a, b;
  int c;

  /* actual initialization */
  a = 10;
  b = 20;
  c = a + b;

  printf ("value of a = %d, b = %d and c = %d\n", a, b, c);

  return 0;
}
```

Global Variables

Global variables are defined outside of a function, usually on top of the program. The global variables will hold their value throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. Following is the example using global and local variables:

```
#include <stdio.h>

/* global variable declaration */
int g;
int main ()
{
  /* local variable declaration */
  int a, b;

  /* actual initialization */
  a = 10;
  b = 20;
  g = a + b;

  printf ("value of a = %d, b = %d and g = %d\n", a, b, g);

  return 0;
}
```

A program can have same name for local and global variables but value of local variable inside a function will take preference. Following is an example:

```
#include <stdio.h>

/* global variable declaration */
int g = 20;

int main ()
{
  /* local variable declaration */
  int g = 10;

  printf ("value of g = %d\n",  g);

  return 0;
}
```

Formal Parameters

Function parameters, formal parameters, are treated as local variables with-in that function and they will take preference over the global variables. Following is an example:

```
#include <stdio.h>

/* global variable declaration */
int a = 20;

int main ()
{
  /* local variable declaration in main function */
  int a = 10;
  int b = 20;
  int c = 0;

  printf ("value of a in main() = %d\n",  a);
  c = sum( a, b);
  printf ("value of c in main() = %d\n",  c);

  return 0;
}

/* function to add two integers */
int sum(int a, int b)
{
    printf ("value of a in sum() = %d\n",  a);
    printf ("value of b in sum() = %d\n",  b);
    return a + b;
}
```

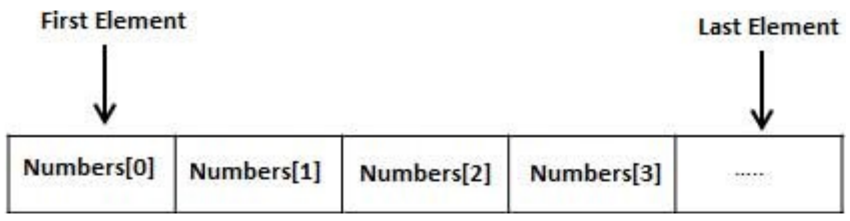
DATA STRUCTURES

1. Arrays

C programming language provides a data structure called the array, which can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows:

```
type arrayName [ arraySize ];
```

This is called a single-dimensional array. The arraySize must be an integer constant greater than zero and type can be any valid C data type. For example, to declare a 10-element array called balance of type double, use this statement:

```
double balance[10];
```

Now balance is a variable array which is sufficient to hold upto 10 double numbers.

Initializing Arrays

You can initialize array in C either one by one or using a single statement as follows:

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets [ ].

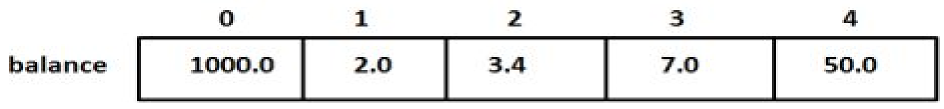
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

```
double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array:

```
balance[4] = 50.0;
```

The above statement assigns element number 5th in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called base index and last index of an array will be total size of the array minus 1. Following is the pictorial representation of the same array we discussed above:



Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:

```
double salary = balance[9];
```

The above statement will take 10th element from the array and assign the value to salary variable. Following is an example which will use all the above mentioned three concepts viz. declaration, assignment and accessing arrays:

```
#include <stdio.h>

int main ()
{
   int n[ 10 ]; /* n is an array of 10 integers */
   int i,j;

   /* initialize elements of array n to 0 */
   for ( i = 0; i < 10; i++ )
   {
      n[ i ] = i + 100; /* set element at location i to i + 100 */
   }

   /* output each array element's value */
   for (j = 0; j < 10; j++ )
   {
      printf("Element[%d] = %d\n", j, n[j] );
   }

   return 0;
}
```

<u>Multi-dimensional Arrays</u>

C programming language allows multidimensional arrays. Here is the general form of a multidimensional array declaration:

```
type name[size1][size2]...[sizeN];
```

For example, the following declaration creates a three dimensional 5 . 10 . 4 integer array:

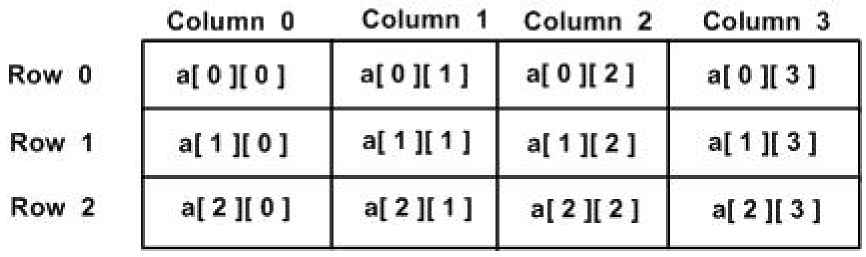
```
int threedim[5][10][4];
```

Two-Dimensional Arrays:

The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size x,y you would write something as follows:

```
type arrayName [ x ][ y ];
```

Where type can be any valid C data type and arrayName will be a valid C identifier. A two-dimensional array can be think as a table which will have x number of rows and y number of columns. A 2-dimensional array a, which contains three rows and four columns can be shown as below:



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in array a is identified by an element name of the form a[i][j], where a is the name of the array, and i and j are the subscripts that uniquely identify each element in a.

Initializing Two-Dimensional Arrays:

Multidimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

```
int a[3][4] = {
 {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
 {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
 {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to previous example:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Accessing Two-Dimensional Array Elements:

An element in 2-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example:

```
int val = a[2][3];
```

The above statement will take 4th element from the 3rd row of the array. You can verify it in the above diagram. Let us check below program where we have used nested loop to handle a two dimensional array:

```
#include <stdio.h>

int main ()
{
   /* an array with 5 rows and 2 columns*/
   int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
   int i, j;

   /* output each array element's value */
   for ( i = 0; i < 5; i++ )
   {
      for ( j = 0; j < 2; j++ )
      {
         printf("a[%d][%d] = %d\n", i,j, a[i][j] );
      }
   }
   return 0;
}
```

Passing Arrays as Function Arguments

If you want to pass a single-dimension array as an argument in a function, you would have to declare function formal parameter in one of following three ways and all three declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received. Similar way you can pass multi-dimensional array as formal parameters.

Way-1

Formal parameters as a pointer as follows. You will study what is pointer in next chapter.

```
void myFunction(int *param)
{
.
.
.
}
```

Way-2

Formal parameters as a sized array as follows:

```
void myFunction(int param[10])
{
.
.
```

```
        .
        }
```

Way-3

Formal parameters as an unsized array as follows:

```
        void myFunction(int param[])
        {
        .
        .
        .
        }
```

Example

Now, consider the following function, which will take an array as an argument along with another argument and based on the passed arguments, it will return average of the numbers passed through the array as follows:

```
#include <stdio.h>

/* function declaration */
double getAverage(int arr[], int size);

int main ()
{
   /* an int array with 5 elements */
   int balance[5] = {1000, 2, 3, 17, 50};
   double avg;

   /* pass pointer to the array as an argument */
   avg = getAverage( balance, 5 ) ;

   /* output the returned value */
   printf( "Average value is: %f ", avg );

   return 0;
}
double getAverage(int arr[], int size)
{
  int    i;
  double avg;
  double sum;

  for (i = 0; i < size; ++i)
  {
    sum += arr[i];
  }

  avg = sum / size;

  return avg;
}
```

POINTERS

Pointers in C are easy and fun to learn. Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. So it becomes necessary to learn pointers to become a perfect C programmer. Let's start learning them in simple and easy steps.

As you know, every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which will print the address of the variables defined:

```
#include <stdio.h>
int main ()
{
   int  var1;
   char var2[10];
   printf("Address of var1 variable: %x\n", &var1  );
   printf("Address of var2 variable: %x\n", &var2  );
   return 0;
}
```

What Are Pointers?
A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before you can use it to store any variable address. The general form of a pointer variable declaration is:

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. The asterisk * you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration:

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

How to use Pointers?

There are few important operations, which we will do with the help of pointers very frequently. **(a)** we define a pointer variable **(b)** assign the address of a variable to a pointer and **(c)** finally access the value at the address available in the pointer variable. This is done by using unary operator ***** that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations:

```
#include <stdio.h>
int main ()
{
   int  var = 20;   /* actual variable declaration */
   int  *ip;        /* pointer variable declaration */
   ip = &var;  /* store address of var in pointer variable*/
   printf("Address of var variable: %x\n", &var  );
   /* address stored in pointer variable */
   printf("Address stored in ip variable: %x\n", ip );
   /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );
   return 0;
}
```

STRUCTURES

Structures are used when you want to process data of multiple data types but you still want to refer to the data as a single entity. A Structure is a data type suitable for grouping data elements together. Lets create a new data structure suitable for storing the date. The elements or fields which make up the structure use the four basic data types.

The syntax for structure declaration,

```
struct stracturename{
      //structure data
};
```

For instance,

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

This declares a NEW data type called date. This date structure consists of three basic data elements, all of type integer. This is a definition to the compiler. It does not create any storage space and cannot be used as a variable. In essence, its a new data type keyword, like int and char, and can now be used 84 to create variables. Other data structures may be defined as consisting of the same composition as the date structure,

```
struct  date   todays_date;
```

defines a variable called ***todays_date*** to be of the same data type as that of the newly defined data type struct ***date.***

ASSIGNING VALUES TO STRUCTURE ELEMENTS

To assign todays date to the individual elements of the structure todays_date, the statement

```
todays_date.day = 21;
todays_date.month = 07;
todays_date.year = 1985;
```

is used. NOTE the use of the **.** element to reference the individual elements within todays_date.

Complete program,

```
#include <stdio.h>
struct date { /* global definition of type date */
      int month;
      int day;
      int year;
};
main()
{
      struct date  today;
      today.month = 10;
      today.day = 14;
      today.year = 1995;
      printf("Todays date is %d/%d/%d.\n", \
      today.month, today.day, today.year );
}
```

INITIALIZING STRUCTURES

This is similar to the initialization of arrays; the elements are simply listed inside a pair of braces, with each element separated by a comma. The structure declaration is preceded by the keyword static

```
static struct date today = { 4,23,1998 };
```

ARRAYS OF STRUCTURES

Consider the following,

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

Lets now create an array called birthdays of the same data type as the structure date

```
struct date birthdays[5];
```

This creates an array of 5 elements which have the structure of date.

```
birthdays[1].month = 12;
birthdays[1].day   = 04;
birthdays[1].year  = 1998;
--birthdays[1].year;
```

For example

```
#include <stdio.h>
struct  date  {        /* Global definition of date */
      int day, month, year;
};
main()
{
      struct date dates[5];
      int i;
      for( i = 0; i < 5; ++i ) {
        printf("Please enter the date (dd:mm:yy)" );
        scanf("%d:%d:%d", &dates[i].day, &dates[i].month,
        &dates[i].year );
      }
}
```

STRUCTURES AND ARRAYS

Structures can also contain arrays.

```
struct month {
      int  number_of_days;
      char name[4];
};
static struct month this_month = { 31, "Jan" };
this_month.number_of_days = 31;
strcpy( this_month.name, "Jan" );
printf("The month is %s\n", this_month.name );
```

Note that the array name has an extra element to hold the end of string nul character.

STRUCTURES WHICH CONTAIN STRUCTURES

Structures can also contain structures. Consider where both a date and time structure are combined into a single structure called date_time, eg,

```
struct date {
      int  month, day, year;
};
struct time {
      int  hours, mins, secs;
};
struct date_time {
      struct date sdate;
      struct time stime;
};
```

This declares a structure whose elements consist of two other previously declared structures. Initialization could be done as follows,

```
static struct date_time today = { { 2, 11, 1985 }, { 3, 3,33 } };
```

which sets the sdate element of the structure today to the eleventh of February, 1985. The stime element of the structure is initialized to three hours, three minutes, thirty-three seconds. Each item within the structure can be referenced if desired, eg,

```
++today.stime.secs;

if( today.stime.secs == 60 ) ++today.stime.mins;
```

Example 2, you might want to process information on students in the categories of name and marks (grade percentages).

```
#include <stdio.h>
struct student{
    char name[30];
    float marks;
}   student1, student2;
int main (void)
{
    struct student student3;
    char s1[30];
    float f;
    scanf ("%s", name);
    scanf (" %f", & f);
    student1.name = s1;
    student2.marks = f;
    printf (" Name is %s \n", student1.name);
    printf (" Marks are %f \n", student2.marks);
}
```

Example 3, You can define structures of arrays or arrays of structures, etc. The following section gives definitions of complex structures.

Program

```
#include <stdio.h>
struct address      \\ A
{
      char  plot[30];
      char struc[30];
      char city[30]
}
struct student      \\ B
{
    char name [30];
    float marks ;
    struct address adr;     \\ C
}
int main ( )
{
    struct student student1; \\ D
    struct student class[20];      \\ E
    class[1].marks = 70;     \\ F
    class[1].name = " Anil ";
    class[1].adr.plot = "7 ";        \\ G
    class[1].adr.street = " Mg Road";
    class[1].adr.city = "mumbai";

    printf( " Marks are %d\n", class[1].marks);
    printf( " name are %s\n", class[1].name);
    printf( " adr.plot is %s\n", class[1].adr.plot);
    printf( " adr.street is %s\n", class[1].adr.stret);
    printf( " adr.city is %s\n", class[1].adr.city);
```

```
        }
```

Explanation

Statement A declares the address of a structure containing the members plot, street and city.

Statement B declares a structure having 3 members: name, marks, and adr. The data type of adr is structure address, which is given by statement C.

Statement D defines the variable student1 of the data type struct student.

Statement E defines an array class with 20 elements. Each element is a structure.

You can refer to marks of the students of class[1] using the notation class[1].marks. class[1] indicates the first element of the array, and since each element is a structure, a member can be accessed using dot notation.

You can refer to the plot of a student of class[1] using the notation class[1].adr.plot. Since the third element of the structure is adr, and plot is a member of adr, you can refer to members of the nested structures.

If you want to refer to the first character of the character array plot, then you can refer it as

```
        Class[1].adr.plot[0]
```

because plot is a character array.

Example 4. Write a program in C that prompts the user for todays date, calculates tomorrows date, and displays the result. Use structures for todays date, tomorrows date, and an array to hold the days for each month of the year. Remember to change the month or year as necessary.

FILES AND STREAMS

A **file** can refer to a disk file, a terminal, a printer, or a tape drive. In other words, a file represents a concrete device with which you want to exchange information. Before you perform any communication to a file, you have to open the file. Then you need to close the opened file after you finish exchanging information with it.

The data flow you transfer from your program to a file, or vice versa, is called a **stream**, which is a series of bytes. There are two formats of streams: -

1. **Text (sequential) stream**, which consists of a sequence of characters (that is, ASCII data). Depending on the compilers, each character line in a text stream may be terminated by a newline character. Text streams are used for textual data, which has a consistent appearance from one environment to another, or from one machine to another.
2. **Binary stream**, which is a series of bytes. The content of an .exe file would be one example. Binary streams are primarily used for non-textual data, which is required to keep the exact contents of the file.

In C, a memory area, which is temporarily used to store data before it is sent to its destination, is called a **buffer**. With the help of buffers, the operating system can improve efficiency by reducing the number of accesses to I/O devices (that is, files).

By default, all I/O streams are buffered. The buffered I/O is also called the high-level I/O. Accordingly, the low-level I/O refers to the unbuffered I/O.

Handling Files

The FILE structure is the file control structure defined in the header file stdio.h. Let's focus on how to open and close a disk data file and how to interpret error messages returned by I/O functions.

- **A pointer of type FILE is called a file pointer**, which references a disk file. A file pointer is used by a stream to conduct the operation of the I/O functions. For instance, the following defines a file pointer called fptr:

  ```
  FILE *fptr;
  ```

  In the FILE structure there is a member, called the file position indicator, that points to the position in a file where data will be read from or written to.

- **To open a file**, use the function fopen(). The function enables you to open a file and associate a stream to the opened file. The syntax for the fopen() function is

  ```
  #include <stdio.h>
  FILE *fopen(const char *filename, const char *mode);
  ```

  Here filename is a char pointer that references a string of a filename. The filename is given to the file that is about to be opened by the fopen() function. mode points to another string that specifies the way to open the file. The fopen() function returns a pointer of type FILE. If an error occurs during the procedure to open a file, the fopen() function returns a null pointer.

  The following list shows the possible ways to open a file by various strings of modes:

  "r" opens an existing text file for reading.

  "w" creates a text file for writing.

  "a" opens an existing text file for appending.

  "r+" opens an existing text file for reading or writing.

  "w+" creates a text file for reading or writing.

  "a+" opens or creates a text file for appending.

  "rb" opens an existing binary file for reading.

  "wb" creates a binary file for writing.

  "ab" opens an existing binary file for appending.

  "r+b" opens an existing binary file for reading or writing.

  "w+b" creates a binary file for reading or writing.

"a+b" opens or creates a binary file for appending.

Note that you might see people use the mode "rb+" instead of "r+b". These two strings are equivalent. Similarly, "wb+" is the same as "w+b"; "ab+" is equivalent to "a+b".

The following statements try to open a file called test.txt:

```
FILE *fptr;
if ((fptr = fopen("test.txt", "r")) == NULL){
   fprintf(stderr,"Cannot open test.txt file.\n");
   perror(“because”); //outputs error description
   exit(1);
}
```

Here "r" is used to indicate that a text file is about to be opened for reading only. If an error occurs when the fopen() function tries to open the file, the function returns a null pointer. Then an error message is printed out by the printf() function and the program is aborted by calling the exit() function with a nonzero value.

- **Closing a file**. After a disk file is read, written, or appended with some new data, you have to disassociate the file from a specified stream by calling the fclose() function. The syntax for the fclose() function is

```
#include <stdio.h>
int fclose(FILE *stream);
```

If fclose() closes a file successfully, it returns 0. Otherwise, the function returns EOF. Normally, the fclose() function fails only when the disk is removed before the function is called or there is no more space left on the disk.

Since all high-level I/O operations are buffered, the fclose() function flushes data left in the buffer to ensure that no data will be lost before it disassociates a specified stream with the opened file.

Note that a file that is opened and associated with a stream has to be closed after the I/O operation. Otherwise, the data saved in the file may be lost; some unpredictable errors might occur during the execution of your program.

INPUTTING/OUTPUTTING SINGLE CHARACTERS

Single characters may be read/written with files by use of the two functions, getc(), and putc().

```
int ch;
ch = getc( input_file );   /*  assigns character to ch  */
```

The getc() also returns the value EOF (end of file), so

```
while( (ch = getc( input_file )) != EOF )
      ......................
```

NOTE that the putc/getc are similar to getchar/putchar except that arguments are supplied specifying

the I/O device.

```
putc('\n', output_file ); /* writes a newline to output file */
```

TESTING FOR THE End Of File TERMINATOR (feof)

This is a built in function incorporated with the stdio.h routines. It returns 1 if the file pointer is at the end of the file.

```
if( feof(input_file)) printf("Ran out of data.\n");
```

THE fprintf AND fscanf STATEMENTS

These perform the same function as printf and scanf, but work on files. Consider,

```
fprintf(output_file, "Now is the time for all..\n");
fscanf(input_file, "%f", &float_value);
```

THE fgets AND fputs STATEMENTS

These are useful for reading and writing entire lines of data to/from a file. If buffer is a pointer to a character array and n is the maximum number of characters to be stored, then

```
fgets(buffer, n, input_file);
```

will read an entire line of text (max chars = n) into buffer until the newline character or n=max, whichever occurs first. The function places a NULL character after the last character in the buffer. The function will be equal to a NULL if no more data exists.

```
fputs(buffer, output_file);
```

writes the characters in buffer until a NULL is found. The NULL character is not written to the output_file.

NOTE: fgets does not store the newline into the buffer, fputs will append a newline to the line written to the output file.

For example, The following program creates a file called NAMES.DAT. The program writes five names to a disk file using fputs():

```
#include <stdio.h>
FILE *fp;
main()
{
      fp = fopen(''NAMES.DAT", "w");  /* Creates a new file */
      fputs("Michael Langston\n", fp);
      fputs("Sally Redding\n", fp);
      fputs("Jane Kirk\n", fp);
      fputs("Stacy Grady\n", fp);
      fputs("Paula Hiquet\n", fp);
      fclose(fp);                      /* Release the file */
      return 0;
}
```

Example 2, The following file writes the numbers from 1 to 100 to a file called NUMS.txt

```
#include <stdio.h>
FILE *fp;
main()
{
   int ctr;
   fp = fopen("NUMS. 1", "wt");   /* Creates a new file */
   if (fp == NULL) {
      printf("Error opening file.\n");
}else {
      for (ctr=1; ctr<101; ctr++) fprintf(fp, ''%d ", ctr);
}
   fclose(fp);
   return 0;
}
```

Adding to a File

You can easily add data to an existing file or create new files by opening the file in append access mode. Data files on the disk rarely are static; they grow almost daily due to (with luck!) increased business. Being able to add to data already on the disk is very useful indeed.

Example, The following program adds three more names to the NAMES.DAT file created in an earlier example.

```
#include <stdio.h>
FILE *fp;
int main()
```

```
{
   fp = fopen("NAMES.DAT", "a");       // Adds to file
   fputs("Johnny Smith\n", fp);
   fputs("Laura Hull\n", fp);
   fputs("Mark Brown\n", fp);
   fclose(fp);                         // Release the file
   return 0;
}
```

Reading from a File

As soon as the data is in a file, you must be able to read that data. You must open the file in a read access mode. There are several ways to read data. You can read character data a character at a time or a string at a time. The choice depends on the format of the data. If you stored numbers using fprintf(), you might want to use a mirror-image fscanf() to read the data.

Files you open for read access (using "r", "rt", and "rb") must exist already, or C gives you an error. You cannot read a file that does not exist. fopen() returns NULL if the file does not exist when you open it for read access.

Another event happens when reading files. Eventually, you read all the data. Subsequent reading produces errors because there is no more data to read. C provides a solution to the end-of-file occurrence. If you attempt to read from a file that you have completely read the data from, C returns the value EOF, defined in stdio.h. To find the end-of-file condition, be sure to check for EOF when performing input from files.

For example, This program asks the user for a filename and prints the contents of the file to the screen. If the file does not exist, the program displays an error message.

```
#include<stdio.h>
FILE *fp;
main()
{
   char filename[12];              // Holds user's filename
   int inChar;                           // Input character
   printf("What is the name of the file you want to see? ");
   gets(filename);
   if ((fp=fopen(filename, "r"))==NULL)
     { printf("\n\n*** That file does not exist ***\n");
       exit();                         // Exits program
     }
   while ((inChar = getc(fp)) != EOF)       /* Reads first */
      {
           putchar(inChar);
      }                   /* character   */
      fclose(fp);
   return 0;
}
```

Random File Records

Random files exemplify the power of data processing with C. Sequential file processing is slow unless you read the entire file into arrays and process them in memory. Random files provide you a way to read individual pieces of data from a file in any order needed and process them one at a time .

Generally, you read and write file *records.* A record to a file is analogous to a C structure. A record is a collection of one or more data values (called *fields*) that you read and write to disk. Generally, you store data in structures and write the structures to disk, where they are called records. When you read a record from disk, you generally read that record into a structure variable and process it with your program.

Unlike some other programming languages, not all C-read disk data has to be stored in record format. Typically, you write a stream of characters to a disk file and access that data either sequentially or randomly by reading it into variables and structures.

The process of randomly accessing data in a file is simple. Consider the data files of a large credit card organization. When you make a purchase, the store calls the credit card company to get an authorization. Millions of names are in the credit card company's files. There is no quick way the credit card company could read every record sequentially from the disk that comes before yours. Sequential files do not lend themselves to quick access. In many situations, looking up individual records in a data file with sequential access is not feasible.