



UNESCO-NIGERIA TECHNICAL & VOCATIONAL
EDUCATION REVITALISATION PROJECT-PHASE
II



NATIONAL DIPLOMA IN COMPUTER TECHNOLOGY



Computer Programming

COURSE CODE: COM113

YEAR I- SEMESTER I

THEORY

Version 1: December 2008

Table of Contents

WEEK 1	Concept of programming	6
	Features of a good computer program	7
	System Development Cycle	9
WEEK 2	Concept of Algorithm	11
	Features of an Algorithm	11
	Methods of Representing Algorithm	11
	Pseudo code	12
WEEK 3	English-like form	15
	Flowchart	16
WEEK 4	Decision table.....	20
	Data flow Diagrams	22
WEEK 5	The flowchart.....	25
	ADVANTAGES OF USING FLOWCHART	33
	DISADVANTAGES OF USING FLOWCHART.....	33
WEEK 6	Designing Algorithm For Common Programming Logic Structures	35
	Simple Sequence.....	35
	Selection.....	35
	Iteration Logic (Repetitive flow)	37
WEEK 7	The Concept of Modular Programming	39
	Modular program planning	39
	Modular Hierarchy plan for the problem	43
WEEK 8	STAGES OF PROGRAM DEVELOPMENT.....	45
	Problem definition:	45
	Develop the algorithm.....	45
	Plan the logic of the program/flowcharting:	45
	Write the computer program:.....	45
	Type the program into computer:	46
	Test and debug the program:.....	46
	Document the work:.....	46
	Program development/execution process	47
WEEK 9	LEVELS OF COMPUTER PROGRAMMING LANGUAGES.....	49
	Low-level Languages.....	49
WEEK 10	Machine language	49

Assembly Language.....	50
High-level languages	51
WEEK 11 THE CONCEPT OF DEBUGGING AND MAINTAINING PROGRAM.....	59
Sources of bugs in a program.....	59
Preventing Bugs	60
Methods of debugging	60
Understand the Problem.....	60
Basic debugging techniques/steps.....	61
Recognize a bug exists.....	61
Isolate source of bug	61
Identify cause of bug.....	62
Determine fix for bug.....	62
Fix and test.....	63
Categories of Program maintenance	66
WEEK 12 THE CONCEPT OF GOOD PROGRAMMING PRACTICE.....	68
Structured Coding Guidelines.....	69
Flow of Control.....	69
Do...Loop Statement	70
Syntax	70
For...Next Statement	71
Using Comments in Code	75
Using Descriptive Names for Variables, Constants and Functions	75
Using Pseudo-code in Comments	75
Using Modular Coding	76
WEEK 13 Program documentation concepts.....	78
Program Design	79
Program Debugging	79
Program Modifications	80
WEEK 15 The Visual Basic environment.....	90
The properties window :	91
Starting Visual Basic.....	93
Stopping Visual Basic.....	93
Getting online help.....	93
The Help Menu	96

Opening Application..... 97
Creating Simple application (Wizard) 98
Running your application..... 104
Creating Executable File..... 104
Saving your application 105
List of Computer Programming Languages..... 106

WEEK 1

SPECIFIC LEARNING OUTCOMES

To understand:

- Concept of programming
- Features of a good program
- Systems development cycle.

Concept of programming

A program is a set of instructions that tells the computer what to do. Computer programming (often shortened to programming or coding), is the process of writing, testing, debugging/troubleshooting and maintaining act of instructions (source code) for solving a problem with the computer. A source code is written in an acceptable computer programming language. The code may be a modification of an existing source or something completely new.

The purpose of programming is to create a program that exhibits a certain described behavior (customization). The process of writing source code requires expertise in many different subjects, including knowledge of the application domain. Alternatively, Programming is the craft of transforming requirements into something that a computer can execute. Problem solving on computer is a task of expressing the solution to the problem in terms of simple concepts, operations and computer code (program) to obtain the results. To achieve this aim, you may proceed as follows.

1. First, understand the problem clearly:- Decide what you want to be calculated by the computer. What will be the input data required? (if any). This is the problem formulation.
2. Write the steps of computation that are necessary to arrive at the solution. This is setting up the algorithm.
3. Prepare a flowchart corresponding to the algorithm.
4. Develop the computer program. Test and run it on the computer.

There is an ongoing debate on the extent to which the writing of programs is an art, a craft or an engineering discipline. Good programming is generally considered to be the measured application art, craft and engineering, with the goal of producing an efficient and maintainable software (program) solution. The discipline differs from many other technical professions in that programmers generally do not need to be licensed or pass any standardized (or governmentally regulated) certification tests in order to call themselves “programmers” or even “software engineers”.

Features of a good computer program

1) **Reliability**

Any developed program for a particular application can be depended upon to do what it is supposed to accomplish. How often the results of a program are correct. This depends on prevention of resulting from data conversion and prevention of errors resulting from buffer overflows, underflows and zero division.

2) **Meeting Users Needs:**

Any developed system has a purpose for which it is developed. A developed program is a failure if it cannot meet the objectives for which it is proposed and designed, that is, if the potential users cannot use it either because it is too complex or too difficult. The usability of an application analysis involving the user.

3) **Development on time within Budgets:**

Estimates of time and cost for writing computer programs have frequently been under or over estimated. The components of a structured disciplined approach to programming are:

- i) Proper control and management of time and cost required.
- ii) Increased programmer productivity
- iii) More accurate estimates.

4) **Error-Free Set of Instruction**

Almost all large set of programs contain errors. If a program is designed and developed in a disciplined structured approach, it minimizes the likelihood of errors and facilitates detection/correction of such errors during program testing.

5) **Error-Resistant Operations:**

A good program should be designed in such a way that it can perform validation run on each input data to determine whether or not they meet the criteria set for them. Eg Reasonableness check, Existence check, Dependency check, etc.

6) **Maintainable Code:**

A good program design will always be easy to change or modify when the need arises. Programs should be written with the maintenance activity in mind. The structure, coding and

documentation of the program should allow another programmer to understand the logic of the program and to make a change in one part of a program without unknowingly introducing an error in another part of the same program.

7) **Portable Code:**

A good program design will be transferable to a different computer having a language translator for that language without substantial changes or modification

8) **Readability:**

The program codes will be easy for a programmer to read and understand the logic involved in the programming.

9) **Storage Saving:**

A good program design is not to be verbous, that is, it will not be allowed to be unnecessary long, thereby consuming much storage that will be required for processing data and storage of information produced from processing.

10) **Efficiency:**

The amount of system resources a program consumes (processor time, memory space, slow devices, network bandwidth and to some extent even user interaction), the less the better.

11) **Robustness:**

How well a program anticipates situations of data type conflict and other incompatibilities that result in run time errors and program halts. The focus is mainly on user interaction and handling of

12) **Usability:**

The clarity and intuitiveness of a programs output can make or break it's success. This involves a wide range of textual and graphical elements that makes a program easy and comfortable to use.

System Development Cycle

Most IT projects work in cycles. First, the needs of the computer users must be analyzed. This task is often performed by a professional Systems Analysts who will ask the users exactly what they would like the system to do, and then draw up plans on how this can be implemented on a real computer based system.

The programmer will take the specifications from the Systems Analyst and then convert the broad brushstrokes into actual computer programs. Ideally at this point there should be testing and input from the users so that what is produced by the programmers is actually what they asked for.

Finally, there is the implementation process during which all users are introduced to the new systems, which often involves an element of training.

Once the users start using the new system, they will often suggest new improvements and the whole process is started all over again.

These are methodologies for defining a systems development cycle and often you will see four key stages, as listed below.

Feasibility Study

Design

Programming

Implementation

WEEK 2

SPECIFIC LEARNING OUTCOMES

To understand:

- The Concept of Algorithm
- Definition of Algorithm
- Features of an Algorithm
- Methods of representing Algorithm

Concept of Algorithm

An algorithm is a set of instructions to obtain the solution of a given problem. Computer needs precise and well-defined instructions for finding solution of problems. If there is any ambiguity, the computer will not yield the right results. It is essential that all the stages of solution of a given problem be specified in details, correctly and clearly moreover, the steps must also be organized rightly so that a unique solution is obtained.

A typical programming task can be divided into two phases:

(a) Problem solving phase

In this stage an ordered sequence of steps that describe solution of the problem is produced.

Their sequence of steps can be called anti-Algorithm

(b) Implementation Phase

In this phase, the program is implemented in some programming languages.

Algorithm may be set up for any type of problems, mathematical/scientific or business. Normally algorithms for mathematical and scientific problems involve mathematical formulars. Algorithms for business problems are generally descriptive and have little use of formula.

Features of an Algorithm

1. It should be simple
2. It should be clear with no ambiguity
3. It should head to unique solution of the problem
4. It should involve a finite number of steps to arrive at a solution
5. It should have the capability to handle unexpected situation.

Methods of Representing Algorithm

Algorithms are statements of steps involved in solving a particular problem. The steps to the solutions are broken into series of logical steps in English related form. Programs are written to solve real life problems.

There can't be a solution if there is no recognized problem and once a problem exist, one must take certain step in order to get a desired solution. The following methods could be used to represent an algorithm.

- Methods of English like form
- Methods of Flowchart
- Methods of Pseudo code
- Methods of Decision table
- Methods of Data flow Diagram (DFD)

Pseudo code

A pseudo code is the English-like representation of the program logic. It does not make use of standard symbols like the flowchart. It is a sequential step by step arrangements of the instructions to be performed to accomplish a task. It is an informal and artificial language that helps programmers develop algorithms.

Example 1

Write a pseudo code for findings the area of a room.

Solution:

- Begin process
- Input room length
- Input room width
- Multiply length by width to get area
- Print area
- End process

Solution for example 3 (below under pseudo code)

- Step 1: Input M1, M2, M3, M4
- Step 2: $\text{Grade} \leftarrow (M1 + M2 + M3 + M4)/4$
- Step 3: If (Grade < 50) then
- Print "FALL"
- Print "Pass"
- End it.

Example 2

Write a Pseudo code for finding the greatest of 3 numbers represented as A, B, and C.

Solution

- Begin process
- Input A,B,C
- If $A > B$ then big = A
- Else big = B
- If big $> C$ then bigst = big
Else bigst = C

Example 3

Write an Algorithm to determine a student's final grade and indicate whether it is passing or failing.

The final grade is calculated as the average of four marks.

Solution

- Input a set of 4 marks
- Calculate their average by summing and dividing by 4.
- If average is below 50
Print "Fail"
else
Print "Pass"

WEEK 3

SPECIFIC LEARNING OUTCOMES

To understand:

- The Concept of Algorithm
- Definition of Algorithm
- Features of an Algorithm
- Methods of representing Algorithm

English-like form

The English form of representing an algorithm entails breaking down the solution steps of the problem into single and sequential English words. The steps are represented in English to say what action should be taken in such a step.

Example 1

Develop an algorithm to obtain a book on computer from your school library located on the fourth floor of the building. You are to proceed to the library from your ground floor classroom.

1. Start from the classroom
2. Climb the stairs to the 4th floor and reach the library
3. Search a book on computer
4. Have the book issued
5. Return to your classroom
6. Stop.

Note: The above algorithm solution of example 1, has been written in simple and clear English way.

There is no

Example 2

Develop an algorithm to find the average of four numbers stored in variables A,B,C,D

Solution

1. Start
2. Read values in variables A,B,C,D
3. Calculate the average as $(A+B+C+D)/4$ and store the result in P.
4. Write the value stored in P
5. Stop.

Example 3

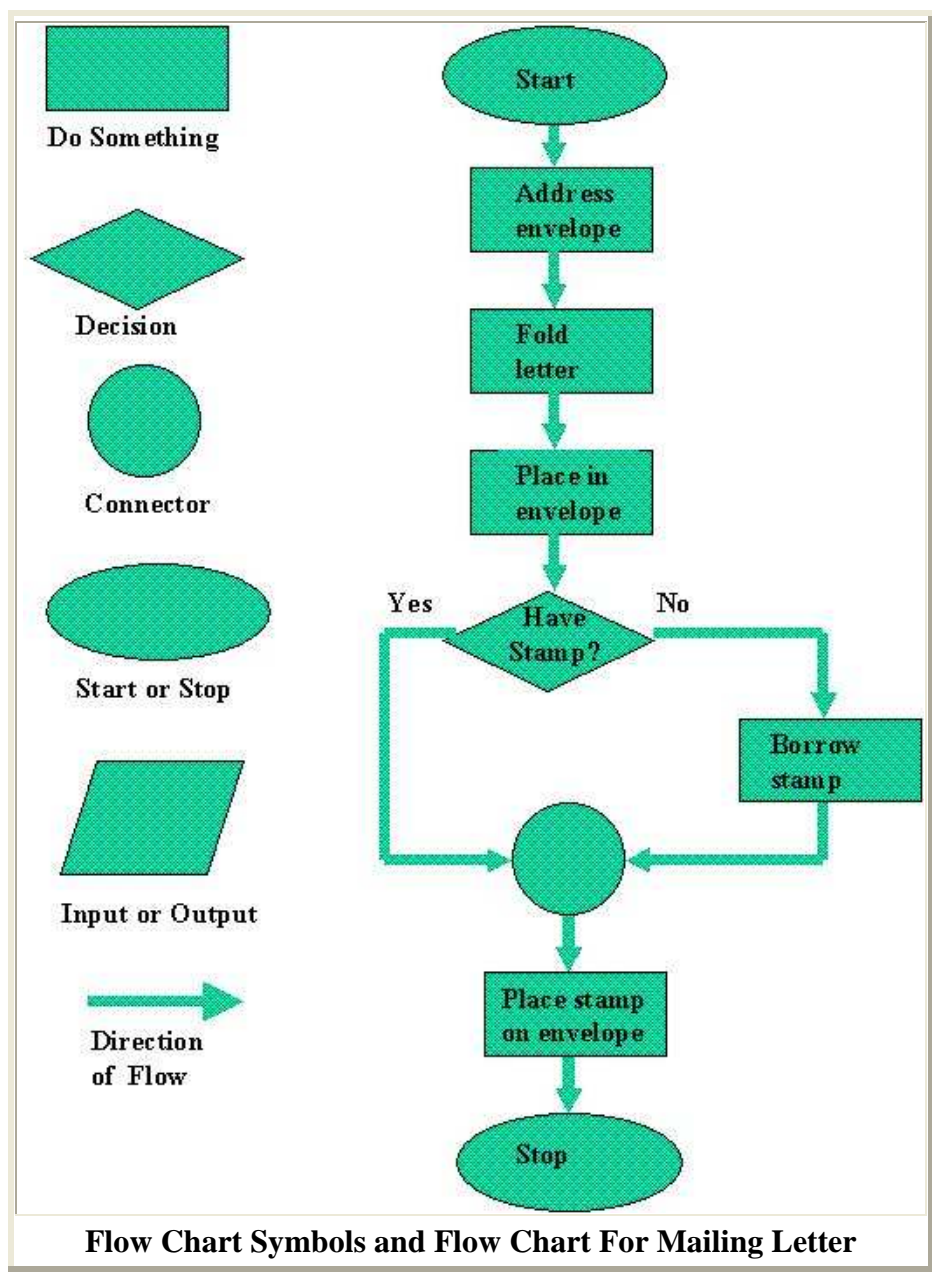
Develop an algorithm to find the average of four numbers stored in variables A, B, C, D. When the value of variable A is zero, no averaging is to be done.

Solution

1. Start
2. Read values stored in variable A,B,C,D
3. If the value of A is Zero, then jump to step 6
4. Calculate the average of A, B, C, D and store the result in variable P.
5. Write the value of P
6. Stop.

Flowchart

Flowchart is a representation of the algorithm using standard symbols. Each symbols has a new function. The Algorithm steps determine which symbol to use to represent it in the flow each step is linked to another step by using the directional arrows.



Example
Write an algorithm
and draw a

flowchart to convert the length in feet to centimeter.

Solution

(Pseudocode)

- Input the length in feet (Lft)
- Calculate the length in an (LCM) by multiplying LFT with 30
- Print Length in Cm (LCM)

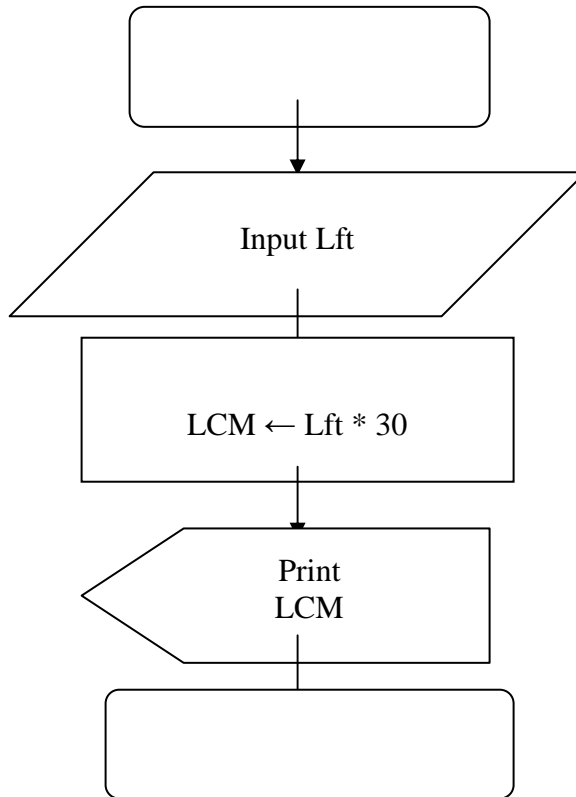
Step 1: Input Lft

Step 2: $Lcm \leftarrow Lft * 30$

Step 3: Print Lcm

Step 4: Stop

Flowchart



WEEK 4

SPECIFIC LEARNING OUTCOMES

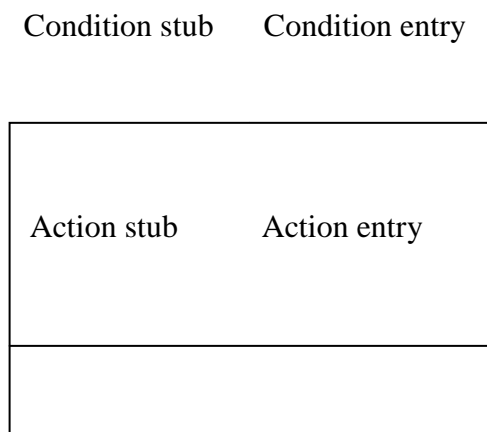
To understand:

- The Concept of Algorithm
- Definition of Algorithm
- Features of an Algorithm
- Methods of representing Algorithm

Decision table

A decision table is a form of truth table that structures the logic of a problem into simple YES and No form. It is easily adapted to the needs of business data processing. It is a rectangle divided into four sections called quadrants. It provides a structure for showing logical relationships between conditions that exist and actions to be taken as a result of these conditions.

The quadrants of a decision table are called the condition stub, the condition entry, the action stub and the action entry respectively.



1. The number of condition is used to determine the number of entries by using the formula, where n is the number of condition.
2. Halving method is used to form the entries in the condition entry e.g if the number of conditions in a question is 3, then the no of entries will be $2^3=8$. Therefore in the condition entry box, the first row will have 4 Ys and 4 Ns. The second row will have 2Ys and 2Ns by 2. Finally the 3rd row will have 1Y and 1N into 4

- 1 Condition stub:- This gives a list of all the conditions that are relevant to the system.
- 2 Condition entry: - Shows a YES or NO entry (abbreviated to Y for YES and N for NO) whether listed condition is present or absent.
- 3 Action stubs: - This quadrant gives a list of all the actions that could be taken by the system, based on the conditions.
- 4 Action entry: - This quadrant indicates whether a specific action will be taken or will not be taken .

The condition entries

- i Y is an indication that the condition is present
- ii N is an indication that the condition is not present
- iii - or blank is an indication that the condition was not tested.

The action entries

- i X is an indication that the listed action to be taken.
- ii blank is an indication that the action is not to be taken.

Advantages of Decision Tables

1. They are simple, practicable and economical .All that is regarded to developed a decision table is a piece of paper and a pencil
2. It makes the system designer to express the logic of the problem in direct and concise terms, which results in the development of an effective and efficient program.
3. It is useful in program documentation i.e decision tables provide a quick and easily understood over view of the system.
4. It is an excellent communication device to breach the gap between the computer personnel who are responsible for developing the system and the non data processing personnel who use the out put of the system.
5. Decision tables are easy to update.
6. It is easy to lean how to use decision table.
7. The complexity and the amount of detail that can be handed by a decision table is un-limited.

Disadvantages

- 1 Total sequence: The total sequence of an operation is not clearly shown in the decision table i.e no overall picture is given as with flowcharts.
- 2 Logic: Where the logic of a system is simple flowcharts always serve the purpose better.

Example

A wholesaler gives discount according to the following rules.

- i Irrespective of the value of the sale and whether it is for cash or credit, existing customers get a 5% discount.
- ii If the sale is for cash, then existing customers receive a 10% discount in total.
- iii If the sale is for over #1000 and for cash then existing customers receive a15% discount in total.

iv New customers never receive a discount of any sort.

You are regarded to construct a limited entry decision table to describe the above process.

SOLUTION

$$n = 3 \quad 2^n = 2^3 = 8$$

<u>Condition stub</u>	<u>Condition entry</u>							
Existing customer?	Y	Y	Y	Y	Y	Y	Y	Y
Cash sales ?	Y	Y	N	N	Y	Y	N	N
Cash sales > #1000 ?	Y	N	Y	N	Y	N	Y	N
<u>Action stub</u>	<u>Action entry</u>							
offer 5% discount					X	X		
.. 10% ..				X				
.. 15% ..		X						
.. no discount					X	X	X	X

<u>Condition stub</u>	<u>Condition entry</u>			
Existing customer	Y	Y	Y	N
Cash sales	Y	Y	N	-
Sales > #1000	Y	N	-	-
<u>Action stub</u>	<u>Action entry</u>			
offer of 5% discount			X	
.. 10% ..		X		
.. 15% ..		X		
no discount			X	

Data flow Diagrams

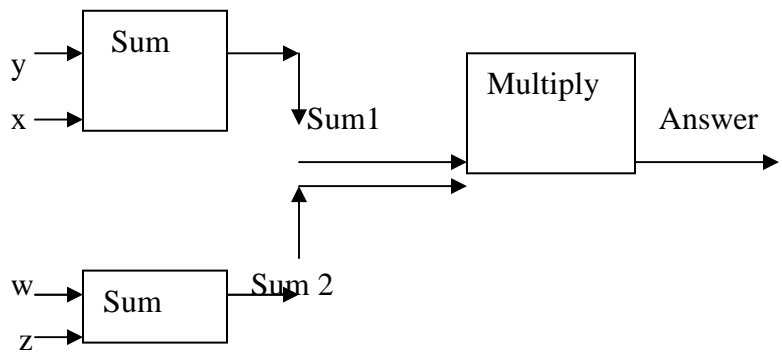
A data flow diagram shows the flow of the data among a set of components. The components may be tasks, software components, or even abstractions of the functionality that will be included in the software system. The actors are not included in the data flow diagrams. The sequence of actions can often be inferred from the sequence of activity boxes.

Rules and Interpretations for correct data flow diagrams.

- 1) Boxes are processes and must be verb phrases
- 2) Arcs represent data and must be labeled with noun phrases.
- 3) Control is not shown. Some sequencing may be inferred from the ordering.
- 4) A process may be a one – time activity, or it may imply a continuous processing.
- 5) Two arcs coming out of a box may indicate that both output are produced or that one or the other is produced.

Example DFD

$$(x + y) * (w + Z)$$



WEEK 5

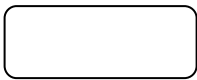


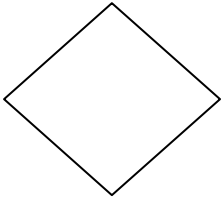


SPECIFIC LEARNING OUTCOMES

To understand:

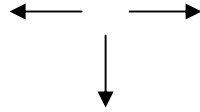
- Definition of flowchart
- Description of flowchart symbols
- Solving simple programming table with flowcharts

The flowchart

A flowchart is a pictorial representation of an Algorithm or of the plan of solution of a problem. It indicates the process of solution, the relevant operations and computations, point of decision and other information that are part of the solution. Flowcharts are of particular importance for documenting a program. Special geometrical symbols are used to construct flowcharts. Each symbol represents an activity. The activity could be input/out of data, computation/processing of data, taking a decision, terminating the solution, etc. The symbols are joined by arrows to obtain a complete flowchart.

<u>Name</u>	<u>Symbol</u>	<u>Use in flowchart</u>
Oval		Denotes the beginning or end of the program.
Parallelogram		Denotes an input operation
Rectangle		Denotes a process to be carried out eg addition, subtraction, division, etc.
Diamond		Denotes a decision (or branch) to be made. The program should continue along one of two routes (eg if/then/else)
Hybrid		Denotes an output operation
Directional		Denotes the directors of logic flow

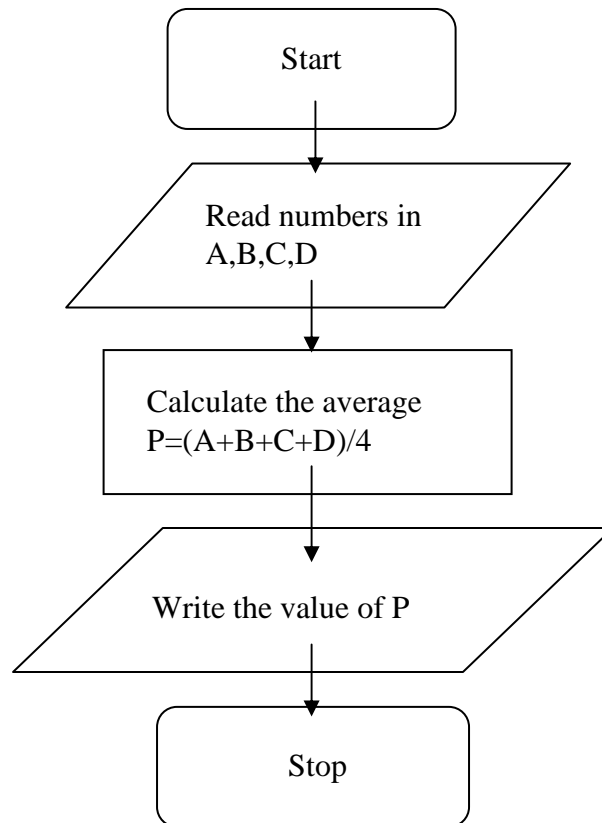
arrows or
flow line



in the program.

Example 1

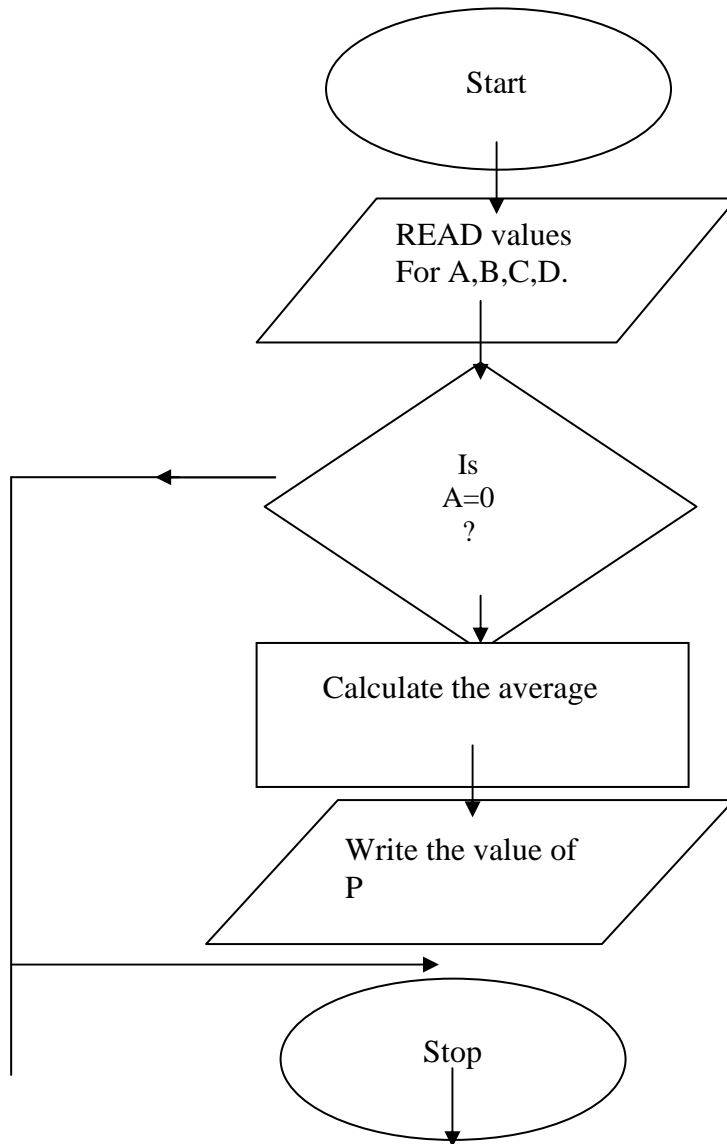
Draw a flowchart to find the average of four numbers stored in variables A,B,C,D

Solution

Example 2

Draw a flowchart to find the average of four numbers stored in variables A,B,C,D. when the value of A is zero, no averaging is to be done.

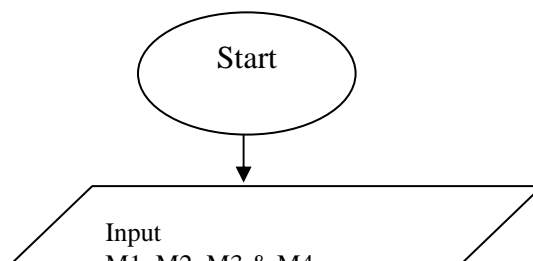
Solution



Example 3

Write a flowchart to determine a student final grade and indicate whether it is pass or fail. The final grade is calculated as the average of four marks.

Solution



Example 4

Write an algorithm and draw a flowchart that will read the two sides of a rectangle and calculate its area.

Solution

Pseudo code

- Input the Width(w) and Length(L) of a rectangle

- Calculate the area (A) by multiplying L with W
- Print A.

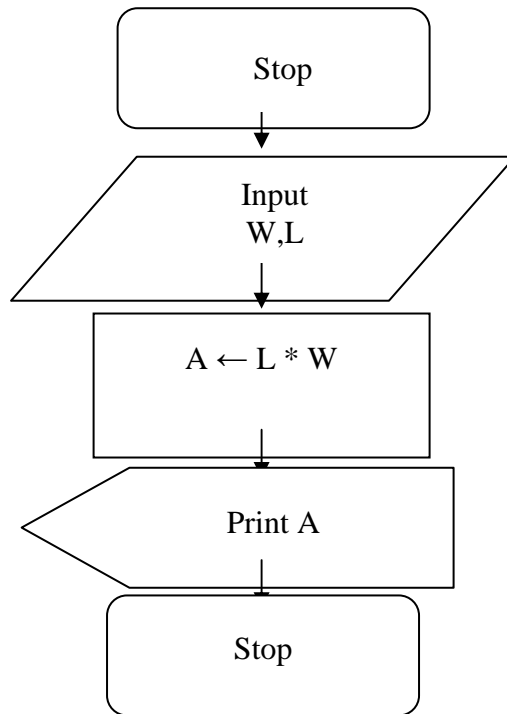
Algorithm

Step 1: Input W, L

Step 2: $A \leftarrow L * W$

Step 3: Print A.

Flowchart



Example 5

Write an algorithm and draw a flowchart that will calculate the roots of a quadratic equation.

$$ax^2 + bx + c = 0$$

Hint: $d = \text{SQrt}(b^2 - 4ac)$, and the roots are:

$$X1 = (-b + d)/2a \text{ and } X2 = (b - d)/ 2a$$

Solution

Pseudo code

- Input the coefficients (a, b, c) of the quadratic equation
- Calculate d
- Calculate X1
- Calculate X2
- Print X1 and X2

Algorithm

Step 1: Input a, b, c

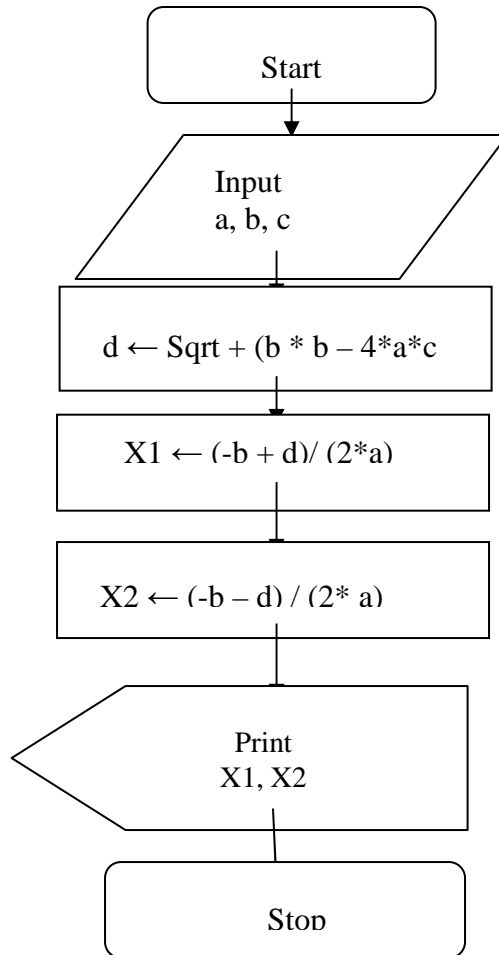
Step 2: $d \leftarrow \text{Sqrt}(b * b - 4 * a * c)$

Step 3: $X1 \leftarrow (-b + d) / (2 * a)$

Step 4: $X2 \leftarrow (-b - d) / (2 * a)$

Step 5: Print X1, X2

Flowchart



Uses of flowcharts

- 1 It gives us an opportunity to see the entire system as a whole.
- 2 It makes us to examine all possible logical outcomes in any process.
- 3 It provides a tool for communicating i.e a flowchart helps to explain the system to others.
- 4 To provide insight into alternative solutions.
- 5 It allows us to see what will happen if we change the values of the variable in the system.

ADVANTAGES OF USING FLOWCHART

1. Communication flowcharts are visual aids for communicating the logic of a system to all concerned.
2. Documentation: flowcharts are a means of documentation because:
 - 3 The analyst/ programmers may leave the arrangement or they may forget the logic of the program.
 - 4 Changes to the procedure are more easily catered for (modification).
 - 5 Flowchart can be understood by new staff coming to the company
- 6 Analysis: flowcharts help to clarify the logic of a system i.e the overall picture of the organization can be seen.
- 7 Consistency: A flowchart is a consistent system of recording. It brightens @ the relationships between different parts of a system.

DISADVANTAGES OF USING FLOWCHART

1. Complex logic :- Where the logic of a problem is complex, the flowchart quickly becomes clustered and lacks clarity.
2. Alterations:- If alterations are required the flowchart may require redrawing completely.
3. Reproduction :- As the flowchart symbols cannot be typed, reproduction of flowchart is often a problem.

WEEK 6

SPECIFIC LEARNING OUTCOMES

To understand:

- * Design algorithm for problems involving.
 - Strictly sequence control structure
 - Selection control structure
 - Iteration control structure

Designing Algorithm For Common Programming Logic Structures

Basic Coding Structures

All computer programs can be coded using only three logic structures (or programs) or combinations of these structures:

1. Simple sequence
2. Selection
3. Repetition

The three structures are useful in a disciplined approach to programming because

1. The program is simplified. Only the three building blocks are used, and there is a single point of entry into the structure and a single point of exit.

2. The three coding structures allow a program to be read from top to bottom making the logic of the program more visible for checking and for maintenance.

Simple Sequence

The simple-Sequence structure consists of one action followed by another. In other words, the flow of control is first to perform operation A and then to perform operation B.. A simple sequence is flowcharted as two process symbols connected by a flowline.

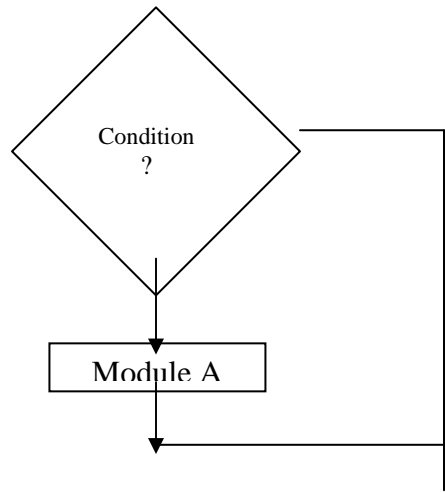
Selection

The selection structure consists of a test for condition followed by two alternative paths for the program to follow. The program selects one of the program control paths depending on the test of the condition. After performing one of two paths, the program control returns to a single point. This pattern can be termed IF .. ELSE because the logic can be stated (for condition P and operations C and D): IF P (is true), perform C; ELSE perform D . A flowchart for the selection structure consists of a decision symbol followed by two paths, each with a process symbol, coming together following the operation symbols.

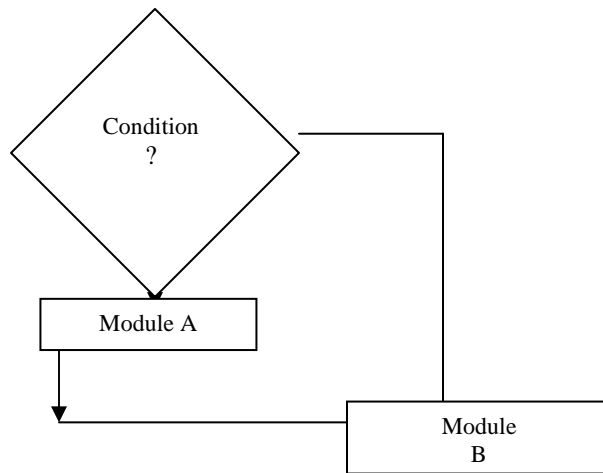
- (a) Single alternative – This structure has the form:

[IF condition, Then:]

[End of IF structure]



(b) Double alternative: This structure has the form



IF Condition, then:
[Module A]

Else
[Module A]

[End of IF structure]

ie IF condition holds, then module A executed; otherwise module B is executed.

(c) Multiple alternatives: This structure has the form:

IF Condition (1), then
[Module A,]

Else if condition (2), then:
[module Az]

:

Else if condition (m), then;
 [module Am]
Else [module B] (END OF IF structure)

Iteration Logic (**Repetitive flow**)

The repetition structure can also be called a loop. In a loop, an operation or a set of operation is repeated until some condition is satisfied. The basic form of repetition is termed DO WHILE in the literature of structured programming. In some languages, the repetition structure might be termed PERFORM UNTIL. In the perform until pattern, the program logic tests a condition; if it is true, the program executes the operation and loops back for another test. If the condition is true, the repetition ceases.

This structure has the form:

Repeat for K = R to S by T:
 [Module]
[End of loop]

Algorithm example for iteration or repetition

For example let us take 10 sets of numbers each set containing three. The problem is to get the biggest number in each set and print it.

Algorithm

- Step 1: Read the total number of sets
- Step 2: Initialize the number of the set as N=1
- Step 3: Read three numbers of a set say A, B, C.
- Step 4: Compare A with B and choose the bigger.
- Step 5: Compare the bigger number with C and Choose the biggest
- Step 6: Print the biggest number,
- Step 7: Increment the number of the set by 1
(N=N+1)
- Step 8: Check whether we have exceeded 10. If not Go – To step 3. Otherwise.
- Step 9: STOP

The flow chart for the same is given below

WEEK 7

SPECIFIC LEARNING OUTCOMES

To understand:

- Explain modular programming concept.
- Explain top-down design technique.
- Illustrate program design with program structure charts, hierarchical Network, Hierarchical.

The Concept of Modular Programming

As program become larger, and more complex, it becomes more difficult to write clear understandable solutions that work correctly. The goal of modular programming is to break up the program into small parts that are more easily understood. The planning, coding and testing can be done on these small, relatively simple units, rather than on one large, complex body of code.

Programmers must develop the skill and the ability to look at a large program and to decompose it into individual factions. Once a programmer has learned to modularize programs, program will be coded more quickly, will be more likely to work correctly, and will certainly be easier to read to be maintained by others.

Virtually all computer scientists recommend modular programming. The only disagreement seems to be at what point a programmer should begin writing what are called “subroutines”. Many programmers wait until programs become hopelessly complex. Then introducing subroutine can save the day. The more practical approach is to being using subroutines early. As programs become more complex, if correct habits need already in place, the programmer doesn’t need to be “rescued” The solution to the program is at hand. A subroutine is a group of statements intended to accomplish an individual task.

When a program is written with individual tasks in subroutine, a mainline, or control program is needed. This control program is sometimes called the program outline, as it presents an overviewed of the program tasks. [Another term sometimes used for the program maintain is the driver.

Modular program planning

There are several popular methods used for planning modular programs. Pseudo code or flowcharts maybe used, with slight modification for the subroutine, or hierarchy charts maybe used. The three methods can be illustrated example programming program as shown below.

Program example

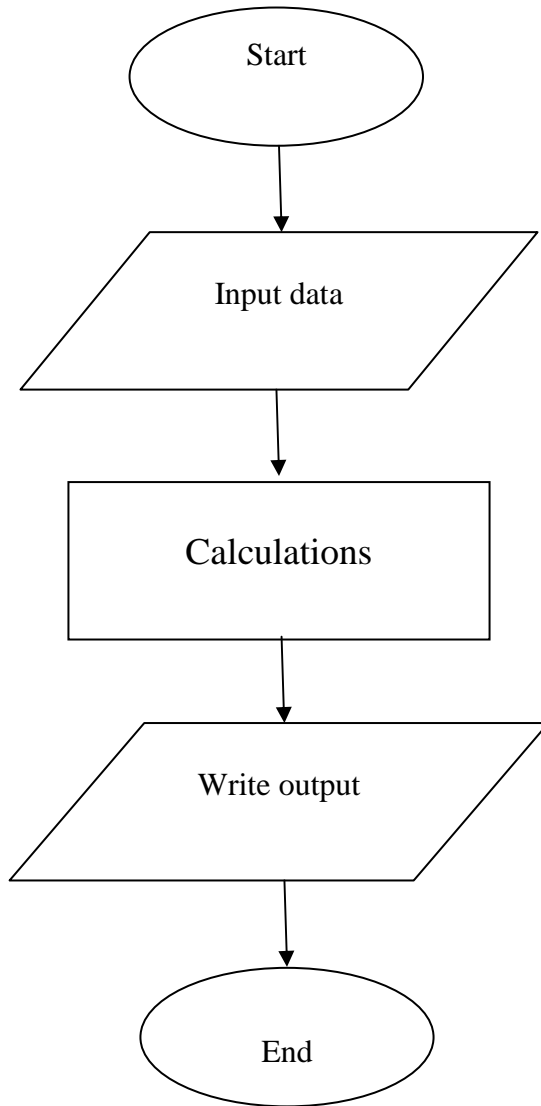
Using modular (subroutines) approach, show the program plan of a computer program for calculating simple interest on a deposit by a customer.

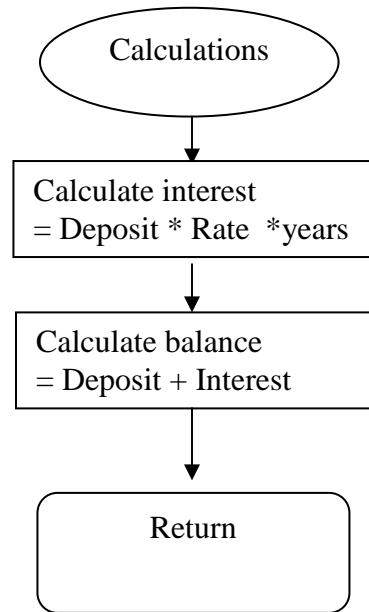
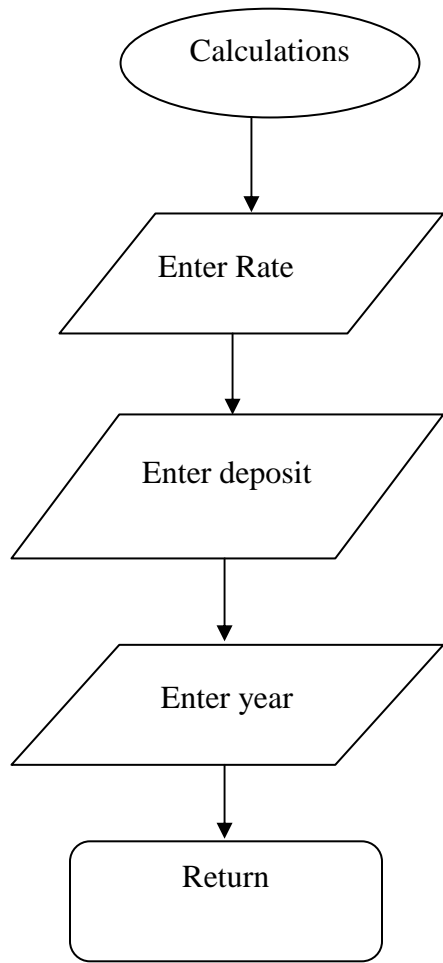
Solution**Modular Pseudo code plan for the problem**

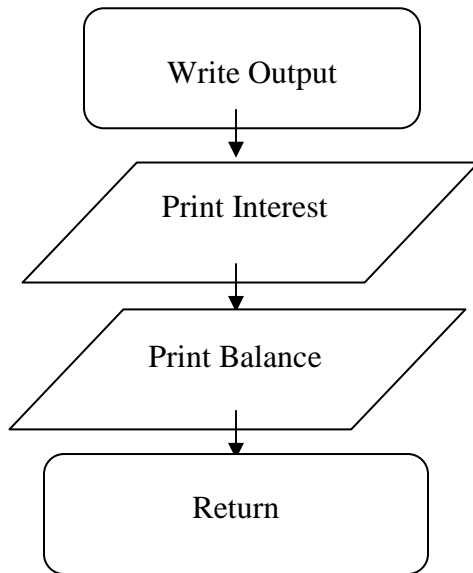
1. Input data
 - 1.1 Prompt and input rate, deposit amount, and number of years.
2. Calculations
 - 2.1 Calculate interest = deposit * rate * years
 - 2.2 Calculate ending balance = deposit + interest
3. Output
 - 3.1 Print interest and ending balance

Modular flowchart plan for the problem

Program mainline Subroutines

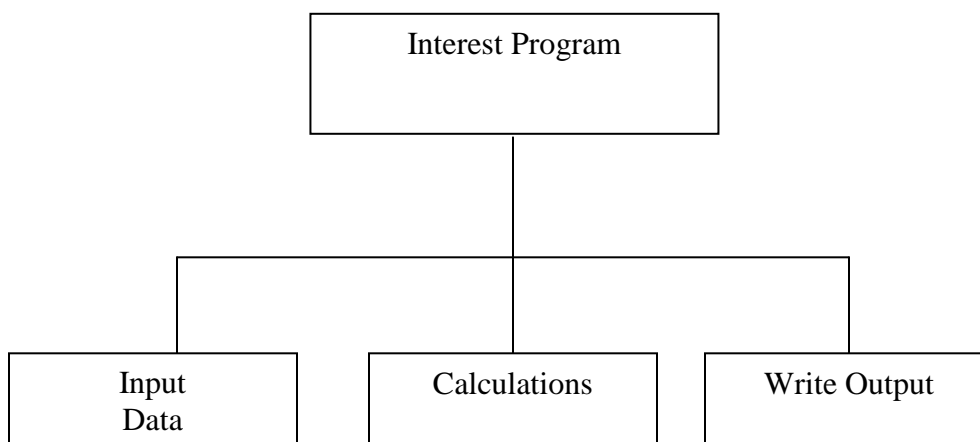






Modular Hierarchy plan for the problem

Many programmers who write modular programs prefer to plan their programs with hierarchy chart. A hierarchy chart is used to plan and show program structure. It is constructed much like an organization chart. As shown the solution below, level A shows the entire program, which is broken down into major program functions on the B level. The modules can be broken into smaller and smaller parts until the coding for each function becomes straightforward. Many programmers use a hierarchy chart to plan the overall structure of a program. Then, when the individual modules are identified, flowcharting or pseudo code will be used to plan the details of the logic.



WEEK 8**SPECIFIC LEARNING OUTCOMES****To understand:**

- Identify the problem and confirm it solvable.
- Design algorithm for the chosen method of solution with flowcharts or pseudo codes.
- Code the algorithm by using a suitable programming language.
- Test run the program on the computer.

STAGES OF PROGRAM DEVELOPMENT

Before computer program is successfully written, documented and installed, it must have passed through the following stages. Each stage has something to contribute to the accomplishment of the whole task. The stages are:

Problem definition:

Before any reasonable and meaningful program could be written, the problem that prompted it must have to be defined. No one solves a problem he does not know. The problem to be solved by computer should be well stated and understood before the solution will be worked out. From the solution, it is expected that the output of the problem is known and the input will be prepared to arrive at the output.

Develop the algorithm

An algorithm is a well defined set of instructions that is used to solve a particular problem in a finite number of steps. It involves unambiguous stating of the procedures and steps necessary to transform the input data into output. It poses a little difficulty to the program planner, and once accomplished successfully, the rest of the solution follows easily.

Plan the logic of the program/flowcharting:

The logic of the program will be planned using any of the program design tools it flowchart, pseudocode or hierarchy chart. The choice of the design tool used depends on the programmer, but the most popular and most handy is the use of the flowchart to organize the thought of the program planner and to check for any logic error or misrepresentation. A flowchart is a pictorial view of the program logic

Write the computer program:

After the design or planning the logic of the program using the flowchart, the next stage is the actual writing of the program using any of the programming languages in a proper sequence. This is called, coding of the program. This is done by strictly obeying the language syntax or following the established rules of the programming language.

Type the program into computer:

The next stage after writing the program, it to key the program into the computer. Any program that will be executed by the computer must be resident in the computer memory. The typing is generally made one line after the other.

Test and debug the program:

The moment the program has been keyed into the computer, the programmer is ready to see if the program is working. The program could be translated into machine language by either a compiler or interpreter depending on the language in use ie for BASIC program, when the command RUN is typed and entered, the program begins executing. If any rules language is broken, the program will not work. The errors must be removed before the program will start producing the output. Testing is very necessary to ensure that the correct and required answers are produced as the output.

Document the work:

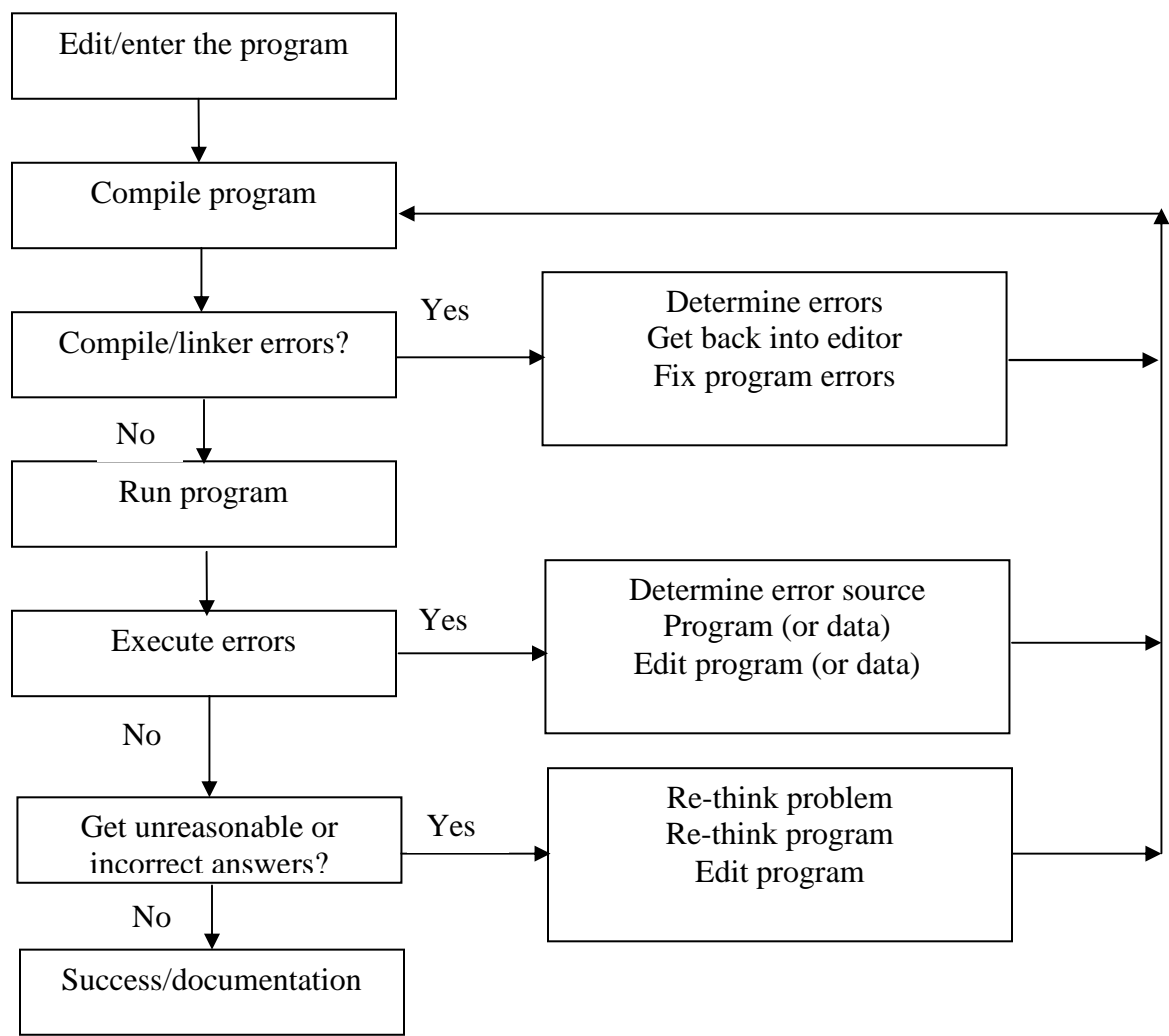
Documentation helps the user to understand the program better. It identifies exactly the purpose of the program. It is always referred to whenever changes are to be made in the program to suite new development. It contains the following parts.

- i) A statement of the problem
- ii) Algorithm and program plans (ie flowchart, hierarchy chart or pseudocode).
- iii) Description of input and output
- iv) Program listing
- v) Test data and results
- vi) Technical details and instruction for the user.

All these are assembled into a finished program documentation.

Program development/execution process

The program development stages/process is illustrated in the diagram below;



WEEK 9

SPECIFIC LEARNING OUTCOMES

To understand:

- Explain machine language, low-level language and High level languages
- Various programming languages
- Differentiate between programming languages

LEVELS OF COMPUTER PROGRAMMING LANGUAGES

All computers whether small or big cannot do anything on their own. They all require a series of instructions (i.e programs) before they can do any processing. It is these programs that will direct the computer to carry out the required task. The programs have to be written out comprehensively: to cover all possibilities: and in the right order before the control unit of the CPU can use them effectively.

Programs can be written in several languages. Just as there are many spoken languages, there are many computer languages. In this lecture we shall study the different levels of computer languages and their forms.

Currently all computer languages can be grouped into three, namely, machine languages, assembly languages and high-level languages. Machine languages and assembly languages are together referred to as low-level languages. The detail characteristics of each group of languages are discussed below.

Low-level Languages

These group of languages are so named because in form they are very close (i.e. similar) to the language the computer understands, and very remote from languages spoken by human beings. Low-level languages are in two forms namely: machine language and assembly language.

Machine language

Machine language is as old as the computer itself. It is the computer's own language. It consists of the code that designates the electrical states in the computer (i.e, on or off): this is expressed as combination of Os and 1s./It is called the computer's own language because codes or instructions written in machine language can be executed directly by the computer; without the need for any translation. This is the only language that has this characteristic.

Each type of computer has its own machine language. That is to say, that different brands of computers cannot understand programs written in another brand's machine language. Talking specifically, a computer made by IBM company has its own language which is different from the one of NCR

company. Even two different models of computers made by the same company do not usually have the same machine language. Thus programs written in machine language are said to be machine-dependent.

Every instruction in machine language programs must specify both the operation to be carried out as well as the storage locations of the data items to be used in the operation. In form, it consists of a series of numbers. The operation part is called opcode or operation code and the remaining part gives the addresses of the data items in memory that will be affected by the operation. Due to these special requirements machine language programming is extremely complex, tedious and time consuming.

For example, the instruction, in machine language, to make the computer add together the numbers currently stored in memory address four and seven and then store the sum in address four will look like this.

1A47

The first two numbers 1A is the operation code for add in IBM 360 machine. On another machine it will be another series of numbers different from the one given.

For effective and efficient program in machine language, the programmer must keep track of which memory locations have been used and the purpose of each memory location. Also the programmer must know every operation code and the action that it causes the computer to take. It is quite lengthy and tedious. To overcome this, the assembly language come into existence.

Assembly Language

In order to relieve programmers the arduous task of writing in machine language, the assembly language was developed. It is very much similar to machine language but instead of writing in series of numbers, convenient symbols and abbreviations are used. Assembly language programming does not require the programmer to remember numeric opcodes and addresses. However, it still requires the programmer to be familiar with the operation codes and the methods of addressing memory locations

for that particular machine. This is because, the assembly language; though at a higher level, still depend very much on the language of that particular machines.

Programs in assembly language cannot be executed directly, it still has to be changed to the machine language during execution. Thus we can see that assembly language too is machine dependent. That is to say, different brands of computers have different assembly language. For this reason, assembly language is still classified as low-level language.

For the IBM 360 computer the machine language code for add is 1A while in assembly language, addition operation code is AR is a mnemonic for "Add Register". For STORE operation the assembly code is the mnemonic STO while TRA stands for TRANSFER Operation and MR stands for MOVE REGISTER operation etc.

Similarly the programmer can assign a name to each memory location. For instance address seven may be given the name P and address four may be given the name Q, thus the instruction.

AR P, Q

In assembly language will be executed as adding the contents of register Q to the contents of P; of course the final result will be in register P.

As mentioned earlier, regardless of which assembly language is used, the computer cannot directly execute the programs written in this language. It has to be translated into the machine language by another special purpose software called translator. The details of the translation process shall be studied in later lectures.

High-level languages

As computers have developed in complexity, so have programming languages. High-level language programming are the result of sophistication in programming languages.

The Machine and Assembly languages discussed before require programmers to construct programs in a form that does not follow normal ways of human thinking, communication and language notation. To avoid this problem High-level languages were developed. Another name for high-level languages is

problem-oriented languages. With this language, programmers' attention are now directed towards problem solving instead of operations going on inside computer. These languages allow mathematicians and Scientists to use common algebraic notations for coding formulas while other lay programmers can write their programs in ordinary sentence form. The time and effort needed to write programs are now reduced considerably and programs are easier to correct and modify.

A large number of high-level languages are in use today. In fact, more are being developed daily as researchers are still going on.

Example of high-level languages commonly in use today include:

- 1) BASIC (Beginners All-purpose Symbolic Instruction Code)
- 2) FORTRAN (Formula Translator). There are many versions of FORTRAN However the modern version is called FORTRAN 77
- 3) COBOL (Common Business Oriented Language). There are many versions
- 4) `C' language
- 5) PL/1
- 6) PASCAL (language named after an ancient French Mathematician and inventor of Pascal engine), etc.

Each of the high-level languages has rules that govern how to write instructions in them. Like any human language, it is the duty of the programmer or user to learn the rules of the language he wants to use.

Unlike low-level language which is machine-dependent, high-level languages are machine-independent. That is to say, a program written in any of the high-level languages can be run

with little or no changes by computer made by many different manufacturers. Thus, as new computers come into existence, programmers do not have to rewrite the existing programs and learn new language as it is the case with assembly programming.

The example below shows how to add two numbers held in variables X and Y placing the sum in X using the most common four high-level languages.

BASIC..... LET X = X + Y

FORTRAN X = X + Y

COBOL..... ADD Y TO X

PASCAL..... X := X + Y

It can be observed that the notation is very similar to human ways of thinking and expression and very remote from the machine language.

WEEK 10

SPECIFIC LEARNING OUTCOMES

To understand:

- Explain the distinguishing features of different programming languages
- Distinguish between system commands and program statements.
- Advantages and disadvantages of different levels of programming languages

MACHINE LANGUAGE

Machine Code or machine language is a low-level programming language that can be understood directly by a computer's central processing unit (CPU). Machine code consists of sequences of binary numbers, or bits, which are usually represented by 1s and 0s, and which form the basic instructions that guide the operation of a computer. The specific set of instructions that constitutes a machine code depends on the make and model of the computer's CPU. For instance, the machine code for the Motorola 68000 microprocessor differs from that used in the Intel Pentium microprocessor.

Writing programs in machine code is tedious and time-consuming since the programmer must keep track of each specific bit in an instruction. Another difficulty with programming directly in machine code is that errors are very hard to detect because the program is represented by rows and columns of 1s and 0s.

Advantages of Machine Language

- 1) Less code is produced
- 2) Storage is saved
- 3) User has direct control of machine instruction
- 4) Execution is faster as no translation is needed
- 5) The programmer knows all the registers and instruction that use them.

Disadvantages of Machine Language

- 1) Cumbersome ie, tedious and difficult to learn
- 2) Programmer's fluency is affected, thereby making the programs developed inefficient.
- 3) The developed programs are error prone and difficult to debug (correct)
- 4) The performance of the system is unreliable.

ASSEMBLY LANGUAGE

Assembly language is type of low-level computer programming language in which each statement corresponds directly to a single machine instruction. Assembly languages are thus specific to a given processor. After writing an assembly language program, the programmer must use the assembler specific to the microprocessor to translate the assembly language into machine code. Assembly language provides

precise control of the computer, but assembly language programs written for one type of computer must be rewritten to operate on another type. Assembly language might be used instead of a high-level language for any of three major reasons: speed, control, and preference. Programs written in assembly language usually run faster than those generated by a compiler; use of assembly language lets a programmer interact directly with the hardware (processor, memory, display, and input/output ports).

Assembly language uses easy-to-remember commands that are more understandable to programmers than machine-language commands. Each machine language instruction has an equivalent command in assembly language. Assembly language makes programming much easier, but an assembly language program must be translated into machine code before it can be understood and run by the computer. Special utility programs called assemblers perform the function of translating assembly language code into machine code. Like machine code, the specific set of instructions that make up an assembly language depend on the make and model of the computer's CPU. Other programming languages such as Fortran, BASIC, and C++, make programming even easier than with assembly language and are used to write the majority of programs. These languages, called high-level languages, are closer in form to natural languages and allow very complicated operations to be written in compact notation.

Advantages of Low Level Language

- 1) Program translation is easier than high level language
- 2) It affords the programmer the opportunity to understand the internal structure of the hardware and its registers.

Disadvantages of Low Level Language

- 1) It is machine dependent, That is, cannot be transferred to another computer.
- 2) Program development is slow as the programmer must have detailed knowledge of the hardware structure.
- 3) Program maintenance is slow and error prone.

HIGH LEVEL LANGUAGE

High-Level Language is a computer language that provides a certain level of abstraction from the underlying machine language through the use of declarations, control statements, and other syntactical structures. In practice, the term comprises every computer language above assembly language. The next generation of language is called the 3rd generation. The computer programmers enjoy using this language because it gives them the fluency, the flexibility and the opportunity to express their thought to the best of their ability. The languages of this generation are called High level language. The high level languages are referred to as machine language and assembly language.

Advantages of High Level Language

- 2) It makes programming easier for the human being.
- 3) High level instructions are easier to understand and faster to code.
- 4) Error correction and testing of program is easier
- 5) They are machine independent. That is, program written for computer can be transferred to another computer with little or no modification.

Disadvantages of High Level Languages

- 1) High level language tends to be inefficient in the use of CPU and other facilities.
- 2) Machine code instructions are produced and then requires more storage spaces.
- 3) More time is required to run the program as it has to be translated.

WEEK 11

SPECIFIC LEARNING OUTCOMES

To understand:

- * Debugging.
- Identify sources of bugs in a program
- Explain syntax, run-time and logical errors.
- Identify techniques of locating bugs in a program
- Explain program maintenance.
- Distinguish between debugging and maintaining a program

THE CONCEPT OF DEBUGGING AND MAINTAINING PROGRAM

Debugging is the art of diagnosing errors in programs and determining how to correct them. "Bugs" come in a variety of forms, including: coding errors, design errors, complex interactions, poor user interface designs, and system failures. Learning how to debug a program effectively, then, requires that you learn how to identify which sort of problem you're looking at, and apply the appropriate techniques to eliminate the problem.

Bugs are found throughout the software lifecycle. The programmer may find an issue, a software tester might identify a problem, or an end user might report an unexpected result. Part of debugging effectively involves using the appropriate techniques to get necessary information from the different sources of problem reports.

Debugging is described as identification and removal of localized implementation errors or bugs from a program or system. Program debugging is often supported by a *debug tool*, a software tool that allows the internal behavior of the program to be investigated in order to establish the existence of bugs. This tool typically offer trace facilities and allow the planting of *breakpoint* in the program at which execution is to be suspended so that examination of partial results is possible and permit examination and modification of the values of program variables when a breakpoint is reached.

In computer program/software, a bug is an error in coding or logic that causes a program to malfunction or to produce incorrect results. The computer software (debug tool) is used to detect, locate, and correct logical or syntactical errors in a computer program. Similarly, in hardware, a bug is a recurring physical problem that prevents a system or set of components from working together properly. To detect, locate, and correct a malfunction or to fix an inoperable system, the term *troubleshoot* is more commonly used in hardware contexts. The three major program error are; syntax error, logical error and run-time error.

Sources of bugs in a program

With coding errors, the source of the problem lies with the person who implements the code. Examples of coding errors include:

- Calling the wrong function ("moveUp", instead of "moveDown")
- Using the wrong variable names in the wrong places ("moveTo(y, x)" instead of "moveTo(x, y)")
- Failing to initialize a variable ("y = x + 1", where x has not been set)
- Skipping a check for an error return

Software users readily see some design errors, while in other cases design flaws make a program more difficult to improve or fix, and those flaws are not obvious to a user. Obvious design flaws are often demonstrated by programs that run up against the limits of a computer, such as available memory, available

disk space, available processor speed, and overwhelming input/output devices. More difficult design errors fall into several categories:

- Failure to hide complexity
- Incomplete or ambiguous "contracts"
- Undocumented side effects

Complex interactivity bugs arise in scenarios where multiple parts of a single program, multiple programs, or multiple computers interact.

Sometimes, computer hardware simply fails, and it usually does so in wildly unexpected ways. Determining that the problem lies not with the software itself, but with the computer(s) on which it is usually complicated by the fact that the person debugging the software may not have access to the hardware that shows the problem.

Preventing Bugs

No discussion of debugging software would be complete without a discussion of how to prevent bugs in the first place. No matter how well you write code, if you write the wrong code, it won't help anyone. If you create the right code, but users cannot work the user interface, you might as well have not written the code. In short, a good debugger should keep an open mind about where the problem might lie.

Although it is outside the scope of this discussion to describe the myriad techniques for avoiding bugs, many of the techniques here are equally useful after the fact, when you have a bug and need to uncover it and fix it. Thus, a brief discussion follows.

Methods of debugging

Understand the Problem

In order to write effective software, the developer must solve the problem the user needs solved. Users, naturally enough, do not think in strict algorithms, windowing systems, web pages, or command line interfaces. Rather, users think of their problems in the way that they think of their problems (yes, that is circular).

Sit down with the intended user, and ask them what they want from the software. Users frequently want more than software can actually deliver, or have contradictory aims, such as software that does more, but doesn't require that they learn anything new. In short, ask the users what their goals are. Absent those goals, users will keep reporting bugs that do not add up to a coherent whole.

Basic debugging techniques/steps

Although each debugging experience is unique, certain general principles can be applied in debugging. This section particularly addresses debugging software, although many of these principles can also be applied to debugging hardware.

The basic steps in debugging are:

- Recognize that a bug exists
- Isolate the source of the bug
- Identify the cause of the bug
- Determine a fix for the bug
- Apply the fix and test it

Recognize a bug exists

Detection of bugs can be done proactively or passively.

An experienced programmer often knows where errors are more likely to occur, based on the complexity of sections of the program as well as possible data corruption. For example, any data obtained from a user should be treated suspiciously. Great care should be taken to verify that the format and content of the data are correct. Data obtained from transmissions should be checked to make sure the entire message (data) was received. Complex data that must be parsed and/or processed may contain unexpected combinations of values that were not anticipated, and not handled correctly. By inserting checks for likely error symptoms, the program can detect when data has been corrupted or not handled correctly.

If an error is severe enough to cause the program to terminate abnormally, the existence of a bug becomes obvious. If the program detects a less serious problem, the bug can be recognized, provided error and/or log messages are monitored. However, if the error is minor and only causes the wrong results, it becomes much more difficult to detect that a bug exists; this is especially true if it is difficult or impossible to verify the results of the program.

The goal of this step is to identify the symptoms of the bug. Observing the symptoms of the problem, under what conditions the problem is detected, and what work-around, if any, have been found, will greatly help the remaining steps to debugging the problem.

Isolate source of bug

This step is often the most difficult (and therefore rewarding) step in debugging. The idea is to identify what portion of the system is causing the error. Unfortunately, the source of the problem isn't always the same as the source of the symptoms. For example, if an input record is corrupted, an error may not occur

until the program is processing a different record, or performing some action based on the erroneous information, which could happen long after the record was read.

This step often involves iterative testing. The programmer might first verify that the input is correct, next if it was read correctly, processed correctly, etc. For modular systems, this step can be a little easier by checking the validity of data passed across interfaces between different modules. If the input was correct, but the output was not, then the source of the error is within the module. By iteratively testing inputs and outputs, the debugger can identify within a few lines of code where the error is occurring.

Identify cause of bug

Having found the location of the bug, the next step is to determine the actual cause of the bug, which might involve other sections of the program. For example, if it has been determined that the program faults because a field is wrong, the next step is to identify why the field is wrong. This is the actual source of the bug, although some would argue that the inability of a program to handle bad data can be considered a bug as well.

A good understanding of the system is vital to successfully identifying the source of the bug. A trained debugger can isolate where a problem originates, but only someone familiar with the system can accurately identify the actual cause behind the error. In some cases it might be external to the system: the input data was incorrect. In other cases it might be due to a logic error, where correct data was handled incorrectly. Other possibilities include unexpected values, where the initial assumptions were that a given field can have only "n" values, when in fact, it can have more, as well as unexpected combinations of values in different fields (field x was only supposed to have that value when field y was something different). Another possibility is incorrect reference data, such as a lookup table containing incorrect values relative to the record that was corrupted.

Having determined the cause of the bug, it is a good idea to examine similar sections of the code to see if the same mistake is repeated elsewhere. If the error was clearly a typo, this is less likely, but if the original programmer misunderstood the initial design and/or requirements, the same or similar mistakes could have been made elsewhere.

Determine fix for bug

Having identified the source of the problem, the next task is to determine how the problem can be fixed. An intimate knowledge of the existing system is essential for all but the simplest of problems. This is because the fix will modify the existing behavior of the system, which may produce unexpected results. Furthermore, fixing an existing bug can often either create additional bugs, or expose other bugs that were already present in the program, but never exposed because of the original bug. These problems are often caused by the program executing a previously untested branch of code, or under previously untested conditions.

In some cases, a fix is simple and obvious. This is especially true for logic errors where the original design was implemented incorrectly. On the other hand, if the problem uncovers a major design flaw that permeates a large portion of the system, then the fix might range from difficult to impossible, requiring a total rewrite of the application.

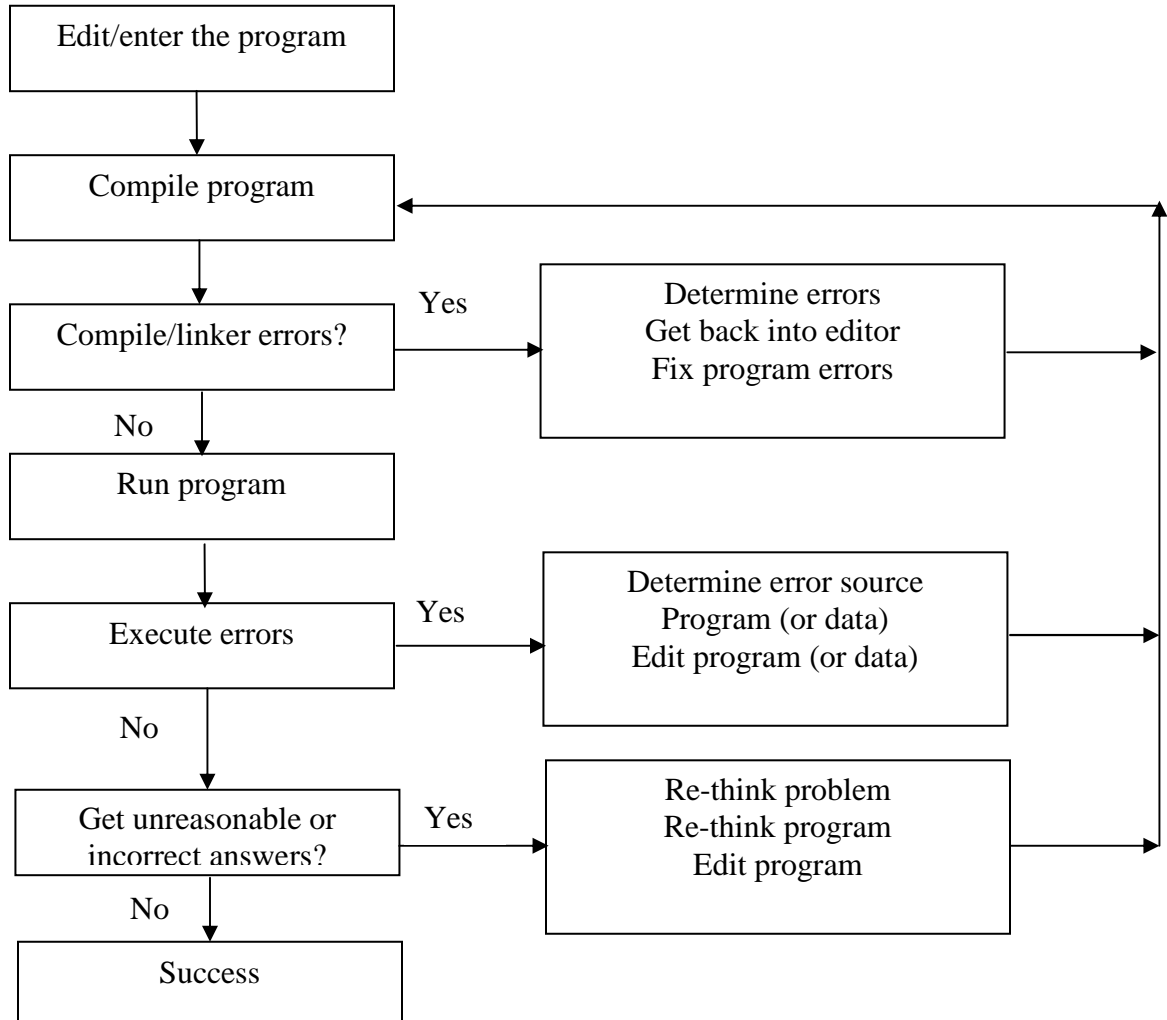
In some cases, it might be desirable to implement a "quick fix", followed by a more permanent fix. This decision is often made by considering the severity, visibility, frequency, and side effects of the problem, as well as the nature of the fix, and product schedules (e.g., are there more pressing problems?).

Fix and test

After the fix has been applied, it is important to test the system and determine that the fix handles the former problem correctly. Testing should be done for two purposes: (1) does the fix now handle the original problem correctly, and (2) make sure the fix hasn't created any undesirable side effects.

For large systems, it is a good idea to have regression tests, a series of test runs that exercise the system. After significant changes and/or bug fixes, these tests can be repeated at any time to verify that the system still executes as expected. As new features are added, additional tests can be included in the test suite.

The diagram below illustrates the fix and test approach of debugging a program.



Syntax of a program

The syntax of a program is the rules defining the legal sequences of symbolic elements in a particular language. The syntax rules define the form of various constructs in the language, but say nothing about the meaning of these constructs. Examples of constructs are; expressions, procedures and programs.

Programming Errors

Error simply means mistake. That is errors occur in programs as a result of system failure (hardware), wrong code/instructions (software) and human error. There are four categories of programming error;

Run-time errors (execution error)

Is an error that occurs during the execution of a [program](#). In contrast, *compile-time* errors occur while a program is being [compiled](#). Runtime errors indicate [bugs](#) in the program or problems that the designers had anticipated but could do nothing about. For example, running out of [memory](#) will often cause a runtime error.

Note that runtime errors differ from [bombs](#) or [crashes](#) in that you can often recover gracefully from a runtime error.

Run-time errors have the following basic characteristics;

- Program is compiled OK, but something goes wrong during execution e.g division by zero or an attempt to read data that does not exist.
- Detected by the computer run-time system
- Computer usually prints error message and stops.

Define logical errors

A problem that causes a program to produce invalid output or to crash (lock up). The problem is either insufficient logic or erroneous logic. For example, a program can crash if there are not enough validity checks performed on the input or on the calculations themselves, and the computer attempts to divide by zero. Bad instruction logic misdirects the computer to a place in the program where an instruction does not exist, and it crashes.

A program with bad logic may produce bad output without crashing, which is the reason extensive testing is required. For example, if the program is supposed to add an amount, but subtracts it instead, bad output results, although the computer keeps running.

Logic errors have the following basic characteristics;

- Program compiles and executes OK but produces unexpected or incorrect results.
- Detected by programmer (i.e You!)
- Hardest to detect, locate and find.

Define syntax errors (compilation error)

Syntax error is a programming error in which the grammatical rules of the language are broken. That is program errors that occur due to violation or disobedience of rules of the programming language. When syntax error occurs, the program execution is halt until the error or bug is detected, located and corrected. Syntax errors can be detected by the compiler, unlike semantic errors which do not become apparent until run-time.

Run-time errors have the following basic characteristics;

- Error in the form of statement: misspelled word, unmatched parenthesis, comma out of place
- Detected by the computer at compiler time
- Computer cannot correct error, so object program is not generated and thus program is not executed
- Computer (compiler) prints error messages, but continues to compile.

Linker errors: These types of errors have the following basic characteristics;

- Prevents the generation of an executable image
- Common linker errors;
 - specifying the wrong header file
 - disagreement among the function prototype, function definition and calls to that function

The difference between run-time, logical and syntax errors?

- Students should identify the differences from the above explanations.

Program maintenance

Program/software maintenance is the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment. This international standard describes the 6 software maintenance processes as:

1. The implementation processes contains software preparation and transition activities, such as the conception and creation of the maintenance plan, the preparation for handling problems identified during development, and the follow-up on product configuration management.
2. The problem and modification analysis process, which is executed once the application has become the responsibility of the maintenance group. The maintenance programmer must analyze each request, confirm it (by reproducing the situation) and check its validity, investigate it and propose a solution, document the request and the solution proposal, and, finally, obtain all the required authorizations to apply the modifications.

3. The process considering the implementation of the modification itself.
4. The process acceptance of the modification, by checking it with the individual who submitted the request in order to make sure the modification provided a solution.
5. The migration process is exceptional, and is not part of daily maintenance tasks. If the software must be ported to another platform without any change in functionality, this process will be used and a maintenance project team is likely to be assigned to this task.
6. Finally, the last maintenance process, also an event which does not occur on a daily basis, is the retirement of a piece of software.

Categories of Program maintenance

E.B. Swanson initially identified three categories of maintenance: corrective, adaptive, and perfective.

- Adaptive maintenance: Modification of a software product performed after delivery to keep a software product usable in a changed or changing environment.
- Perfective maintenance: Modification of a software product after delivery to improve performance or [maintainability](#).

Preventive maintenance: Modification of a software product after delivery to detect and correct latent faults in the software product before they become effective faults.

Difference between program maintenance and debugging

A common perception of maintenance is that it is merely fixing bugs. However, studies and surveys over the years have indicated that the majority, over 80%, of the maintenance effort is used for non-corrective actions. Key findings shows that program maintenance is really evolutionary developments and that maintenance decisions are aided by understanding what happens to systems (and software) over time. While Debugging is a very important task in the software development process, because an erroneous program can have significant consequences for its users. Some languages are more prone to some kinds of faults because their specification does not require compilers to perform as much checking as other languages. Use of a static analysis tool can help detect some possible problems.

WEEK 12

SPECIFIC LEARNING OUTCOMES

To understand:

*Employ structured approach to both flowcharting and program development.

THE CONCEPT OF GOOD PROGRAMMING PRACTICE

Structured Programming is a general term referring to programming that produces programs with clean flow, clear design, and a degree of modularity or hierarchical structure. Benefits of structured programming include ease of maintenance and ease of readability by other programmers.

Structured Programming is one step beyond modular programming with guidelines for “good” modules and “poor” modules. The structured programming guidelines also define “proper” flow of control and coding standards (such as indentation). In many large programming projects where statistics have been kept, it has been shown that structured programming has many demonstrable advantages over the old style, unstructured programs, such as:

1. Programs are more reliable. Fewer bugs appear in testing and later operation.
2. Programs are easier to read and understand
3. Programs are easier to test and debug.
4. Programs are easier to maintain.

Most commercial programming shops report that at least 50 percent of programmer time is spent making changes and correction in existing programs rather than developing new programs (some report more than 90 percent maintenance). Anything that will save time in correction and maintenance can save a company considerable money. It is easy to see why most commercial shops hiring programmers insist on structured programming techniques.

The current definition of structured programming includes standards for program design, coding and testing that are designed to create proper, reliable, and maintainable software. These standards include coding guidelines and rules for flow of control and module formation.

Structured Coding Guidelines

The structured coding guidelines are designed to make programs more reliable and easier to understand.

1. Use meaningful variable names
2. Code only one statement per line.
3. Use REMarks to explain program logic.
4. Indent and align all statements in a loop.
5. Indent the THEN and ELSE actions of an IF statement.

Flow of Control

In 1964, Italians Bohm and Jacopini proved mathematically that any program logic can be accomplished with just three control structures. Within a few years, studies were done declaring the GOTO statement to be harmful to good programming. In fact, in comparisons of selected large programming projects, there was a direct correlation between the number of GOTO statements and program bugs found.

BASIC was not designed as a structured language, but some of the current additions to the language now permit the programmer to adhere to the three “proper” constructs. All programming can be done with combinations of these three construct.

Iteration – This is the loop structure. The BASIC statement learned for looping are the WHILE/WEND.

Others include;

- Looping
- Do...Loop Statement
- For...Next Statement

Visual Basic allows a procedure to be repeated as many times as long as the processor could support. This is generally called looping .

Do...Loop Statement

Repeats a block of statements while a condition is **True** or until a condition becomes **True**.

Syntax

```
Do [{ While | Until } condition]  
[statements]  
[Exit Do]  
[statements]
```

Loop

Or, you can use this syntax:

```
Do  
[statements]  
[Exit Do]  
[statements]
```

```
Loop [{ While | Until } condition]
```

The **Do Loop** statement syntax has these parts:

Part	Description
Condition	Optional. Numeric expression or string expression that is True or False . If condition is Null, condition is treated as False .
Statements	One or more statements that are repeated while, or until, condition is True .

Remarks

Any number of **Exit Do** statements may be placed anywhere in the **Do...Loop** as an alternate way to exit a **Do...Loop**. **Exit Do** is often used after evaluating some condition, for example, **If...Then**, in which case the **Exit Do** statement transfers control to the statement immediately following the **Loop**.

When used within nested **Do...Loop** statements, **Exit Do** transfers control to the loop that is one nested level above the loop where **Exit Do** occurs.

Example

This example shows how **Do...Loop** statements can be used. The inner **Do...Loop** statement loops 10 times, sets the value of the flag to **False**, and exits prematurely using the **Exit Do** statement. The outer loop exits immediately upon checking the value of the flag.

```
Dim Check, Counter
Check = True: Counter = 0 ' Initialize variables.
Do ' Outer loop.
  Do While Counter < 20 ' Inner loop.
    Counter = Counter + 1 ' Increment Counter.
    If Counter = 10 Then ' If condition is True.
      Check = False ' Set value of flag to False.
      Exit Do ' Exit inner loop.
    End If
  Loop
Loop Until Check = False ' Exit outer loop immediately.
```

For...Next Statement

Repeats a group of statements a specified number of times.

Syntax

```
For counter = start To end [Step step]
[statements]
Exit For
[statements]

Next [counter]
```

The **For...Next** statement syntax has these parts:

Part	Description
Counter	Required. Numeric variable used as a loop counter. The variable can't be a Boolean or an array element.
Start	Required. Initial value of counter.
End	Required. Final value of counter.
Step	Optional. Amount counter is changed each time through the loop. If not

	specified, step defaults to one.
Statements	Optional. One or more statements between For and Next that are executed the specified number of times.

Remarks

The step argument can be either positive or negative. The value of the step argument determines loop processing as follows:

Value	Loop executes if
Positive or 0	counter <= end
Negative	counter >= end

After all statements in the loop have executed, step is added to counter. At this point, either the statements in the loop execute again (based on the same test that caused the loop to execute initially), or the loop is exited and execution continues with the statement following the **Next** statement.

Tip Changing the value of counter while inside a loop can make it more difficult to read and debug your code.

Any number of **Exit For** statements may be placed anywhere in the loop as an alternate way to exit. **Exit For** is often used after evaluating of some condition, for example **If...Then**, and transfers control to the statement immediately following **Next**.

You can nest **For...Next** loops by placing one **For...Next** loop within another. Give each loop a unique variable name as its counter. The following construction is correct:

```
For I = 1 To 10
  For J = 1 To 10
    For K = 1 To 10
      ...
    Next K
  Next J
Next I
```

Note If you omit counter in a **Next** statement, execution continues as if counter is included. If a **Next** statement is encountered before its corresponding **For** statement, an error occurs.

Example

This example uses the **For...Next** statement to create a string that contains 10 instances of the numbers 0 through 9, each string separated from the other by a single space. The outer loop uses a loop counter variable that is decremented each time through the loop.

```
Dim Words, Chars, MyString
For Words = 10 To 1 Step -1 ' Set up 10 repetitions.
  For Chars = 0 To 9 ' Set up 10 repetitions.
    MyString = MyString & Chars ' Append number to string.
  Next Chars ' Increment counter
  MyString = MyString & " " ' Append a space.
Next Words
```

Example

```
For counter=1 to 10
    display.Text=counter
Next
```

Example

```
For counter=1 to 1000 step 10
    counter=counter+1
Next
```

Example

```
For counter=1000 to 5 step -5
    counter=counter-10
```

Next

One Entry, One Exit.

The primary rule for program modules is the each module must have only one entry point and one exit point. So even though BASIC will allow a GOSUB to a line number within a subroutine and will allow multiple RETURN statements, such violations of the “one-entry, one exit” rule should be avoided.

The “Black Box” Concept

A “black box” (program module) is designed to accomplish a task. Generally, some data is input to the module, a transformation occurs, and data is output from the module. The details of what happens within the “black box” are not important to the overall program. What is important is that for a given input, the module will reliably produce the correct output. That module could be replaced by another – perhaps in another language such as assembler – without changing the rest of the program. It is important that each module “stand alone.”

Module Cohesion

Choosing the correct statements to combine into module is an important skill for programmers to develop. “Good” or “bad” module design is often an elusive concept when beginning to modularize programs.

Cohesion refers to the internal strength of a module. It is an indication of how closely related each of the statements in a module are to one another. As cohesion is increased, module independence, clarity, maintainability, and portability are increased.

Module Coupling

Coupling refers to the connections, or interfaces, between modules. As a general rule, modules should be loosely coupled; that is, what goes on inside one module should not affect the operation in other modules.

The control for execution of program modules must “come from above.” Looking at a hierarchy chart, a lower level module cannot determine what a higher level module should do – or even a module at the same level. For example, do not allow the detail read routine to determine that it is time to do final total calculations. That decision must be made by the mainline.

When a decision will determine what function to perform, place that decision at as high a level as possible.

Good programming practice

Good Programming Concept or Style makes code/program easier to maintain and modify. Maintaining and modifying code is made much easier by following a few often-overlooked techniques.

Whether the original programmer or someone else needs to make a change in the code the job is much easier if the original programmer used lots of comments, gave the variables and constants descriptive names, and sketched out the basic structure of the program at the very beginning in pseudo-code.

Using Comments in Code

The use of comments can mean the difference between code which any competent programmer can maintain or modify and a program that even the original programmer has trouble figuring out. Every routine should start with at least one comment that documents the purpose of the routine and any non-obvious dependencies or effects that the routine may have on other portions of the program.

In a development environment in which several programmers will be contributing code, adding a comment identifying the person who wrote the code will definitely help others to know who to ask if a question should arise.

Using Descriptive Names for Variables, Constants and Functions

It can be very tempting to use short names for variables and constants but it is not a good programming practice. A name such as *DateOfBirth* is much easier for other programmers to understand than *dob*.

Global variable names can all start with a lower case "g" so that any programmer looking at the code will instantly know which entities are local and which are global. Likewise, a lower case "k" can be the first letter of constants. This type of self-documenting code greatly reduces the need for comments and some typical errors.

Using Pseudo-code in Comments

When first developing the structure of a program it can be very helpful to write out the different routines in pseudo-code. This is language which resembles a cross between English and the programming language that the code will eventually be written in.

Using pseudo-code allows the programmer to concentrate on the conceptual aspects of the program without being distracted by syntax rules. The pseudo-code can also be the basis of the comments so it can server two purposes.

Using Modular Coding

Whenever possible, the lines of a routine should fit entirely on one screen of the editor. By keeping routines short, it is easier to comprehend them and see errors. Having short routines also forces the programmer to break each task into distinct sub-tasks, each of which is easier to maintain and modify in the future.

Modular coding also has the advantage of creating reusable routines that can be used in other programs. Once a routine is debugged and verified it is easier to copy and paste it into another program than to write it all over again.

Following these simple suggestions will make a programmer's code easier to maintain and modify. It may seem like more work, but in the end the net result is greater efficiency and fewer mistakes.

WEEK 13

SPECIFIC LEARNING OUTCOMES

To understand:

- * Employ program documents technique HIPS, data flow diagram, pseudo-cal.
- Explain graphic user interface, GUI.
- Define interactive processing.

Program documentation concepts

Program documentations

This is the act of keeping/maintaining all materials that serve primarily to describe a system/program and make it more readily understandable rather than to contribute in some way to actual operation of the system. Documentation is frequently classified according to purpose; thus for a given system there may be requirements documents, design document, and so on.

Why is program documentation important?

- 1) The main purpose of program documentation is to describe the design of your program. The documentation also provides the framework in which to place the code. as coding progresses, the code is inserted into the framework already created by the program documentation.
- 2) Documentation is important to tell other programmers what the program does and how it works. In the "real world" and in some classes here at BGSU, programmers often work in teams to develop code. Documentation helps others on the team to understand your work.
- 3) Maintenance and debugging are needed sooner or later for most programs and these are frequently done by someone other than the original programmer. Documentation can help the programmer who is making the modifications understand your code.
- 4) Documenting your program during development helps you to maintain your sanity.

When should program documentation be done?

When designing your program, you must spend time thinking about how to structure your program, what modules are needed, and the algorithms and processes you will use in the modules. You must think about what sort of data structures and objects (e.g., arrays, files or linked lists) are needed. This thinking must be done before you start coding, or you will find yourself wasting time writing useless code that is full of errors. It is very important to record this creative process so that the programmers that follow you do not duplicate work that you have already done.

Before writing the code, you should write the documentation to describe the design of each component of your program. Writing documentation for the modules before writing the code helps you define exactly what each module should do and how it will interact with other modules. Focusing on the design and the steps needed to solve the problem can help prevent errors in the completed program.

What information should be in the program documentation?

For an individual module, it is important to record (1) who has written the module, (2) when the module was written or modified, (3) why the module was written or modified, (4) how the module interacts with other modules, (5) what special algorithms were used, if any, and (6) acknowledge outside sources for ideas and algorithms.

For data structures, it is important to record (1) what data structure is used, (2) why a particular structure was used, (3) what data is contained in the structure, and (4) how the data structure is implemented.

Goals of good documentation:

1. Aid in designing good programs
2. Aid in debugging programs
3. Make programs clear and understandable once written
4. Make structure of program well-organized

Good documentation is a great aid to producing clear, well written, and understandable programs, and can save much programming and computing time. Good documentation is especially necessary for programming projects requiring either a long period of time by one programmer, any period of time by more than one programmer, or modifications to any code by another other than the original author. Good documentation techniques can be helpful in the following ways:

Program Design

Many beginning programmers seem to write programs in haphazard and unplanned ways, and often add comments only after the program is running. This method not only leads to poorly-structured programs, but also usually results in wasted time, and is not feasible except for relatively trivial programs.

A much better method is to write most of the overall comments with a flow chart first, specifying the structure and convention of the program, and then writing the program to fit. This usually leads to cleaner-coded, well-structured programs, which are produced in less time than those written by most novice programmers.

Program Debugging

Program debugging is aided by documenting a program before and during its creation, rather than afterward. Many mistakes can be avoided by having programming conventions well specified before writing the code. The very act of adding a comment to a statement often helps identify errors in the statement, because it forces the programmer to think about the function of the statement. Finally, good

documentation is useful if help is required from someone else, since it aids one in the understanding the program quickly. (It also makes other people much more willing to look at the program)

Program Modifications

Clear and complete documentation is absolutely invaluable with a program must be modified, especially if anyone but the original programmer is making the changes. It may be noted that useful programs tend to be modified often.

Program documentation techniques

When using an object oriented programming language, such as C++, programmers often create their own classes and then declare objects of these class types. These programs are frequently composed of several files — one or more header files containing class definitions, implementation files containing class functions, and a file containing the main program. The following describes what documentation should appear in each of these files.

Header files

Documentation in the header file must clearly describe the class interface. That is, the task performed by each member function should be described so that a client program which has declared objects of this class type will know exactly what this class can do. This documentation should be written so even a non-programmer can understand it. Header file documentation should appear before the class declaration statement and contain the following.

HIPS

Human Interactive Proofs (HIPs) are challenges meant to be easily solved by humans, while remaining too hard to be economically solved by computers. HIPs are increasingly used to protect services against automatic script attacks. To be effective, a HIP must be difficult enough to discourage script attacks by raising the computation and/or development cost of breaking the HIP to an unprofitable level. At the same time, the HIP must be easy enough to solve in order to not discourage humans from using the service. Early HIP designs have successfully met these criteria.

However, the growing sophistication of attackers and correspondingly increasing profit incentives have rendered most of the currently deployed HIPs vulnerable to attack. Yet, most companies have been reluctant to increase the difficulty of their HIPs for fear of making them too complex or unappealing to humans. The purpose of this study is to find the visual distortions that are most effective at foiling computer attacks without hindering humans. The contribution of this research is that we discovered that;

- i. Automatically generating HIPs by varying particular distortion parameters renders HIPs that are too easy for computer hackers to break, yet humans still have difficulty recognizing them, and
- ii. It is possible to build segmentation-based HIPs that are extremely difficult and expensive for computers to solve, while remaining relatively easy for humans.

HIPs, or Human Interactive Proofs, are challenges meant to be easily solved by humans while remaining too hard to be solved economically by computers. For instance, a HIP challenge (or HIP) could be a pixel image of distorted characters, and the proper response would be the ASCII string of corresponding characters. HIPs are increasingly used to protect services against automatic script attacks. Examples of such services include email (spam), online registrations (fraud, denial of service, or DoS), ticket/event reservations (DoS), online voting (stuffing), login (DoS), chat rooms, weblogs, etc.

Many companies such as Yahoo, Microsoft, TicketMaster, Register.com, and Google, are currently using HIPs to protect their online services. To be effective, a HIP must be difficult enough to discourage script attacks by raising the computation and/or development costs of breaking the HIP to an unprofitable level. At the same time, the HIP must be easy enough to not discourage humans from using the service.

Early HIP designs have successfully met these criteria. For instance, when MSN Hotmail deployed its first HIP, hotmail registrations dropped by 19% without impacting customer support inquiries. A study of the data revealed that the drop corresponded to mail accounts acquired by scripts for the purpose of spamming. However, the growing sophistication of attackers and increasing profit incentives have rendered most of the currently deployed HIPs vulnerable to attacks. Yet, most companies have been reluctant to increase the difficulty of their HIPs.

An example character based HIP for fear of making them too complex or unappealing to humans. This has raised an important question: Is it possible to design human-friendly HIPs that are easy for humans but difficult for computers? Work on distinguishing computers from humans traces back to the original *Turing test* which asks that a human distinguish between another human and a machine by asking questions of both. In contrast, we are interested in building a computer program designed to distinguish between another computer program and a human.

Such programs have been called reverse Turing tests, HIPs, or CAPTCHAs (Completely Automated Public Turing Tests to Tell Computer and Human Apart). An overview of this work can be found in. Construction of HIPs of practical value is difficult because it is not sufficient to develop challenges to which humans are somewhat more successful than machines. This is because the cost of failure from using machines to solve the puzzles may be very small. In practice, if one wants to block automated scripts, a challenge at which humans are about 90% successful and machines are 1% successful, may not be sufficient, especially when

the cost of failure and repetition is low for the machine. At the same time, the identical challenge must not put too much burden on the human in order to avoid discouraging the use of the service.


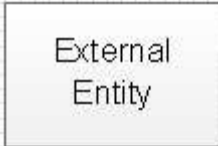
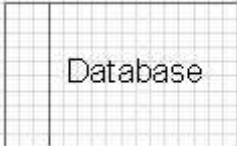
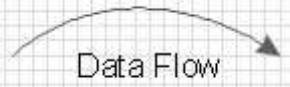
Data flow diagram

Data flow diagrams with Concept Draw PRO

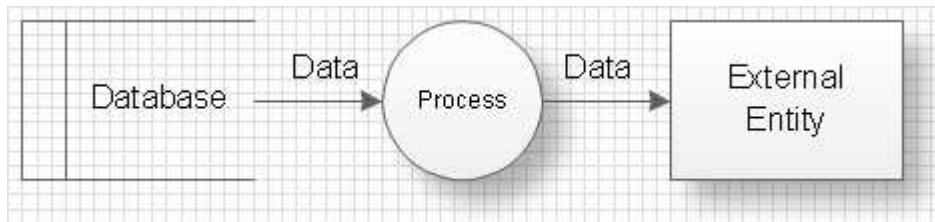
Data flow diagrams (DFD) are the part of the SSADM method (Structured Systems Analysis and Design Methodology), intended for analysis and information systems projection. Data flow diagrams are intended for graphical representation of data flows in the information system and for analysis of data processing during the structural projection. By means of data flow diagrams it is possible to show visually the work of the information system and results of this work. Data flow diagram visualizes processes, data depositories and external entities in information systems and data flows connecting these elements.

Data flow diagrams consist of following components:

- Processes and functions which represent actions happened in information system;
- External entities which represent in the system data ingoing and outgoing from it;
- Data depositories which represent places in system where data can be saved for definite period of time;
- Data flows, indicating direction and character of data flowing in the considered information system.

Diagram element	Graphical presentation
Process	
External Entity	
Data Store	
Data Flow	

This variant of presentation of data flow diagram objects got the name of Yourdon ? de Marco notation.



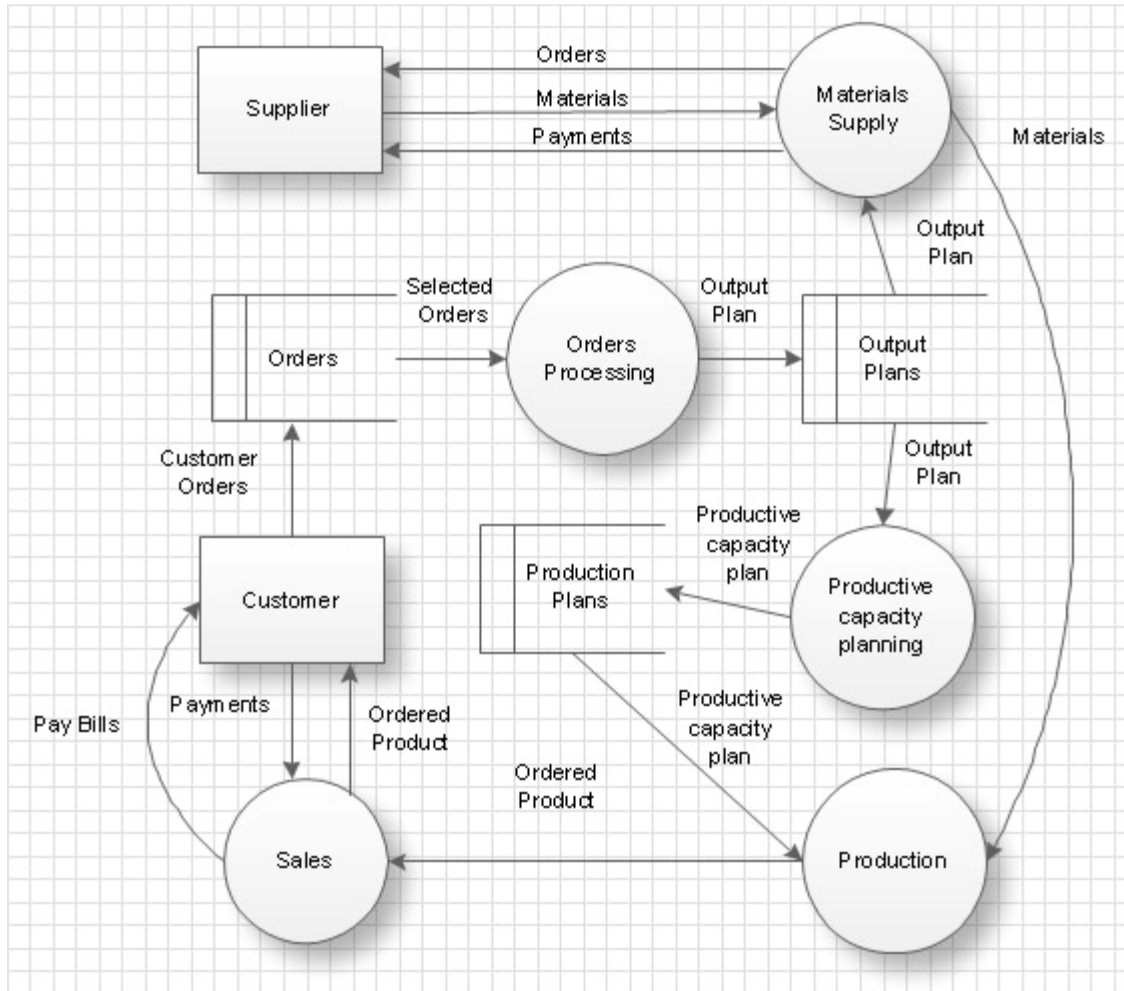
Objects of data flow diagrams are interpreted in the following way:

- Processes transform input data flows into output data flows;
- Data depositories serve only for keeping of ingoing data and do not change them;
- Data flows changes in external entities do not considered.

Every diagram object should have a name. Each data flow is denoted with indication of transferred data and with the possibility of indication of the format of these data. Data flow diagram should not involve more than 10 objects, excluding arrows, representing data flows. In case of more complicated system the totality of several objects (as a rule, processes) is united and represented on the diagram in the form of one object. This complicated compound object is presented in the form of a separate flows diagram. Each component has a number, at this, diagrams illustrating compound objects are numerated starting from the number of an object which they describe. For example components of the diagram of the description of an object with number 5 will be numerated 5.1, 5.2 etc.

For clearness there is a possibility to duplicate notations of used data depositories and external entities. Processes can't be duplicated. For example if one the same data depository is used in several different processes it is better to duplicate it on the diagram but not to create several intricate data flow arrows from one object to several processes. This concerns external entities as well. Duplicates of data depositories are marked with the double line from the left side, external entities duplicates - with the asterisk.

Data flows on the diagram can ramify and merge that implies branching or confluence of data in the information system.



Thus the information system is represented by a planner in the form of the high level DFD in which objects are worked out in details by diagrams of the lower level with the preset level of detailed elaboration. There is also another way of looking at which all events in the system are described at once and each event is represented in the form of process transforming data flows, further these subruns are grouped for getting diagrams of the higher level.

Pseudo – code

This section should describe, in an easily readable and modular form, how the software system will solve the given problem.

The term “pseudo code” is not intended to refer to a precise form of expression. Rather it refers to using standard English terms in a restricted manner to describe the algorithmic process involved. Good pseudo code must use a restricted subset of English, in such a way that it resembles a good high level programming language. Pseudo code must be formatted similarly to actual code. The pseudo code description of the problem should state the problem solution so clearly that it can easily be translated to the programming language to be used. Thus, it must include flow of control. The pseudo code for the system driver should

appear first. The pseudo code for subroutines in a system component should be grouped together, with the component identified.

Graphic user interface (GUI)

In [computer science](#) and [human-computer interaction](#), the *user interface (of a computer program)* refers to the graphical, textual and auditory information the program presents to the user, and the control sequences (such as keystrokes with the [computer keyboard](#), movements of the [computer mouse](#), and selections with the [touchscreen](#)) the user employs to control the program.

Currently (as of 2008) the following types of user interface are the most common:

- [Graphical user interfaces](#) accept input via devices such as computer keyboard and mouse and provide articulated graphical output on the computer monitor. There are at least two different principles widely used in GUI design: [Object-oriented user interfaces](#) (OOUIs) and [application oriented interfaces](#).
- **Web-based user interfaces** or **web user interfaces** (WUI) accept input and provide output by generating [web pages](#) which are transmitted via the [Internet](#) and viewed by the user using a [web browser](#) program. Newer implementations utilize [Java](#), [AJAX](#), [Adobe Flex](#), Microsoft .NET, or similar technologies to provide realtime control in a separate program, eliminating the need to refresh a traditional HTML based web browser.

Interactive processing

Definition: Interactive processing is the performance of tasks on a computer system that involves continual exchange of information between the computer and a user; the opposite of batch processing.

WEEK 14

SPECIFIC LEARNING OUTCOMES

To understand:

- * The concept of OO programming.
- the features of OO programming.
- the concept of properties, events, objects and classes.

Object oriented (OO) program

Object-oriented languages are outgrowths of functional languages. In object-oriented languages, the code used to write the program and the data processed by the program are grouped together into units called objects. Objects are further grouped into classes, which define the attributes objects must have.

A simple example of a class is the class Book. Objects within this class might be Novel and Short Story. Objects also have certain functions associated with them, called methods. The computer accesses an object through the use of one of the object's methods. The method performs some action to the data in the object and returns this value to the computer. Classes of objects can also be further grouped into hierarchies, in which objects of one class can inherit methods from another class. The structure provided in object-oriented languages makes them very useful for complicated programming tasks.

Features of OOP

Object-oriented programming (OOP) languages, such as C++ and Java, are based on traditional high-level languages, but they enable a programmer to think in terms of collections of cooperating objects instead of lists of commands. Objects, such as a circle, have properties such as the radius of the circle and the command that draws it on the computer screen. Classes of objects can inherit features from other classes of objects. For example, a class defining squares can inherit features such as right angles from a class defining rectangles. This set of programming classes simplifies the programmer's task, resulting in more "reusable" computer code. Reusable code allows a programmer to use code that has already been designed, written, and tested. This makes the programmer's task easier, and it results in more reliable and efficient programs.

Object-oriented programming began with Simula, a programming language developed from 1962 to 1967. Simula introduced definitive features of OOP, including objects and inheritance. Graphical user interface (GUI) is a feature that allows the user to select commands using a mouse. GUIs became a central feature of operating systems such as Macintosh OS and Windows.

Objects oriented programming languages

The most popular OOP language is C++, VB, JAVA, PASCAL, COBOL, Java, an OOP language that can run on most types of computers regardless of platform. In some ways Java represents a simplified version

of C++ but adds other features and capabilities as well, and it is particularly well suited for writing interactive applications to be used on the World Wide Web.

Java

Java is an object-oriented programming language. Java facilitates the distribution of both data and small applications programs, called applets, over the Internet. Java applications do not interact directly with a computer's central processing unit (CPU) or operating system and are therefore platform independent, meaning that they can run on any type of personal computer, workstation, or mainframe computer. This cross-platform capability, referred to as "write once, run everywhere," has caught the attention of many software developers and users. With Java, software developers can write applications that will run on otherwise incompatible operating systems such as Windows, the Macintosh operating system, OS/2, or UNIX.

To use a Java applet on the World Wide Web (WWW)—the system of software and protocols that allows multimedia documents to be viewed on the Internet—a user must have a Java-compatible browser, such as Navigator from Netscape Communications Corporation, Internet Explorer from Microsoft Corporation, or HotJava from Sun Microsystems. A browser is a software program that allows the user to view text, photographs, graphics, illustrations, and animations on the WWW. Java applets achieve platform independence through the use of a *virtual machine*, a special program within the browser software that interprets the bytecode—the code that the applet is written in—for the computer's CPU. The virtual machine is able to translate the platform-independent bytecode into the platform-dependent machine code that a specific computer's CPU understands.

Applications written in Java are usually embedded in Web pages, or documents, and can be run by clicking on them with a mouse. When an applet is run from a Web page, a copy of the application program is sent to the user's computer over the Internet and stored in the computer's main memory. The advantage of this method is that once an applet has been downloaded, it can be interacted with in real time by the user. This is in contrast to other programming languages used to write Web documents and interactive programs, in which the document or program is run from the server computer.

WEEK 15

Visual Basic

Step in Developing Applications

The visual Basic development environment makes building an application a straight forward process.

There are three primary steps involved in building a visual Basic application.

- (2) Draw the user interface by placing controls on the form.
- (3) Assign properties to controls.
- (4) Attach code to control events (and perhaps write other procedures)

Note:

- (2) These same steps are followed whether you are building a very simple application or one involving many controls and many lines of code.
 - (3) The event – driven nature of visual Basic allows you to build your application in stages and test it at each stage. You can build one procedure, or part of a procedure, at a time and try it until it works as described. This minimizes errors and gives you, the programmer, confidence as your application takes shape.
 - (4) As you progress in your programming skills, always remember to take above sequential approach to building a visual Basic application Build a little, test a little, modify a little and test again. You will quickly have a completed application.
- Microsoft VB is the fastest and easiest way to create applications for Microsoft windows.
 - The visual part refers to the method used to create the graphical user interface (GUI).
 - The Basic part refers to the BASIC Beginners All – Purpose symbolic construction code Language.

With VB an individual can build simple applications in minutes. VB enables you to write object oriented programmes or simple programs.

VB editions:- Learning editions
 Professional Edition
 Enterprises edition

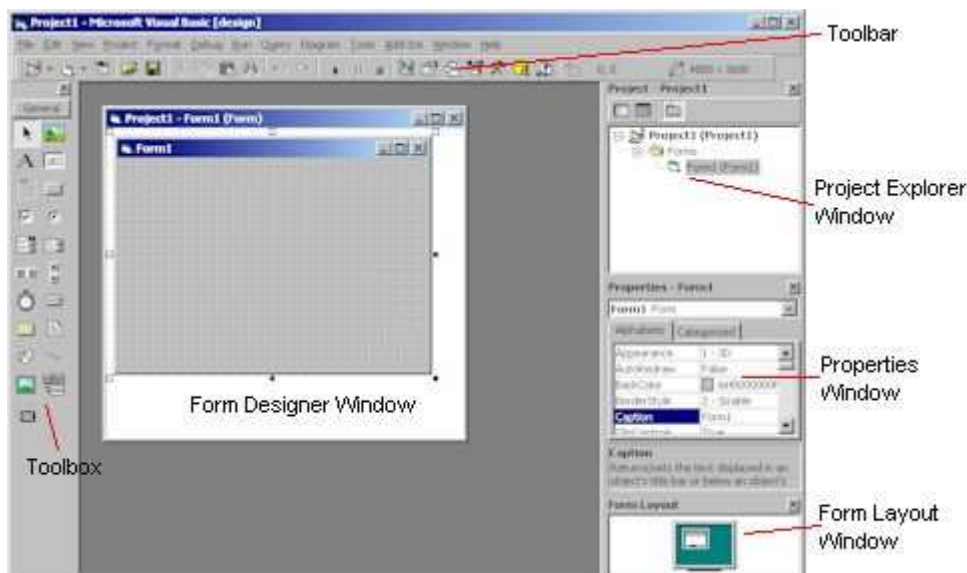
How windows work

Windows is an GUI operating system. With GUI it easily recognized graphic icons be selected using the mouse and commands chosen from menus, This is much easier for the user than typing in the specific lines of code that were required by MS-Dos in order to perform basic operations.

In GUI operating system, more than one application can be open at the same time. Processor time is shared between computing tasks and this called multitasking.

The Visual Basic environment

The Visual Basic environment is made up of several windows. The initial appearance of the windows on your screen will depend on the way your environment has been set up.



The tool bar The Visual Basic tool bar functions like the tool bar in any other Microsoft application. It provides shortcuts for many of the common operating commands. It also shows you the dimensions and location of the form currently being designed.



The tool box The tool box gives you access to the controls that you use on a form.



A control is an object such as a button, label or grid.

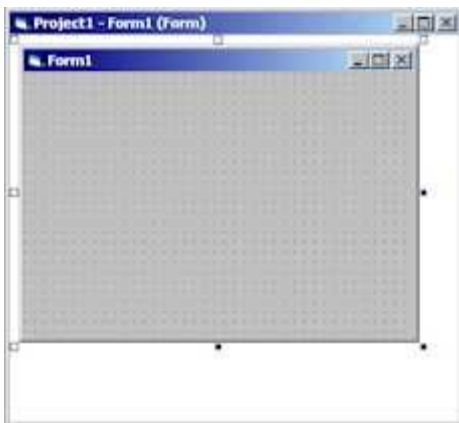
Controls are used on forms to display output or get input.

Each control appears as a button in the tool box. If the control you are looking for is not in the toolbox, select Components from the Project menu.

If the tool box is not displayed on your screen, or if at any time during the exercises you close it, choose Toolbox from the View menu.

The form designer window

This window is where you design the forms that make up your user interface.



If the form designer window is not displayed on your screen, or if at any time during the exercises you close it, choose Object from the View menu.

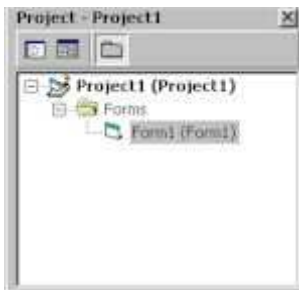
The properties window :

A form, and each control on it, has a set of properties which control its characteristics such as size, position and color.

The properties window lists all the properties a control has and their value. The default value of a property can be changed by setting the property value using the properties window when you design your application or changed by assigning a new value in code while your application is running. If the properties window is not displayed on your screen, or if at any time during the exercises you close it, choose Properties Window from the View menu.

The project explorer window

A project is a collection of the forms and code that make up an application. Each form in your application is represented by a file in the project explorer window.



A form file contains both the description of the screen layout for the form and the program code associated with it. If the project explorer window is not displayed on your screen, or if at any time during the exercises you close it, choose Project Explorer from the View menu.

The form layout window

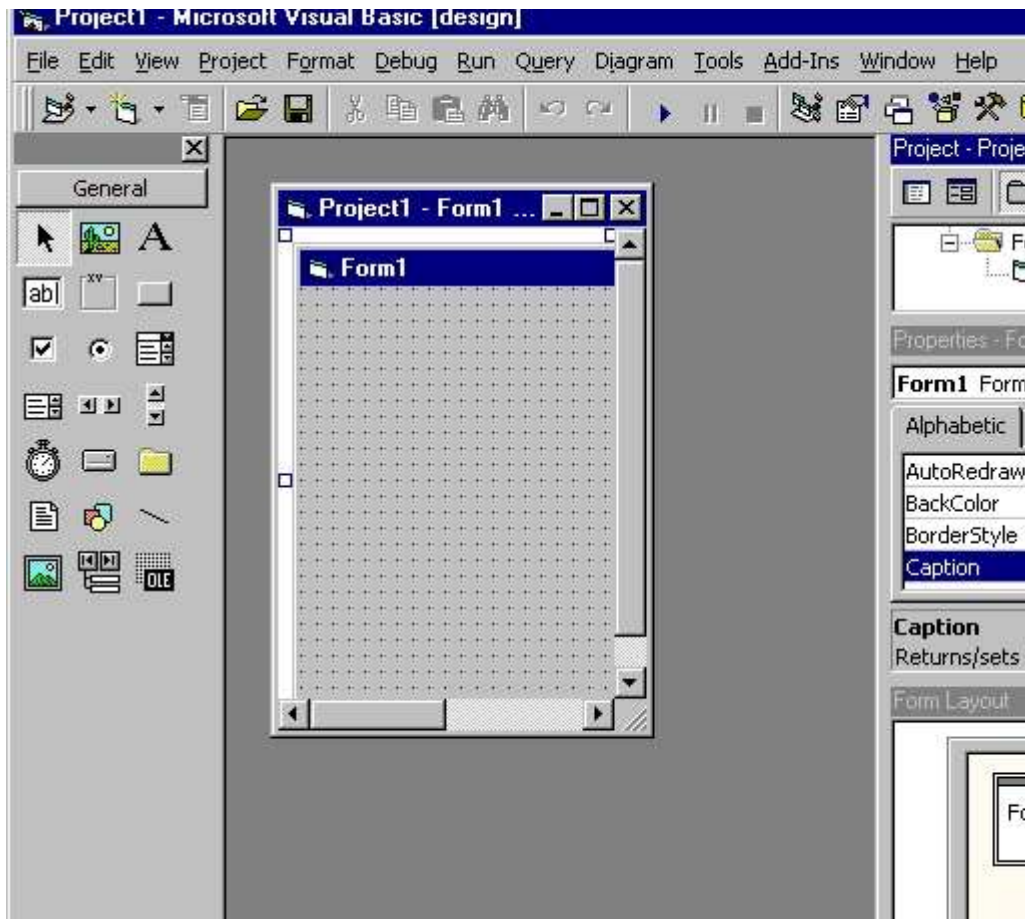
Move the form in the screen in this window to set the position of your form when your application is running.



You may wish to close the form layout window to allow more space for the properties window. To open the window again, select Form Layout Window from the View menu.

Starting Visual Basic

- From the **Windows Start** menu, choose **Programs, Microsoft Visual Studio 6.0**, and then Microsoft Visual Basic 6.0.
- Visual Basic 6.0 will display the following dialog box as shown in this figure



Stopping Visual Basic

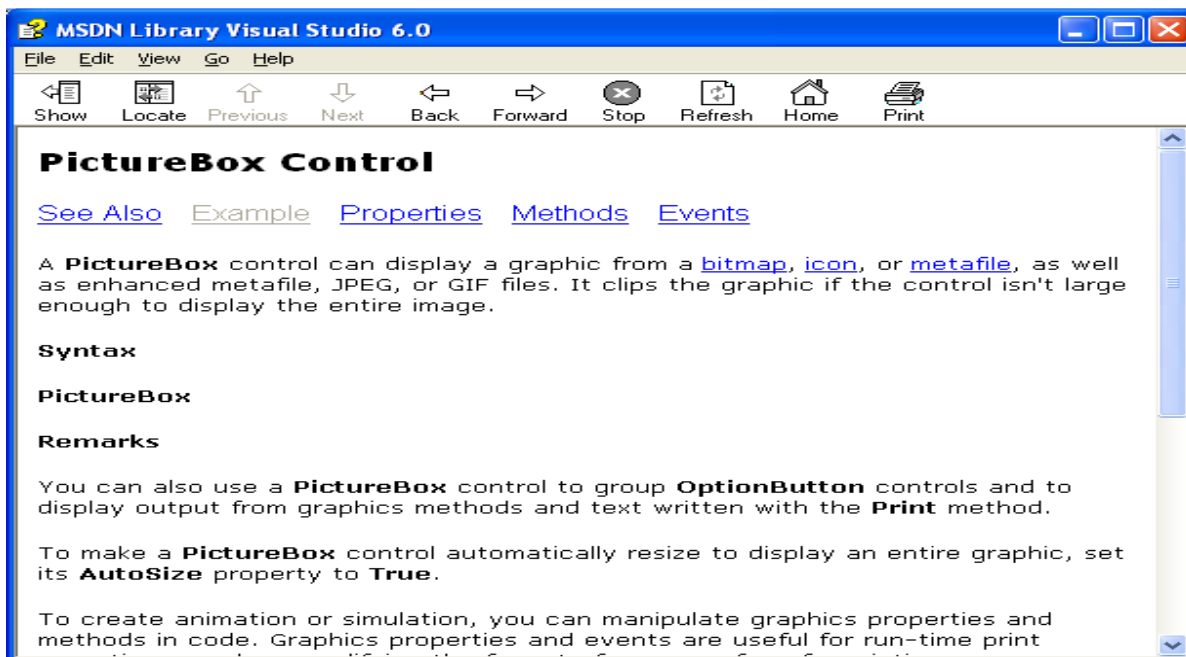
- From the **File** menu, choose **Exit** and then Microsoft Visual Basic 6.0. ask you to save changes in your project.

Getting online help

If you've used online help before, you may not think you need to read this section. Although you might be able to figure out Visual Basic's online help yourself, the help is fairly advanced and varies from most other

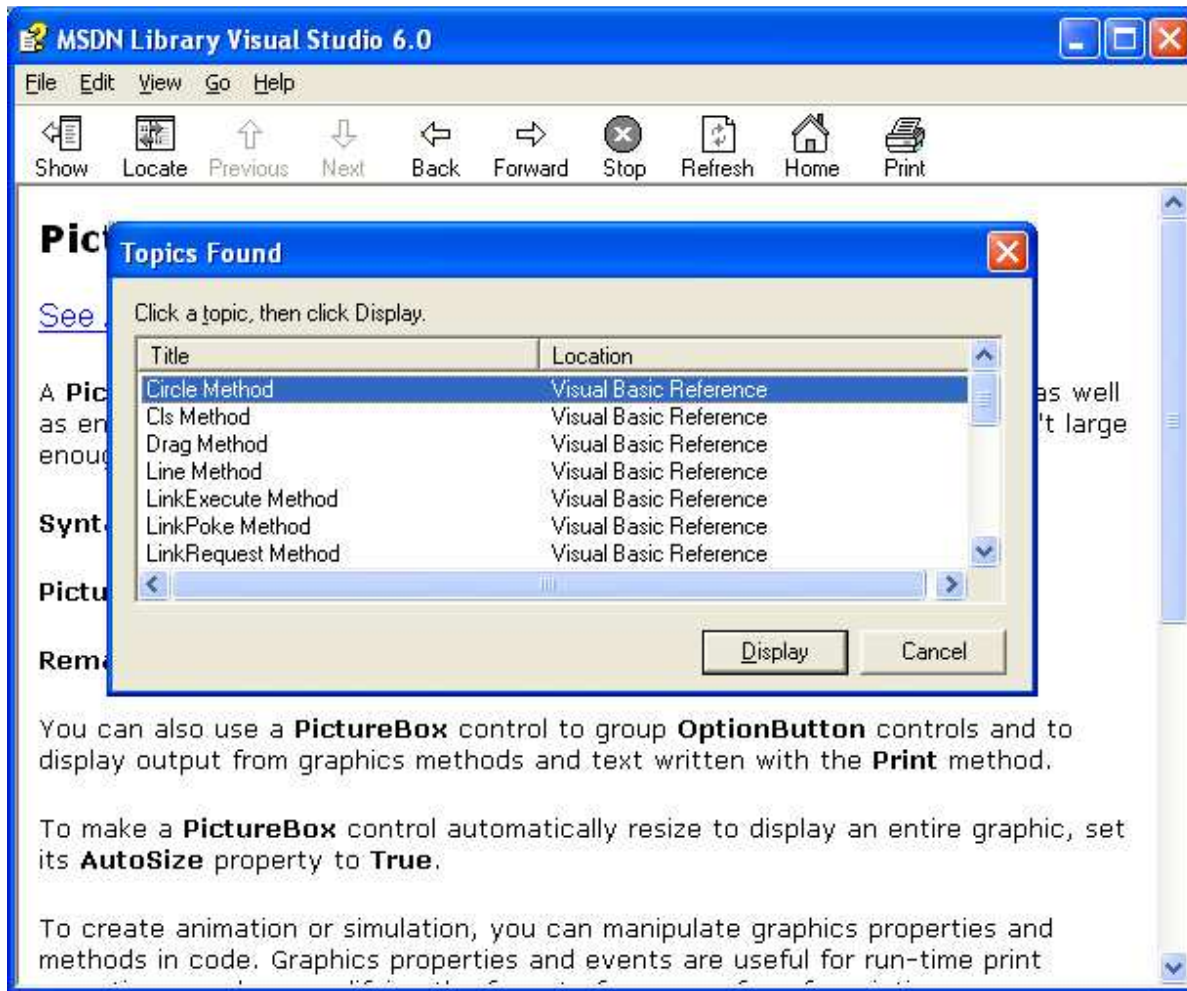
online help you may be used to. This topic section describes some of the help tools available from within Visual Basic.

The content-sensitive nature of Visual Basic's help system extends to almost every menu option, screen element, control, window, and language command. When you want help and aren't sure exactly where to turn first, press F1 and let Visual Basic give it a try. For example, if you think you need to use the Picture Box control but want to read a description first to make sure that you have the right control, click the Toolbox's Picture Box control and then press F1. Visual Basic sees that you've clicked the Picture Box and returns with the help screen shown in this figure



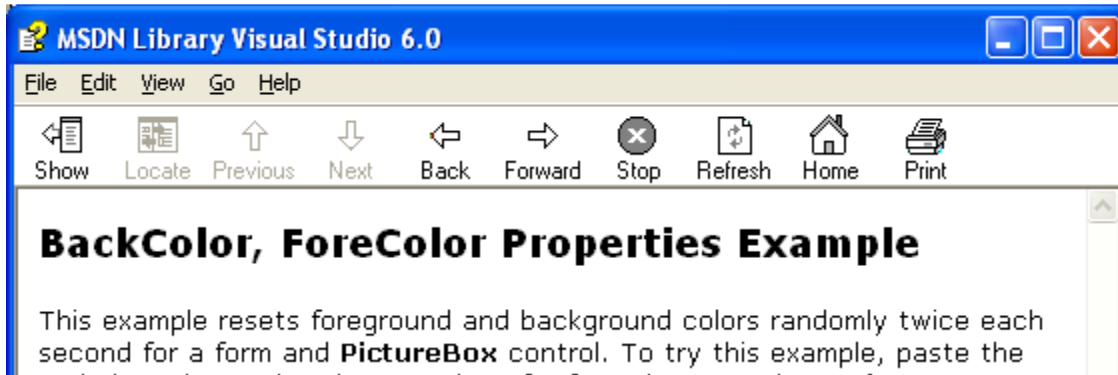
Click any screen element and press F1 for help

Throughout the help screens, Microsoft has scattered numerous links to related topics. When you click any underlined word or phrase inside a help window, Visual Basic responds with a pop-up definition or an additional help screen. Often, so may related topics appear throughout the help system that when you click a link, Visual Basic displays a scrolling Topics Found list, from which you can choose the description that most closely matches the topic you need.



Help links often provide several alternatives.

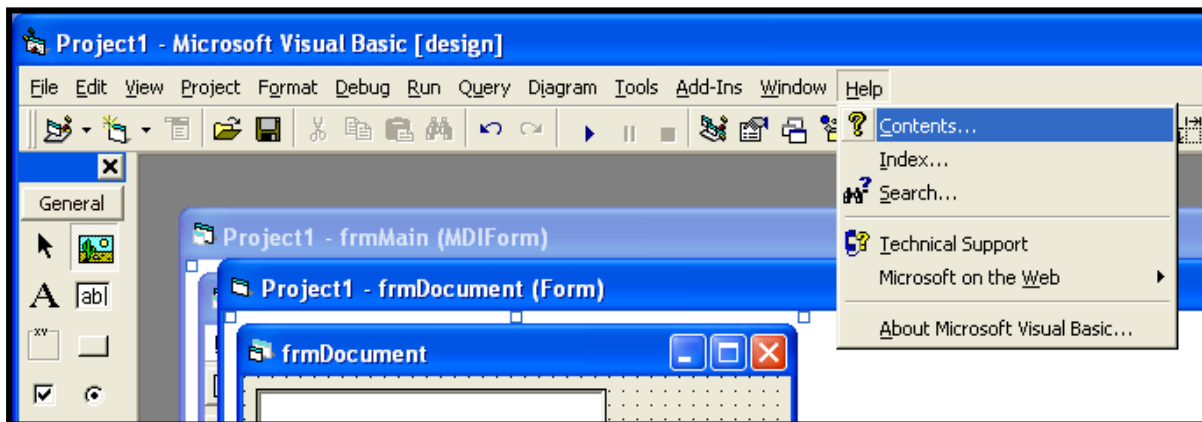
When you click an Example hypertext link, Visual Basic displays a window similar to the one shown in Figure. Although the help might look ambiguous at this point, you'll grow to appreciate the helpful suggestion when you begin learn the Visual Basic language. The Example help link shows you real Visual Basic language code that uses the item you've requested help for. As a programmer, you'll therefore see how to implement the item inside your own Visual Basic code by looking at the sample Visual Basic provides.



Visual Basic shows you sample code that uses the property or control.

The Help Menu

When you choose the first topic on the **H**elp menu, **M**icrosoft Visual Basic Topics, Visual Basic displays a help dialog box . This dialog box contains the usual Windows-like help tools. You can open and close the book icons on the Contents page to read about different Visual Basic topics. You can search for a particular topic in the index by clicking the Index tab. To locate every occurrence of a particular help reference word or phrase, you can click the Find tab to build a comprehensive help database that returns multiple occurrences of topics.

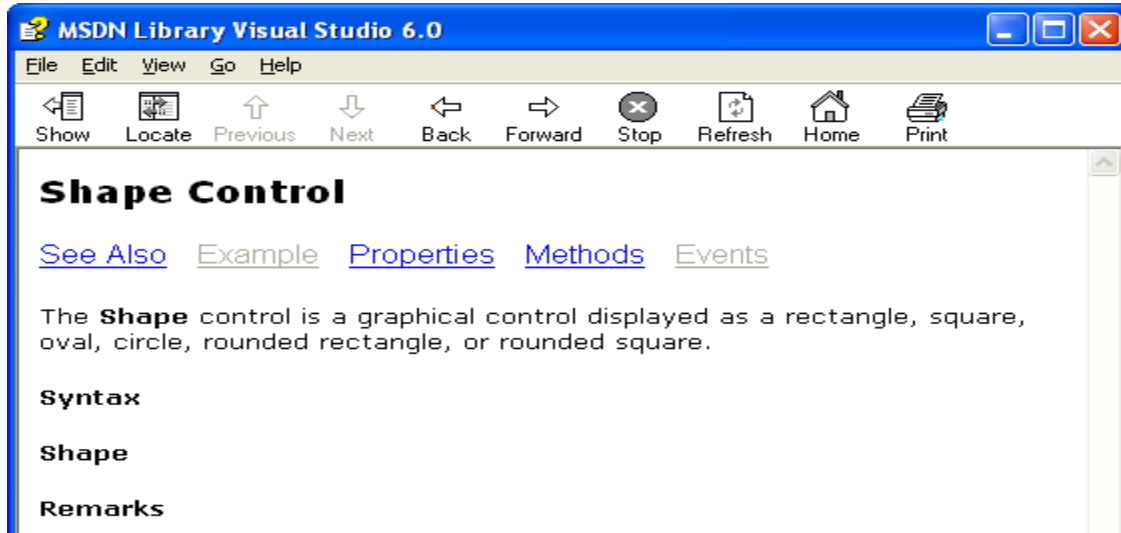


Example :

Get an instant definition for help links with a dotted underline.


Pop-up definition, Hyperlinks

Close the help window by clicking the window's Close button.

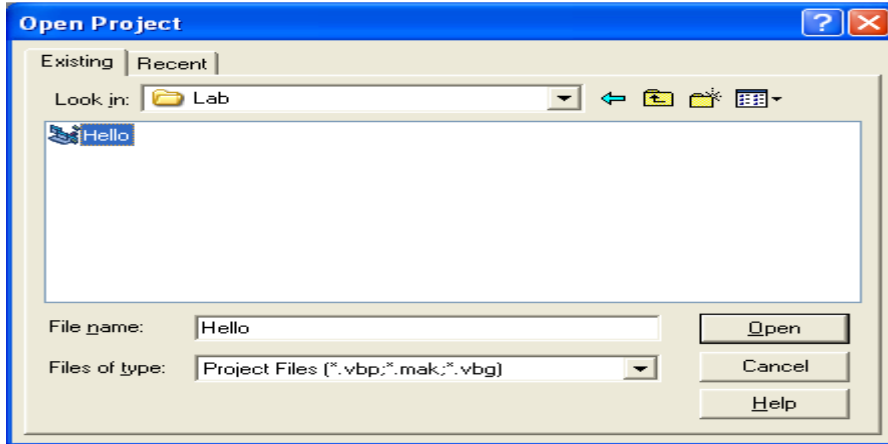


Opening Application

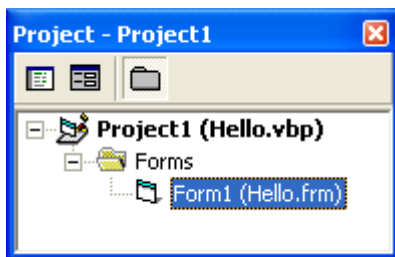
To open a project, you can do one of two things:

- Click File menu , Open project...
- Click the tool  and specify the project you want to open.

Then select Hello project and press Open.

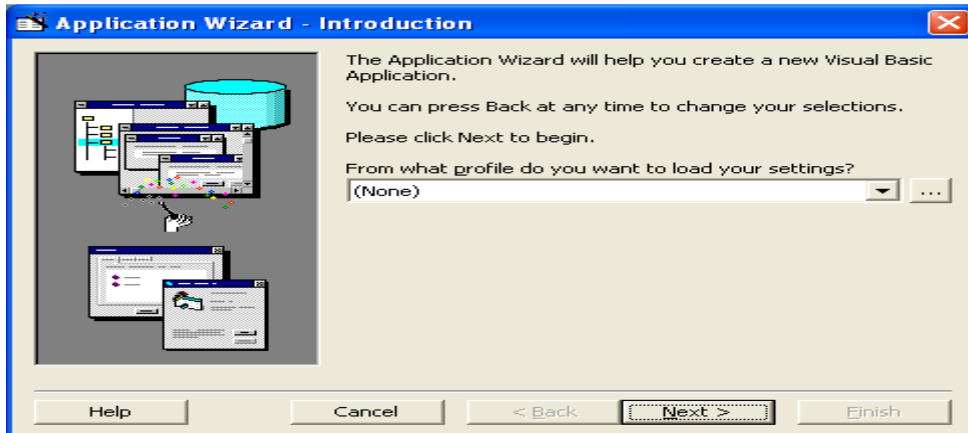


The project window will display the file “Hello.frm” from your project.



Creating Simple application (Wizard)

You start the application wizard from the New Project dialog box or by choosing New Project from the File menu. Click the VB Application Wizard icon to start the wizard. This Figure shows the application wizard's opening screen.



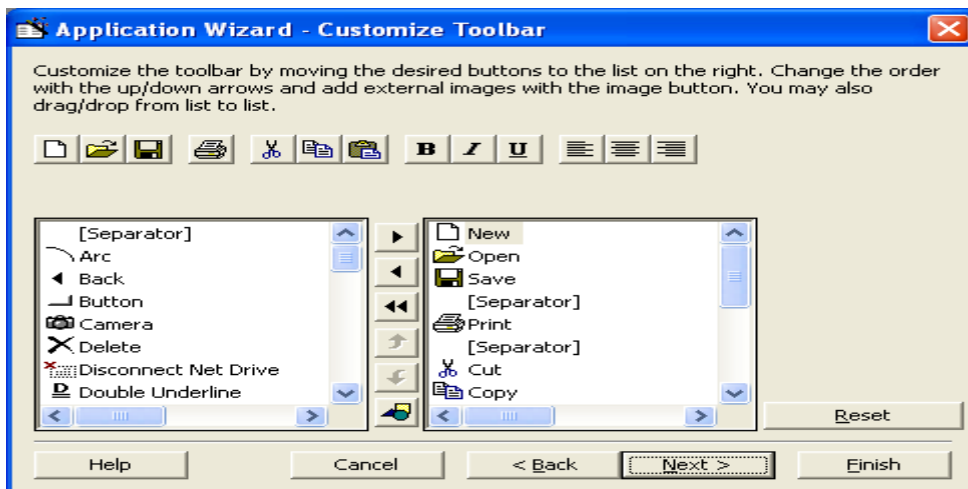
Example

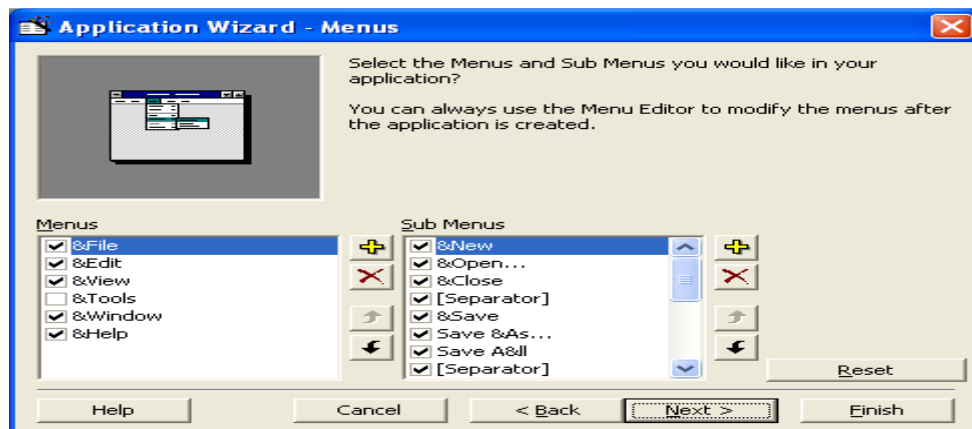
Assuming that you started the application wizard in the previous section, follow these steps to build your first application:

- 1- Click the Next button to display the Interface Type dialog box. The wizard can generate one of three types of user interfaces for the application you're generating:
 - MDI (Multiple Document Interface) lets you create a program window that contains embedded windows called child windows.
 - SDI (Single Document Interface) lets you create a program with one or more windows that exist at the same level (not windows within windows).
 - Explorer Style lets you create programs that somewhat take on the Books Online appearance, with a summary of topics or windows in a left pane and the matching program details in the right pane.
- 2- The MDI option should already be selected. If not, click the MDI option.



3- Click Next to display the menu selection dialog box. You can select certain menu options that will appear on your application's menu bar. By using the dialog box's options, you can help ensure that your application retains the standard Windows program look and feel. (You can add your own menu options after the wizard generates the program's initial shell.) For now, leave these options selected: File, Edit, Window, and Help.

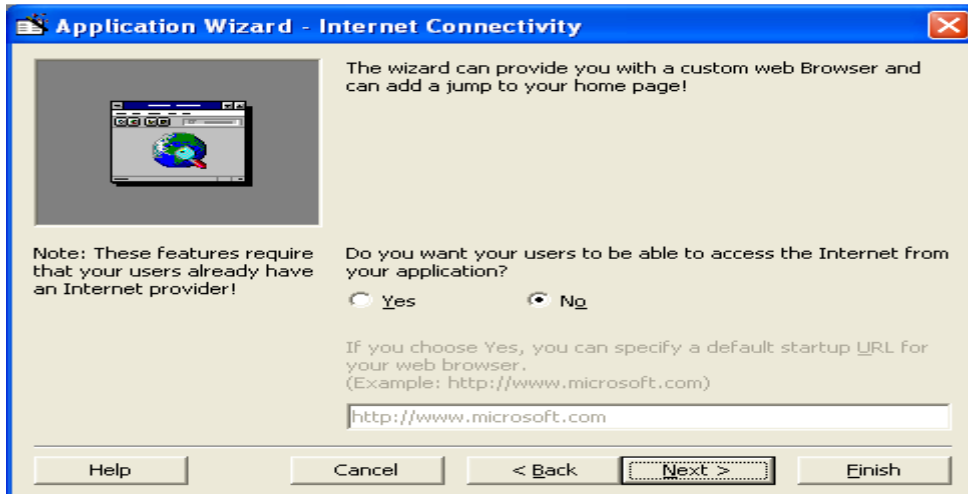




- 4- Click Next to display the wizard's Resources dialog box. A resource might be a menu, a text string, a control, a mouse cursor, or just about any item that appears in a program.

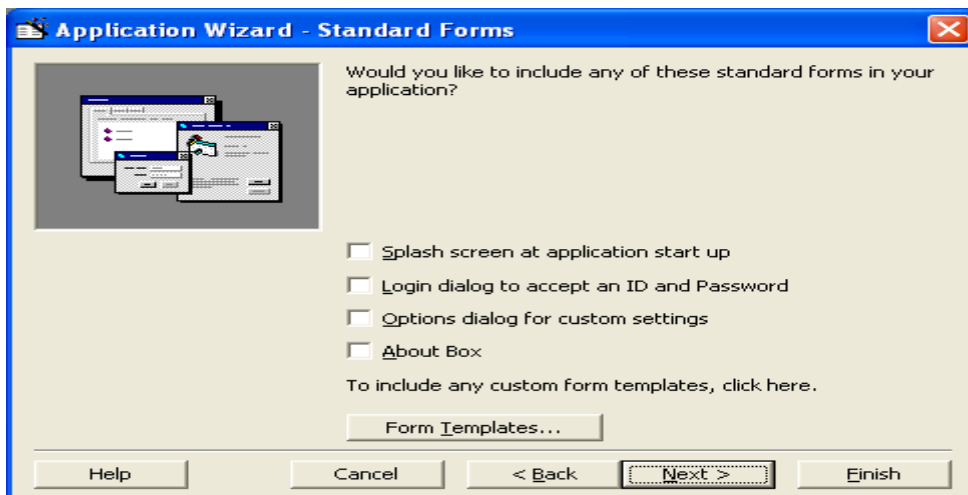


- 5- Click Next, you'll bypass the Internet connectivity dialog box because you don't need to add such connectivity to your first application shell.

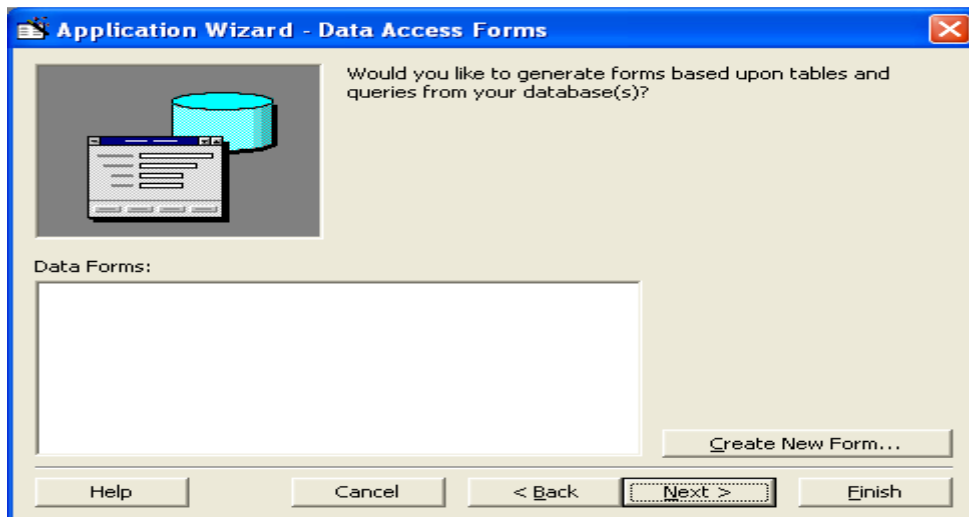


6- Determines which forms appear in your application:

- A splash form is an opening title form that your users see when they first run your application.
- A login form requests the user's ID and password, in case you want to add security features to your application.
- The options dialog box gives users the ability to modify certain application traits.
- The About box is accessed from most Windows Help menus and provides your program description and version.



8. Check the About Box but leave the other options unchecked.
9. Click the Next button twice to display the final application wizard dialog box. (You'll bypass the database access dialog box because you won't be retrieving database data in this first application.)



10. Click the Finish button. The wizard generates the application before your eyes. You'll see the wizard generating forms and titles; without the wizard, you would have to perform these steps yourself. When finished, the application wizard displays a dialog box to tell you that the application is completed.



11. Click OK to close the final application wizard dialog box. A summary report appears, to describe the generated program.



Running your application

Now that the form is complete you can see it in action by running it.

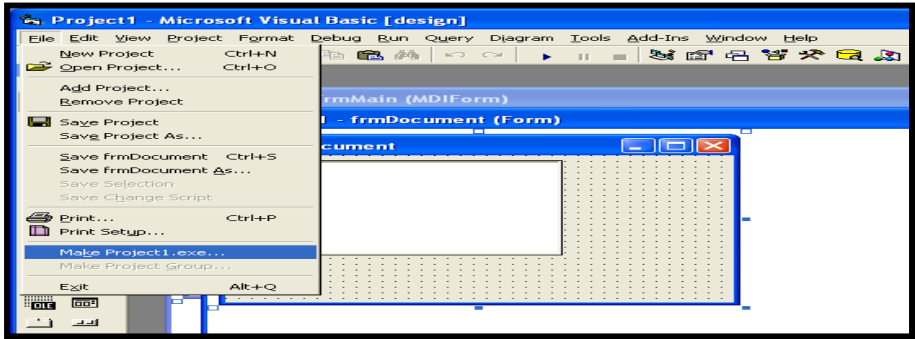
When you have written code for the buttons, running the application will allow you to activate the code. For now your buttons will not do anything.



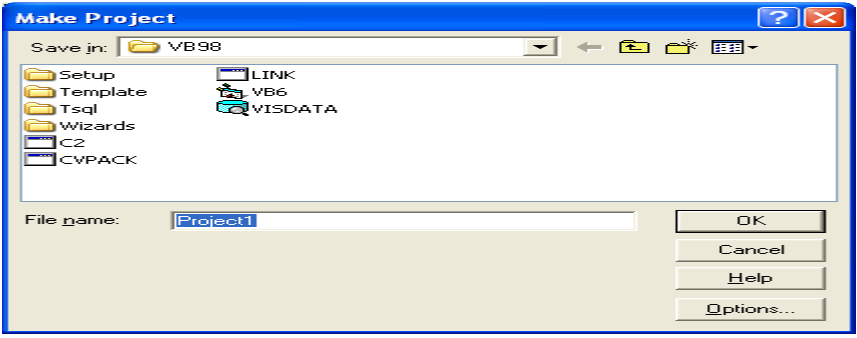
Your form will appear like a window from any other Microsoft application.

Creating Executable File

- Click File, Then Make Project1.exe...



- Specify the location and the name of the project, then click OK.



Saving your application

The last step in this chapter is to save your application so that you can use it for the exercises later in the book.



Save Button

Visual Basic first asks you to save the form and then the project file. Remember that each represents a separate file.



Specify the filename for the form as hello.frm. The file extension “frm” indicates that the file is a form file.



Always take care to ensure that you save all the files that make up a project.

List of Computer Programming Languages

ADA	Augusta Ada Byron (Lady Lovelace)	1979	Derived from Pascal, used primarily by the military.
ALGOL	ALGOrithmic Language	1960	First structured procedural programming language, used mainly for solving math problems.
APL	A Programming Language	1961	Interpreted language using a large set of special symbols and terse syntax. Used primarily by mathematicians.
BASIC	Beginners All-Purpose Symbolic Instruction Code	1965	Very popular high-level programming language, frequently used by beginning programmers.
C	Predecessor was Bell Laboratory's Programming Language	1972	Compiled, structured, programming language commonly used in many workplaces because its programs are easy to transfer between different types of computers.
C++	Advanced version of C. Developed at ATT Bell Labs.	1985	C++ is used in numerous fields, such as accounting and finance systems, and computer-aided design. Supports object-oriented programming.
COBOL	COmmon Business-Oriented Language	1959	English-like programming language, emphasizes data structures. Widely used, especially in businesses.
FORTH	FOuRTH-Generation language (4 GL)	1970	Interpreted, structured language, easily extended. Provides high functionality in limited space.

Fortran	FORMula TRANslation	1954	Initially designed for scientific and engineering uses, a high-level, compiled language now used in many fields. Introduced several concepts such as variables, conditional statements, and separately compiled subroutines.
HTML	HyperText Markup Language	1989	Designed for publishing hypertext on the Internet.
JAVA	Sun Microsystems developers drank a lot of coffee when coding for this.	1990	Originally developed for use in set-top boxes, transitioned to the World Wide Web in 1994.
LISP	LISt Processing	1960	A list-oriented programming language, mainly used to manipulate lists of data. Interpreted language, often used in research, generally considered the 'standard' language for Artificial Intelligence (AI) projects.
LOGO	Derived from Greek <i>logos</i> , meaning word	1968	Programming language often used with children. Features a simple drawing environment and several higher-level features from LISP. Primarily educational.
Modula-2	MODULAR Language, designed as secondary phase of Pascal (Niklaus Wirth devised both)	1980	Language that emphasizes modular programming. High-level language based on Pascal, characterized by lack of standard functions and procedures.
Pascal	Blaise PASCAL, mathematician and inventor of first computing device	1971	Compiled, structured language, based on ALGOL. Adds data types and structures while simplifying syntax. Like C language, it is a standard development language for microcomputers.
PERL	Practical Extraction and Report Language	1988	It is a text-processing language that looks like a combination of C and several Unix text processing utilities.
PILOT	Programmed Inquiry, Language Or Teaching	1969	Programming language used primarily to create applications for computer-aided instruction. Contains very little syntax.
PL/1	Programming Language One	1964	Designed to combine the key features of Fortran, COBOL, and ALGOL, a complex programming language. Compiled, structured language capable of error handling and multitasking, used in some academic and research environments.
SGML	Standard Generalized Markup Language	1986	Designed as a metalanguage, it is used as an international standard for the description of marked-up electronic text.
SQL	Structured Query Language	1986	Designed to be used for creating complex databases and accessing data in a relational database.
VB	Visual Basic	1990	Sometimes called the Rapid Applications Development system, is used to build applications quickly.
XML	Extensible Markup Language	1977	Used for creating arbitrarily-structured documents and Web pages; it is commonly associated with the Internet.