# Unit -1
# Introduction to Computer Programming

The ability to process the input according to the specified instruction, i.e. the user not only inputs data but also the instruction specifying what is to be done with the data. It is because of this property that a computer behaves as a general purpose machine a machine that can solve any problem if is instructed appropriately

***How computers can be used to solve problem***

1) Identify the problem to be solved identify & define
2) Develop a method for solving the problem –one must then find out a way (manually) to solve the problem. The solution is called an algorithm
3) Analyze the various algorithm to obtain the most suitable one
4) Express the solution in clear steps- many tools exist for this purpose such as flowcharts, decision tables, pseudo codes etc.
5) Translate the algorithm into a computer programming language also known as a computer program. programming languages are of such nature that both the computer and the programmer can understand what is written
6) The programmer is submitted to the computer for execution. The program is then translated into the machine (native) language of the computer in terms of current/ voltage and the same is submitted to the computer for execution.

The essence of the above discussion is that using a computer for problem
Solving involves 2 clear cut activities:-

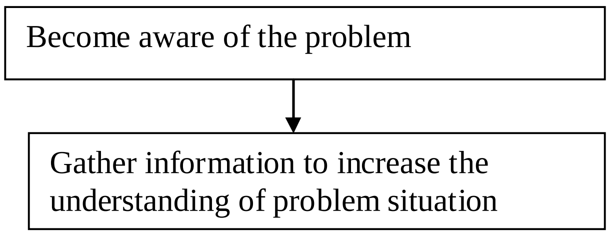
1) development of suitable algorithm
2) converting the algorithm into a program

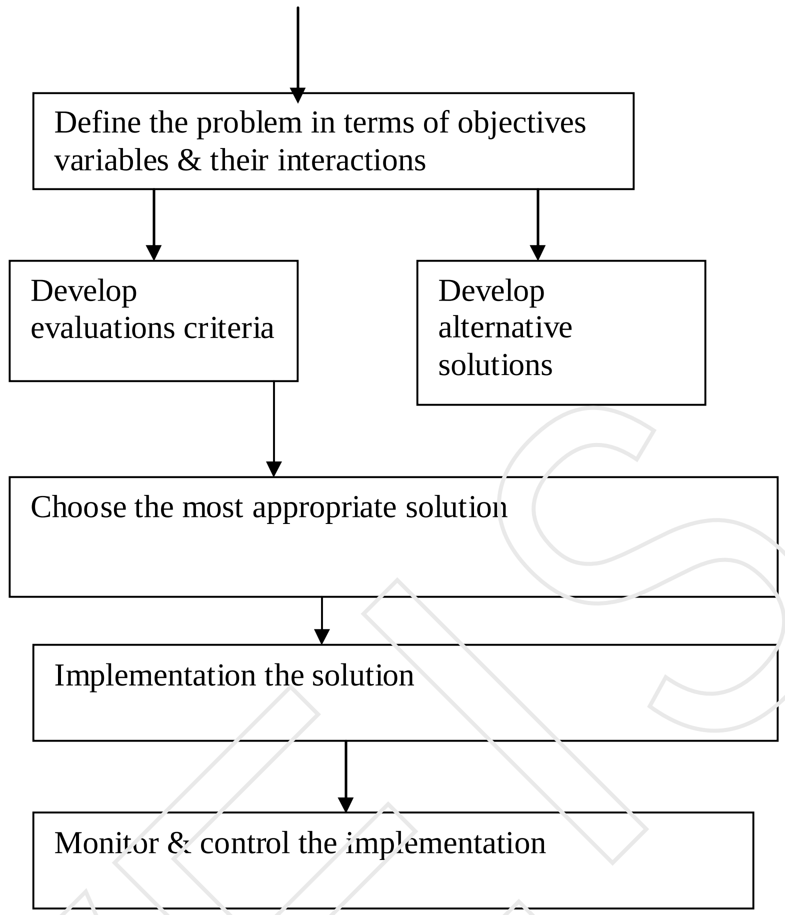
***Problem definition***

1) It is first and most important step in determining the information needs of an s/w can be determined by:
2) Studying the existing system
3) Using questionnaires
4) Setting the tentative information needs
5) Reviewing the more detailed statements of information needs with users

To use any of these steps, first the requirements are determined Analysis of t
The problem produces a clear accurate problem definition.

**Problem solving**


Become aware of the problem
Gather information to increase the understanding of problem situation


Define the problem in terms of objectives variables & their interactions
Develop evaluations criteria
Develop alternative solutions
Choose the most appropriate solution
Implementation the solution
Monitor & control the implementation

## Goals and objectives

All the systems, by definition have to have a purpose. The purpose is to accomplish meaningful goals & objectives. The goal is to solve the need of the user desired from the s/w.

1) **Mission**:- It is the broad statement of the purpose of the organization, to grow continuously through the effort involved by studying to be in organization
2) **Goals**: it is the general statement of what is to be accomplished. E.g., reduce time to respond to the effort without increasing the number of person's help.
3) **Objectives**:-It is the statement of measurable results to be achieved in a specified time frame. e.g., reduce order processing time from 5 to 3 days.

**Problem Identification and definition**

1) Identifying key persons involved
**2)** Identifying key variables & their inter-relationships
**3)** Specifying the objectives of the solution

**Similarities b/w problems**

1) **Speed of processing**:- the speed at which computer based data processing system can respond adds to these systems

VEIS

2) **Volume of work**: - the volume of information to be handled at each location should be calculated, both for normal & peak hours to avoid bottlenecks in operation.
The steps in this analysis are:-
- Calculate the number of messages flowing to & from every point in the system.
- Calculate the total transmission time using the data from above 2 points, plus the transmission speed of the type of communication being analyzed
- Calculate the volumes for the peak periods

3) **Reliability**: it includes
- The robustness of the design
- Availability of alternative computing facilities in the event of breakdown.
- The provision of sufficient equipment & staff to handle peak loads

4) **Accuracy**: this determines how accurately the data is being processed by the machine.

5) **Security**: - many aspects of security, those which particularly concern the systems Analyst is:-
a) Confidentiality
b) Privacy
c) Security of data

6) **Cost**: - It is associated with the activities of development as well as operation of data processing system. Operation includes evaluation & maintenance of the system by the users. Real cost of processing includes 5 different categories
a) Manpower
b) Hardware
c) Software
d) Consumables & supplies
e) Overhead costs

VEIS

# UNIT 2
# Algorithm Development

## Programming

To solve a computing problem, its solution must be specified in terms of sequence of computational steps such that they are effectively solved by a human agent or by a digital computer.

**Programming language**

1) the specification of the sequence of computational steps in a particular programming language is termed as a program
2) the task of developing programs is called programming
3) the person engaged in programming activity is called programmer

## Algorithm

### *Rules for developing algorithm*

There are no set rules for developing algorithms. While writing an efficient algorithm, the following points must be kept in mind

1) every procedure should carefully specify the input & output requirements
2) meaning of variables should be clearly defined
3) The flow of program should generally be forward except for normal looping & unavoidable instance.

### *Techniques of problem solving*

Problem solving an art in that it requires enormous intuitive power & a science for it takes a pragmatic approach.

Here a rough outline of a general problem solving approach.

1) Write out the problem statement include information on what you are to solve & consider why you need to solve the problem
2) Make sure you are solving the real problem as opposed to the perceived problem. To check to see that you define & solve the real problem
3) Draw & label a sketch. Define & name all variables and /or symbols. Show numerical values of variables if known.
4) Identify & name
    a. relevant principles, theories & equations
    b. system & subsystems
    c. dependent & independent variables

    d. known & unknowns
    e. Inputs & outputs
    f. Necessary information
5) List assumptions and approximations involved in solving the problem. Question the assumptions and then state which ones are the most reasonable for your purposes.
6) Check to see if the problem is either under-specified, figure out how to find the missing information. If over-specified, identify the extra information that is not needed.
7) Relate problem to similar problem or experience
8) Use an algorithm
9) Evaluate and examine and evaluate the answer to see it makes sense.

## **Top down Approach**

The top-down approach is based on the fact that large problems become more manageable if they are divided into a number of smaller ands simpler tasks which can be tackled separately. What is really required is that each of these parts has the properties of a module.
Top-down design approach is performed in a special way. The main program is written first. It is tested before sub programs are written. To do third, the actual sub programs are replaced with stubs. The stubs simply test to see if the data is passed correctly.

**Advantage**

• Easy to visualize functionality.
• Sense of completeness in the requirement.
• Easy to show the progress of development.

**Disadvantage**

• UI driven approach hence high possibility of redundant business logics.
• Since an UI is readily available no developer would write Unit test cases.
• No Concrete layer to rely on, as both presentation & Business Logic keep evolving.
• Lack of concrete test suits to ensure one layer is tied up.

## **Bottom up Approach**

A bottom up approach would be to write the most basic subroutine in the hierarchy first and then use them to make more sophisticated subroutines. The pure bottom-up approach is generally not recommended because it is difficult to anticipate which low level subroutines will be needed for any particular program. It can often be a useful first step to produce a library of basic functions and procedures before embarking on a major project...

**Advantage**

• Solid Business Logic, hence zero redundancy
• Good Unit test case can be written to validate changes.

• Developer has only option to use unit testing tools to test the Logic.
• Easy to manage changes and modification.

**Disadvantage**

• Effort involved to write test cases.
• Progress of implementation cannot be show very effectively.

VEIS

## Techniques for the design of algorithm

### Flowchart

Flowcharts are maps or graphical representations of a process. Steps in a process are shown with symbolic shapes, and the flow of the process is indicated with arrows connecting the symbols. Computer programmers popularized flowcharts in the 1960's, using them to map the logic of programs. In quality improvement work, flowcharts are particularly useful for displaying how a process currently functions or could ideally function. Flowcharts can help you see whether the steps of a process are logical, uncover problems or miscommunications, define the boundaries of a process, and develop a common base of knowledge about a process. Flowcharting a process often brings to light redundancies, delays, dead ends, and indirect paths that would otherwise remain unnoticed or ignored. But flowcharts don't work if they aren't accurate, if team members are afraid to describe what actually happens, or if the team is too far removed from the actual workings of the process.

A flowchart (also spelled flow-chart and flow chart) is a schematic representation of a process. They are commonly used in business/economic presentations to help the audience visualize the content better, or to find flaws in the process.

#### Types of Flowcharts

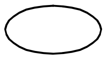
There are four basic types of flowcharts: Basic, Process. Deployment, and Opportunity.

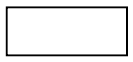
**Basic** flowcharts quickly identify all the major steps in a process. They are used to orient a team with
The major steps by just giving a broad overview of the process.

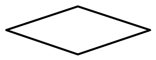
**Process** flowcharts examine the process in great detail. They provide a comprehensive listing of all
The major and sub-steps in a process.

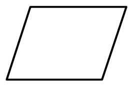
**Deployment** flowcharts are similar to Process flowcharts in that they are very detailed but also indicate the people who are involved in the process. This could be very useful when the process involves cooperation between functional areas.
**Opportunity** flowcharts highlight decision step and check point. They are used for very complicated

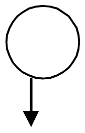
*Processes because they highlight specific opportunities for improvement.*

 **Ovals** are used to represent starting and ending points to the flowchart process.


**Rectangles** are used to describe an action taken or a task completed.


**Diamonds** contain questions requiring a “Yes” or “No” decision.


**Data Input/Output** uses a skewed rectangle to represent a point in the process where data is
entered or retrieved.


**Connectors**

**Flow Chart Example 1**

Draw a flowchart to find the sum of first 50 natural numbers.

Answer: The required flowchart is given in Fig. 1.

VEIS

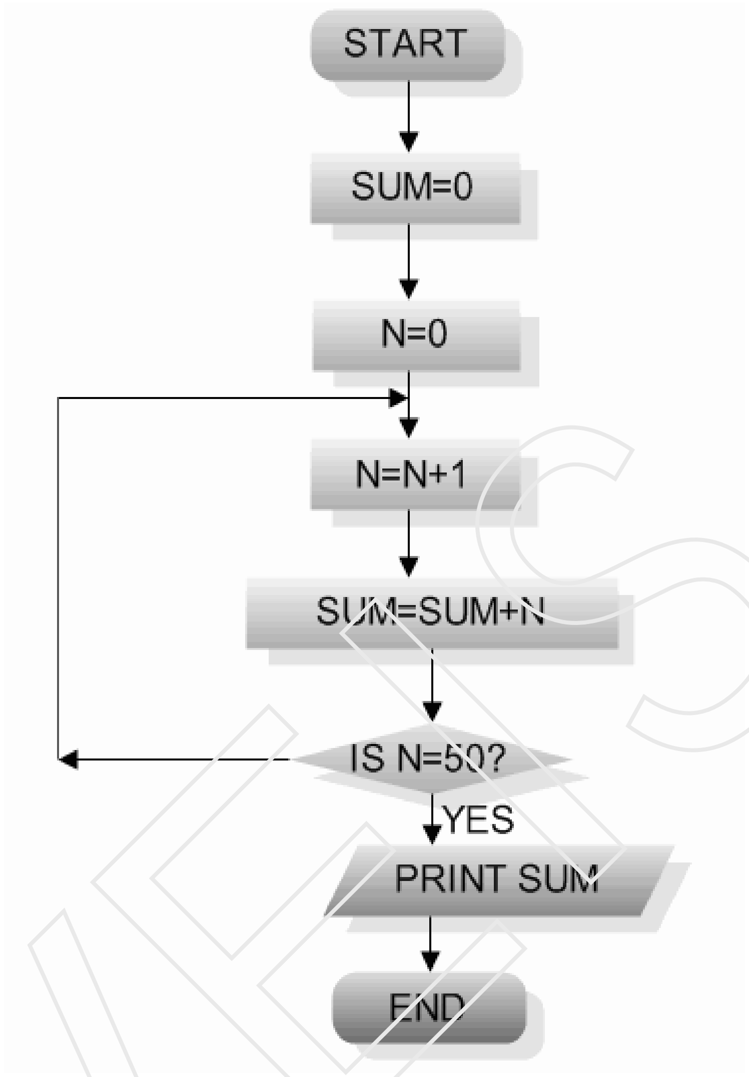


**Fig. 1 Sum of first 50 natural numbers**

VES

**Flow Chart Example 2**

Draw a flowchart to find the largest of three numbers A, B, and C.

Answer: The required flowchart is shown in Fig 2

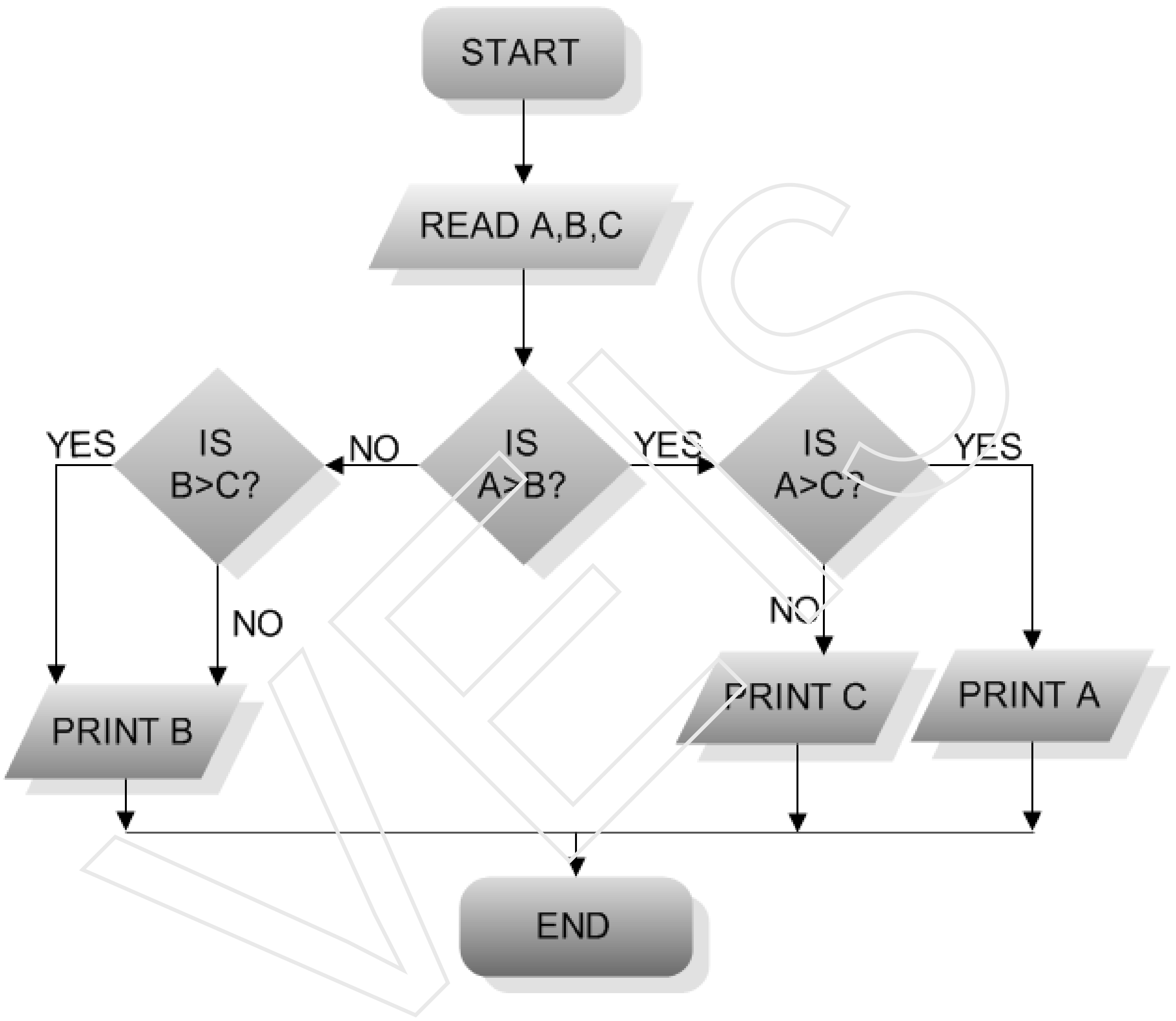


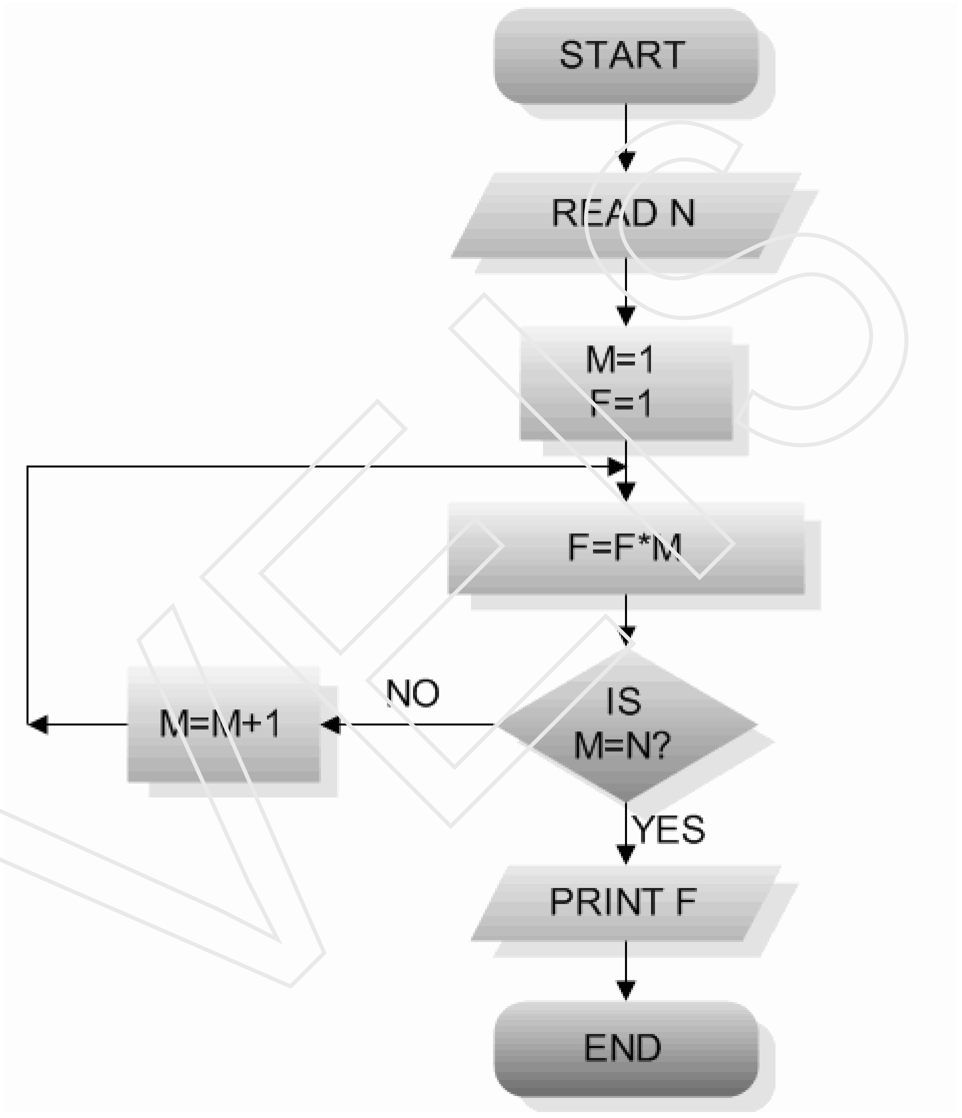
**Fig 2 Flowchart for finding out the largest of three numbers**

**Flowchart Example 3**

Draw a flowchart for computing factorial N (N!)

Where N! = 1?2?3?....N .

The required flowchart has been shown in fig 3



## Pseudo code

Pseudo code is a kind of structured English for describing algorithms. It allows the designer to focus on the logic of the algorithm without being distracted by details of language syntax. At the same time, the pseudo code needs to be complete. It describes the entire logic of the algorithm so that implementation becomes a rote mechanical task of translating line by line into source code. Pseudo code cannot be compiled nor executed, and there is no real formatting or syntax rules. It is simply one step - an important one - in producing the final code. The benefit of pseudo code is that it enables the programmer to concentrate on the algorithms without worrying about all the syntactic details of a particular programming language. In fact, you can write pseudocode without even knowing what programming language you will use for the final implementation.

Some of the conventions which are used while writing pseudocodes are as follows:

1) All statements in a loop should be indented.
2) 2) All alphanumeric values should be enclosed in single or double quotes.
3) The beginning and end of a pseudocode is marked with keywords like start and end respectively.
4) All statements must include certain key words which denote an operation.

The Input Statement

The following verbs can be used to accept or input data from the keyboard of from an existing form like a file.

Accept or Read

For example:

Accept Name

Read Name

The output Statement

The following verbs can be used to display output

Write name

Display name

Ques. Accept two numbers add them and display the result

```
Start
Accept n1
Accept n2
Sum=n1+n2
```

VEIS

```
Display sum
End
```

**Ques. A student appears for a test in 3 students. Each test is out of 100 marks. The percentage of each student has to be calculated and depending on the percentage calculated, grades are given as under:**

| Percentage | Grade |
|---|---|
| >=80 | A |
| 50-79 | B |
| <50 | F |

**Answer**

```
Start
Accept m1,m2 ,m3
Per=(m1+m2+m3)/3
If per >=80
        Grad='a'
Else
        If per >=50 and per<80
                Grad='b'
Else
                Grad='f'
End if
End if
Display grad
End
```

**Example of loop in pseudo code**

**Ques Write pseudocode to find the factorial**

**Answer**

```
Start
```

VEIS

```
Display 'enter a value for n'
Accept n
Fact=1
While n!=0
Equal to
Do
        Fact=fact*n
N=n-1
Enddo
Display 'the factorial value is:',fact
End
```

## Translating Algorithm into program

Next step in programming after designing a suitable algorithm is that of translating the algorithm into a suitable computer program. The different approaches to accomplish this task are discussed here under:

### Linear Programming

Linear program is a method for straightforward programming in a sequential manner. This type of programming does not involve any decision making. General model of these linear programs is:

1. Read a data value
2. Computer an intermediate result
3. Use the intermediate result to computer the desired answer
4. Print the answer
5. stop

## Structured Programming

Structured programming (sometimes known as *modular programming*) is a subset of procedural programming that enforces a logical structure on the program being written to make it more efficient and easier to understand and modify. Certain languages such as Ada, Pascal, and dBase are designed with features that encourage or enforce a logical program structure.

Structured programming frequently employs a top-down design model, in which developers map out the overall program structure into separate subsections. A defined function or set of similar functions is coded in a separate module or sub module, which means that code, can be loaded into memory more efficiently and that modules can be reused in other programs. After a module has been tested individually, it is then integrated with other modules into the overall program structure.

## <u>Advantages of Structured Programming</u>

**Easy to write:**

Modular design increases the programmer's productivity by allowing them to look at the big picture first and focus on details later. Several Programmers can work on a single, large program, each working on a different module. Studies show structured programs take less time to write than standard programs. Procedures written for one program can be reused in other programs requiring the same task. A procedure that can be used in many programs is said to be **reusable**

**Easy to debug**
Since each procedure is specialized to perform just one task, a procedure can be checked individually. Older unstructured programs consist of a sequence of instructions that are not grouped for specific tasks. The logic of such programs is cluttered with details and therefore difficult to follow.
**Easy to Understand**

The relationship between the procedures shows the modular design of the program.
Meaningful procedure names and clear documentation identify the task performed by each module. Meaningful variable names help the programmer identify the purpose of each variable.

**Easy to Change**
since a correctly written structured program is self-documenting; it can be easily understood by another programmer

## Structured Programming Constructs

Use only three constructs

- sequence (statements, blocks)
- selection (if, switch)
- iteration (loops like while and for)

## Sequence

_ any valid expression terminated by a semicolon is a statement.
_ Statements may be grouped together by surrounding them with a pair of curly braces .
_ Such a group is syntactically equivalent to one statement and can be inserted wherever one statement is legal.

## Selection

The selection constructs allow us to follow different paths in different situations. We may also think of them as enabling us to express decisions.
The main selection construct is
**if (*expression*)**

***statement*1**
**else**
**. *statement*2**

*statement*1 is executed if and only if *expression* evaluates to some non-zero number. If *expression* evaluates to 0, *statement*1 is not executed. In that case, *statement*2 is executed.
if and else are independent constructs, in that if can occur without else (but not the reverse).Any else is paired with the most recent else-less if, unless curly braces enforce a different scheme. Note that only curly braces, not parentheses, must be used to enforce the pairing. Parentheses

## Iteration

Looping is a way by which we can execute any some set of statements more than one times continuously .In c there are mainly three types of loops are use :

- while Loop
- do while Loop
- For Loop

The control structures are easy to use because of the following reasons:
1) They are easy to recognize
2) They are simple to deal with as they have just one entry and one exit point
3) They are free of the complications of any particular programming language

### Modular Design of programs

One of the key concepts in the application of programming is the design of a program as a set of units referred to as blocks or modules. A style that breaks large computer programs into smaller elements called modules. Each module performs a single task; often a task that needs to be performed multiple times during the running of a program.

Each module also stands alone with defined input and output.

Since modules are able to be reused they can be designed to be used for multiple programs. By debugging each module and only including it when it performs its defined task, larger programs are easier to debug because large sections of the code have already been evaluated for errors. That usually means errors will be in the logic that calls the various modules.

Languages like Modula-2 were designed for use with modular programming. Modular programming has generally evolved into object-oriented programming.

Programs can be logically separated into the following functional modules:

1) Initialization
**2)** Input
**3)** Input Data Validation
**4)** Processing

5) Output
6) Error Handling
7) Closing procedure

*Basic attributes of modular programming*

- Input
- Output
- Function
- Mechanism
- Internal data

## Control Relationship between modules

The structure charts show the interrelationships of modules by arranging them at different levels and connecting modules in those levels by arrows. An arrow between two modules means the program control is passed from one module to the other at execution time. The first module is said to call or invo0ke the lower level modules.

*There are three rules for controlling the relationship between modules.*

1) There is only one module at the top of the structure. This is called the root or boss module.
2) The root passes control down the structure chart to the lower level modules. However, control is always returned to the invoking module and a finished module should always terminate at the root.
3) There can be more than one control relationship between two modules on the structure chart, thus, if module A invokes module B, then B cannot invoke module A.

### Communication between modules

1) **Data :** Shown by an arrow with empty circle at its tail.
2) **Control :** Shown by a filled-in circle at the end of the tail of arrow

### Module Design Requirements

A hierarchical or module structure should prevent many advantages in management, developing, testing and maintenance. However, such advantages will occur only if modules fulfill the following requirements.

a) **Coupling** : In computer science, coupling is considered to be the degree to which each program module relies on other modules, and is also the term used to describe connecting two or more systems. Coupling is broken down into loose coupling, tight coupling, and decoupled. Coupling is also used to describe software as well as systems. Also called dependency

## Types of coupling:

**Content coupling (high)**

Content coupling is when one module modifies or relies on the internal workings of another module (e.g. accessing local data of another module).
Therefore changing the way the second module produces data (location, type, timing) will lead to changing the dependent module.

**Common coupling**

Common coupling is when two modules share the same global data (e.g. a global variable).
Changing the shared resource implies changing all the modules using it.

**External coupling**

External coupling occurs when two modules share an externally imposed data format, communication protocol, or device interface.

**Control coupling**

Control coupling is one module controlling the logic of another, by passing it information on what to do (e.g. passing a what-to-do flag).

**Stamp coupling (Data-structured coupling)**

Stamp coupling is when modules share a composite data structure and use only a part of it, possibly a different part (e.g. passing a whole record to a function which only needs one field of it).
This may lead to changing the way a module reads a record because a field, which the module doesn't need, has been modified.

**Data coupling**

Data coupling is when modules share data through, for example, parameters. Each datum is an elementary piece, and these are the only data which are shared (e.g. passing an integer to a function which computes a square root).

**Message coupling (low)**

This is the loosest type of coupling. Modules are not dependent on each other; instead they use a public interface to exchange parameter-less messages

**No coupling**

Modules do not communicate at all with one another.

**2) <u>Cohesion:</u>** Cohesion is a measure of degree of how strongly a class focuses on its responsibilities. It is of the following two types:

- **High cohesion:** This means that a class is designed to carry on a specific and precise task. Using high cohesion, methods are easier to understand, as they perform a single task.
- **Low cohesion:** This means that a class is designed to carry on various tasks. Using low cohesion, methods are difficult to understand and maintain.

3) **<u>Span of control:</u>** It means number of modules subordinate to calling module. Limit of span of control may vary from 5 to 7 modules.

4) **<u>Size:</u>** The number of instructions contained in a module should be limited ats that module size is small

5) **<u>Shared Use:</u>** Functions should not be duplicated in separated modules but established in a single module that can be invoked by any other module when needed.

VEIS

# UNIT 3

## Types of programming Language

### Low Level Language

First-generation language is the lowest level computer language. Information is conveyed to the computer by the programmer as binary instructions. Binary instructions are the equivalent of the on/off signals used by computers to carry out operations. The language consists of zeros and ones. In the 1940s and 1950s, computers were programmed by scientists sitting before control panels equipped with toggle switches so that they could input instructions as strings of zeros and ones.

**Advantages**

- Fast and efficient
- Machine oriented
- No translation required

**Disadvantages**

- Not portable
- Not programmer friendly

### Assembly Language

Assembly or assembler language was the second generation of computer language. By the late 1950s, this language had become popular. Assembly language consists of letters of the alphabet. This makes programming much easier than trying to program a series of zeros and ones. As an added programming assist, assembly language makes use of mnemonics, or memory aids, which are easier for the human programmer to recall than are numerical codes

## Assembler

An assembler is a program that takes basic computer instructions and converts them into a pattern of bits that the computer's processor can use to perform its basic operations. Some people call these instructions assembler language and others use the term *assembly language In other words* An **assembler** is a computer program for translating assembly language — essentially, a mnemonic representation of machine language — into object code. A **cross assembler** (see cross compiler) produces code for one processor, but runs on another.
As well as translating assembly instruction mnemonics into opcodes, assemblers provide the ability to use symbolic names for memory locations (saving tedious calculations and manually updating addresses when a program is slightly modified), and macro facilities for performing textual substitution — typically used to encode common short sequences of instructions to run inline instead of in a subroutine.

## .High Level Language

The introduction of the compiler in 1952 spurred the development of third-generation computer languages. These languages enable a programmer to create program files using commands that are similar to spoken English. Third-level computer languages have become the major means of communication between the digital computer and its user.

By 1957, the International Business Machine Corporation (IBM) had created a language called FORTRAN (Formula Translator). This language was designed for scientific work involving complicated mathematical formulas. It became the first high-level programming language (or "source code") to be used by many computer users.

Within the next few years, refinements gave rise to ALGOL (Algorithmic Language) and COBOL (Common Business Oriented Language). COBOL is noteworthy because it improved the record keeping and data management ability of businesses, which stimulated business expansion.

**Advantages**

- Portable or *machine independent*
- Programmer-friendly

**Disadvantages**

- Not as efficient as low-level languages
- Need to be translated

**Examples:** C, C++, Java, FORTRAN, Visual Basic, and Delphi.

## Interpreter

An **interpreter** is a computer program that executes other programs. This is in contrast to a compiler which does not execute its input program (the source code) but translates it into executable machine code (also called object code) which is output to a file for later execution. It may be possible to execute the same source code either directly by an interpreter or by compiling it and then executing the machine code produced.
It takes longer to run a program under an interpreter than to run the compiled code but it can take less time to interpret it than the total required to compile and run it. This is especially important when prototyping and testing code when an edit-interpret-debug cycle can often be much shorter than an edit-compile-run-debug cycle.

Interpreting code is slower than running the compiled code because the interpreter must analyze each statement in the program each time it is executed and then perform the desired action whereas the compiled code just performs the action. This run-time analysis is known as "interpretive overhead". Access to variables is also slower in an interpreter because the mapping of identifiers to storage locations must be done repeatedly at run-time rather than at compile time.

## Compiler

A program that translates *source code* into *object code*. The compiler derives its name from the way it works, looking at the entire piece of source code and collecting and reorganizing the instructions. Thus, a compiler differs from an *interpreter*, which analyzes and executes each line of source code in succession, without looking at the entire program. The advantage of interpreters is that they can execute a program immediately. Compilers require some time before an executable program emerges. However, programs produced by compilers run much faster than the same programs executed by an interpreter.
Every high-level programming language (except strictly interpretive languages) comes with a compiler. In effect, the compiler is the language, because it defines which instructions are acceptable.

## Fourth Generation

Fourth-generation languages attempt to make communicating with computers as much like the processes of thinking and talking to other people as possible. The problem is that the computer still only understands zeros and ones, so a compiler and interpreter must still convert the source code into the machine code that the computer can understand. Fourth-generation languages typically consist of English-like words and phrases. When they are implemented on microcomputers, some of these languages include graphic devices such as icons and onscreen push buttons for use during programming and when running the resulting application.

Many fourth-generation languages use Structured Query Language (SQL) as the basis for operations. SQL was developed at IBM to develop information stored in relational databases. Examples of fourth-generation languages include PROLOG, an **artificial intelligence** language

## Introduction of C & Pascal

The common concepts which are used in languages are as follows:

***Size and Range of Data Types on 16 bit machine.***

| **TYPE** | **SIZE (Bits)** | **Range** |
|---|---|---|
| Char or Signed Char | 8 | -128 to 127 |
| Unsigned Char | 8 | 0 to 255 |
| Int or Signed int | 16 | -32768 to 32767 |
| Unsigned int | 16 | 0 to 65535 |
| Short int or Signed short int | 8 | -128 to 127 |
| Unsigned short int | 8 | 0 to 255 |
| Long int or signed long int | 32 | -2147483648 to 2147483647 |
| Unsigned long int | 32 | 0 to 4294967295 |
| Float | 32 | 3.4 e-38 to 3.4 e+38 |
| Double | 64 | 1.7e-308 to 1.7e+308 |
| Long Double | 80 | 3.4 e-4932 to 3.4 e+4932 |

## Variable

Variable is a name of memory location where we can store any data. It can store only single data (Latest data) at a time. In C, a variable must be declared before it can be used. Variables can be declared at the start of any block of code, but most are found at the start of each function.

A declaration begins with the type, followed by the name of one or more variables. For example,

Data Type Name_of_Variable_Name;

int a,b,c;

Constants

Character constants are usually just the character enclosed in single quotes; 'a', 'b', 'c'. Some characters can't be represented in this way, so we use a 2 character sequence.

| | |
|---|---|
| '\n' | newline |
| '\t' | tab |
| '\\ ' | backslash |
| '\'' | single quote |
| '\0' | null (used automatically to terminate character strings). |

## C Language Operator Precedence Chart

Operator precedence describes the order in which C reads expressions. For example, the expression a=4+b*2 contains two operations, an addition and a multiplication. Does the C compiler evaluate 4+b first, then multiply the result by 2, or does it evaluate b*2 first, then add 4 to the result? The operator precedence chart contains the answers. Operators higher in the chart have a higher precedence, meaning that the C compiler evaluates them first. Operators on the same line in the chart have the same precedence, and the "Associativity" column on the right gives their evaluation order

**Operator Precedence Chart**

| **Operator Type** | **Operator** | **Associativity** |
|---|---|---|
| Primary Expression Operators | () [] . -> *expr*++ *expr*-- | left-to-right |
| Unary Operators | * & + - ! ~ ++*expr* --*expr* (*typecast*) sizeof() | right-to-left |
| Binary Operators | * / % | left-to-right |

| | | |
|---|---|---|
| | + - | |
| | >> << | |
| | < > <= >= | |
| | == != | |
| | & | |
| | ^ | |
| | \| | |
| | && | |
| | \|\| | |
| Ternary Operator | ?: | right-to-left |
| Assignment Operators | = += -= *= /= %= >>= <<= &= ^= \|= | right-to-left |
| Comma | , | left-to-right |

VEIS

## Operators Introduction

An operator is a symbol which helps the user to command the computer to do a certain mathematical or logical manipulations. Operators are used in C language program to operate on data and variables. C has a rich set of operators which can be classified as

1. Arithmetic operators
2. Relational Operators
3. Logical Operators

4. Assignment Operators
5. Increments and Decrement Operators
6. Conditional Operators
7. Bitwise Operators
8. Special Operators

## 1. Arithmetic Operators

All the basic arithmetic operations can be carried out in C. All the operators have almost the same meaning as in other languages. Both unary and binary operations are available in C language. Unary operations operate on a singe operand, therefore the number 5 when operated by unary – will have the value –5.

**Arithmetic Operators**

| **Operator** | **Meaning** |
|---|---|
| + | Addition or Unary Plus |
| – | Subtraction or Unary Minus |
| * | Multiplication |
| / | Division |
| % | Modulus Operator |

Examples of arithmetic operators are

x + y
x - y
-x + y
a * b + c
-a * b

etc., here a, b, c, x, y are known as operands. The modulus operator is a special operator in C language which evaluates the remainder of the operands after division.
When an arithmetic operation is performed on two whole numbers or integers than such an operation is called as integer arithmetic. It always gives an integer as the result. Let x = 27 and y = 5 be 2 integer numbers. Then the integer operation leads to the following results.

x + y = 32
x – y = 22
x * y = 115
x % y = 2

x / y = 5

**2. Relational Operators**

Often it is required to compare the relationship between operands and bring out a decision and program accordingly. This is when the relational operator come into picture. C supports the following relational operators.

| Operator | Meaning |
|---|---|
| < | is less than |
| <= | is less than or equal to |
| > | is greater than |
| >= | is greater than or equal to |
| == | is equal to |

It is required to compare the marks of 2 students, salary of 2 persons; we can compare those using relational operators.

A simple relational expression contains only one relational operator and takes the following form.

exp1 relational operator exp2

Where exp1 and exp2 are expressions, which may be simple constants, variables or combination of them. Given below is a list of examples of relational expressions and evaluated values.

6.5 <= 25 TRUE

-65 > 0 FALSE

10 < 7 + 5 TRUE

**3. Logical Operators**

C has the following logical operators, they compare or evaluate logical and relational expressions.

| Operator | Meaning |
|---|---|
| && | Logical AND |

VEIS

| | |
|---|---|
| \|\| | Logical OR |
| ! | Logical NOT |

**Logical AND (&&)**

This operator is used to evaluate 2 conditions or expressions with relational operators simultaneously. If both the expressions to the left and to the right of the logical operator is true then the whole compound expression is true.

**Example**

a > b && x = = 10

The expression to the left is a > b and that on the right is x == 10 the whole expression is true only if both expressions are true i.e., if a is greater than b and x is equal to 10.

**Logical OR (||)**

The logical OR is used to combine 2 expressions or the condition evaluates to true if any one of the 2 expressions is true.

**Example**

a < m || a < n

The expression evaluates to true if any one of them is true or if both of them are true. It evaluates to true if a is less than either m or n and when a is less than both m and n.

**Logical NOT (!)**

The logical not operator takes single expression and evaluates to true if the expression is false and evaluates to false if the expression is true. In other words it just reverses the value of the expression.

**For example**

! (x >= y) the NOT expression evaluates to true only if the value of x is neither greater than or equal to y

**4. Assignment Operators**

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

VEIS

**Example**

x = a + b

**5. Increment and Decrement Operators**

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

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

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

Consider the following

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

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

Suppose if we rewrite the above statement as

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

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

**6. Conditional or Ternary Operator**

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

exp1 ? exp2 : exp3

The ternary operator works as follows

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

**For example**

```
a = 10;
b = 15;

x = (a > b) ? a : b
```

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

## CONTROL FLOW STATEMENT

### IF- Statement:

It is the basic form where the if statement evaluate a test condition and direct program execution depending on the result of that evaluation.

Syntax:

```
If (Expression)
{
Statement 1;
Statement 2;
}
```

### IF-ELSE Statement:

An if statement may also optionally contain a second statement, the ``else clause," which is to be executed if the condition is not met. Here is an example:

```
if(n > 0)
        average = sum / n;
else    {
        printf("can't compute average\n");
        average = 0;
        }
```

VEIS

## Switch Case

This is another form of the multi way decision. It is well structured, but can only be used in certain cases where;

- Only one variable is tested, all branches must depend on the value of that variable. The variable must be an integral type. (int, long, short or char).
- Each possible value of the variable can control a single branch. A final, catch all, default branch may optionally be used to trap all unspecified cases.

Hopefully an example will clarify things. This is a function which converts an integer into a vague description. It is useful where we are only concerned in measuring a quantity when it is quite small.

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

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

## Loops

Looping is a way by which we can execute any some set of statements more than one times continuously .In c there are mainly three types of loops are use :

- while Loop
- do while Loop
- For Loop

### While *Loop*

Loops generally consist of two parts: one or more *control expressions* which (not surprisingly) control the execution of the loop, and the *body,* which is the statement or set of statements which is executed over and over.

The general syntax of a while loop is

```
                Initialization

          while( expression )
{
                    Statement1
                    Statement2
                    Statement3

}
```

The most basic *loop* in C is the while loop. A while loop has one control expression, and executes as long as that expression is true. This example repeatedly doubles the number 2 (2, 4, 8, 16, ...) and prints the resulting numbers as long as they are less than 1000:

```
          int x = 2;

          while(x < 1000)
                    {
                    printf("%d\n", x);
                    x = x * 2;
                    }
```

(Once again, we've used braces {} to enclose the group of statements which are to be executed together as the body of the loop.)

### For Loop

Our second loop, which we've seen at least one example of already, is the for loop. The general syntax of a while loop is

VEIS

```
        for( Initialization; expression; Increments/decrements )
{
                Statement1
                Statement2
                Statement3

}
```

The first one we saw was:

```
        for (i = 0; i < 10; i = i + 1)
                printf ("i is %d\n", i);
```

(Here we see that the for loop has three control expressions. As always, the *statement* can be a brace-enclosed block.)

**Do While Loop**

This is very similar to the while loop except that the test occurs at the end of the loop body. This guarantees that the loop is executed at least once before continuing. Such a setup is frequently used where data is to be read. The test then verifies the data, and loops back to read again if it was unacceptable.

```
do
{       printf("Enter 1 for yes, 0 for no :");
        scanf("%d", &input_value);
} while (input_value != 1 && input_value != 0)
```

**Defining a Structure**

A structure type is usually defined near to the start of a file using a typedef statement. typedef defines and names a new type, allowing its use throughout the program. typedefs usually occur just after the #define and #include statements in a file.

Here is an example structure definition.

```
  typedef struct {
        char name[64];
        char course[128];
        int age;
        int year;
  } student;
```

## Pointers

A pointer is a variable that points to or references a memory location in which data is stored. In the computer, each memory cell has an address that can be used to access that location so a pointer variable points to a memory location we can access and change the contents of this memory location via the pointer.

**Pointer declaration:**

A pointer is a variable that contains the memory location of another variable in which data is stored. Using pointer, you start by specifying the type of data stored in the location. The asterisk helps to tell the compiler that you are creating a pointer variable. Finally you have to give the name of the variable. The syntax is as shown below.

```
type * variable name
```

**The following example illustrate the declaration of pointer variable**

```
Int                                                          *ptr;
float *string;
```

**Address operator:**

Once we declare a pointer variable then we must point it to something we can do this by assigning to the pointer the address of the variable you want to point as in the following example:

```
ptr=&num;
```

## Structured programming

a structured program consists of a hierarchical collection of individual modules, which appear more abstract at the top levels and more detailed at the lower levels. This fits with our overall strategy of hierarchical decomposition, which we've followed from structured system specification through structured design and now down to structured programming. The process of building a program in this fashion is called "top-down programming," or step-wise refinement.

Another point I stressed during our discussion of program design is that each of these modules (or processes, in design terms) communicates with others through well-defined data interfaces. These data interfaces typically are subroutine parameter lists or function argument lists. Each part of the program appears to other parts as a black box that simply performs its assigned function in some unknown way.

VEIS

# CHAPTER 4
# Programming Tools

To make programming easy so that programmers concentrate on the programs to be solved rather than writing the program itself a number of automated tools have been developed so far and many are still being added in the list. The efficiency of a programmer increases manifold by application of appropriate development tools. In some cases the programmers do not have any choice than to employ these tools. In some cases the programmers do not have any choice than to employ these tools.
In general, a computer-based problem solving involves the following steps:

- The problem is identified and studied.
- Having understood the problem, it is stated clearly and unambiguously.
- The solution is conceived as an idea.
- The idea is put to test for its correctness and efficiency usually in black and white on a piece of paper.
- A definition set of steps – algorithm-is designed that would go into solving the problem at hand.
- The algorithm is translated into an appropriate programming language, which remain still on paper.
- The program is entered and stored on a storage device in the computer
- The stored text-form program is translated into the computer's machine language either line-by line or all the lines at same time.
- The program is debugged in case it suffers from any errors .
- The operating system, when commanded, loads this program into the memory of the computer and passes the control to this program.
- The program executes and the problem is solved

Editor
To store a program in textual form on a disk one has to key-in the program into a file. A program that enables a programmer to key in and/or modify a program and subsequently store and/or retrieve the same is called as an editor.

## Interpreter

An **interpreter** is a computer program that executes other programs. This is in contrast to a compiler which does not execute its input program (the source code) but translates it into executable machine code (also called object code) which is output to a file for later execution. It may be possible to execute the same source code either directly by an interpreter or by compiling it and then executing the machine code produced.
It takes longer to run a program under an interpreter than to run the compiled code but it can take less time to interpret it than the total required to compile and run it. This is especially important when prototyping and testing code when an edit-interpret-debug cycle can often be much shorter than an edit-compile-run-debug cycle.

Interpreting code is slower than running the compiled code because the interpreter must analyse each statement in the program each time it is executed and then perform the desired action whereas the compiled code just performs the action. This run-time analysis is known as "interpretive overhead". Access to variables is also slower in an interpreter because the mapping of identifiers to storage locations must be done repeatedly at run-time rather than at compile time.

**Interpreter characteristics:**

- relatively little time is spent analyzing and processing the program
- the resulting code is some sort of intermediate code
- the resulting code is interpreted by another program
- program execution is relatively slow

## COMPILER

A program that translates *source code* into *object code*. The compiler derives its name from the way it works, looking at the entire piece of source code and collecting and reorganizing the instructions. Thus, a compiler differs from an *interpreter*, which analyzes and executes each line of source code in succession, without looking at the entire program. The advantage of interpreters is that they can execute a program immediately. Compilers require some time before an executable program emerges. However, programs produced by compilers run much faster than the same programs executed by an interpreter.
Every high-level programming language (except strictly interpretive languages) comes with a compiler. In effect, the compiler is the language, because it defines which instructions are acceptable.

A compiler is likely to perform many or all of the following operations:
Lexical processing
Preprocessor
Parsing
Semantic analysis
Code optimizations
Code generation

## Compiler characteristics:

- spends a lot of time analyzing and processing the program
- the resulting executable is some form of machine- specific binary code
- the computer hardware interprets (executes) the resulting code
- program execution is fast

## Debugger

VEIS

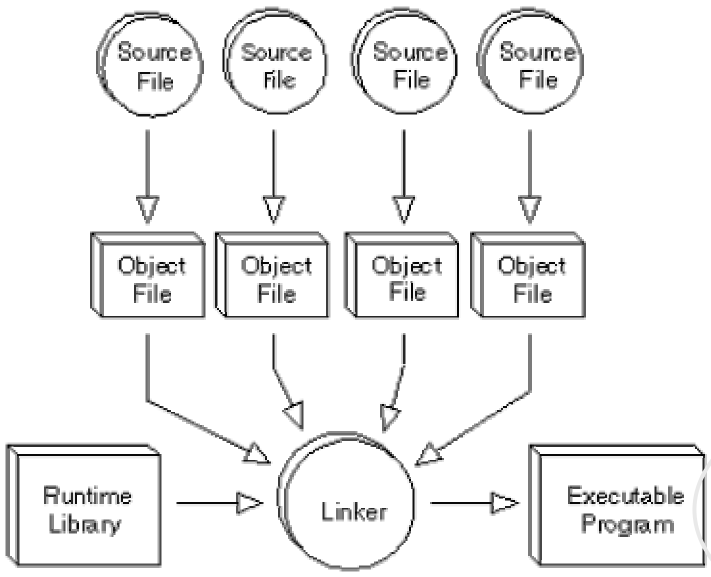
A debugger is a computer program that is used to test and debug other programs. The code to be examined might alternatively be running on an *instruction set simulator* (ISS), a technique that allows great power in its ability to halt when specific conditions are encountered but which will typically be much slower than executing the code directly on the appropriate processor In detail,
In computers, debugging is the process of locating and fixing or bypassing bugs (errors) in computer program code or the engineering of a hardware device. To *debug* a program or hardware device is to start with a problem, isolate the source of the problem, and then fix it. A user of a program that does not know how to fix the problem may learn enough about the problem to be able to avoid it until it is permanently fixed. When someone says they've debugged a program or "worked the bugs out" of a program, they imply that they fixed it so that the bugs no longer exist.

Debugging is a necessary process in almost any new software or hardware development process, whether a commercial product or an enterprise or personal application program. For complex products, debugging is done as the result of the unit test for the smallest unit of a system, again at component test when parts are brought together, again at system test when the product is used with other existing products, and again during customer beta test, when users try the product out in a real world situation. Because most computer programs and many programmed hardware devices contain thousands of lines of code, almost any new product is likely to contain a few bugs. Invariably, the bugs in the functions that get most use are found and fixed first

## Integrated development environment

An integrated development environment (IDE) is a programming environment that has been packaged as an application program, typically consisting of a code editor, a compiler, a debugger, and a graphical user interface (GUI) builder. The IDE may be a standalone application or may be included as part of one or more existing and compatible applications. The BASIC programming language, for example, can be used within Microsoft Office applications, which makes it possible to write a WordBasic program within the Microsoft Word application. IDEs provide a user-friendly framework for many modern programming languages, such as Visual Basic, Java, and PowerBuilder.

IDEs for developing HTML applications are among the most commonly used. For example, many people designing Web sites today use an IDE (such as HomeSite, DreamWeaver, or FrontPage) for Web site development that automates many of the tasks involved.



Also called *link editor* and *binder*, a linker is a program that combines object modules to form an executable program. Many programming languages allow you to write different pieces of code, called *modules*, separately. This simplifies the programming task because you can break a large program into small, more manageable pieces. Eventually, though, you need to put all the modules together. This is the job of the linker.

In addition to combining modules, a linker also replaces symbolic addresses with real addresses. Therefore, you may need to link a program even if it contains only one module.

VEIS

## Loader

In a computer operating system , a loader is a component that locates a given program (which can be an application or, in some cases, part of the operating system itself) in offline storage (such as a hard disk ), loads it into main storage (in a personal computer, it's called random access memory ), and gives that program control of the computer (allows it to execute its instruction s).

A program that is loaded may itself contain components that are not initially loaded into main storage, but can be loaded if and when their logic is needed. In a multitasking operating system, a program that is sometimes called a *dispatcher* juggles the computer processor's time among different tasks and calls the loader when a program associated with a task is not already in main storage. (By program here, we mean a binary file that is the result of a programming language compilation, linkage editing, or some other program preparation process.)

**The Visual C++ Development Environment**

Before you begin your quick tour around the Visual C++ development environment, you should start Visual C++ on your computer so that you can see firsthand how each of the areas are arranged and how you can change and alter that arrangement yourself.

After Developer Studio (the Microsoft Visual development environment) starts, you see a window that looks like Figure 1.1. Each of the areas has a specific purpose in the Developer Studio environment. You can rearrange these areas to customize the Developer Studio environment so that it suits your particular development needs.

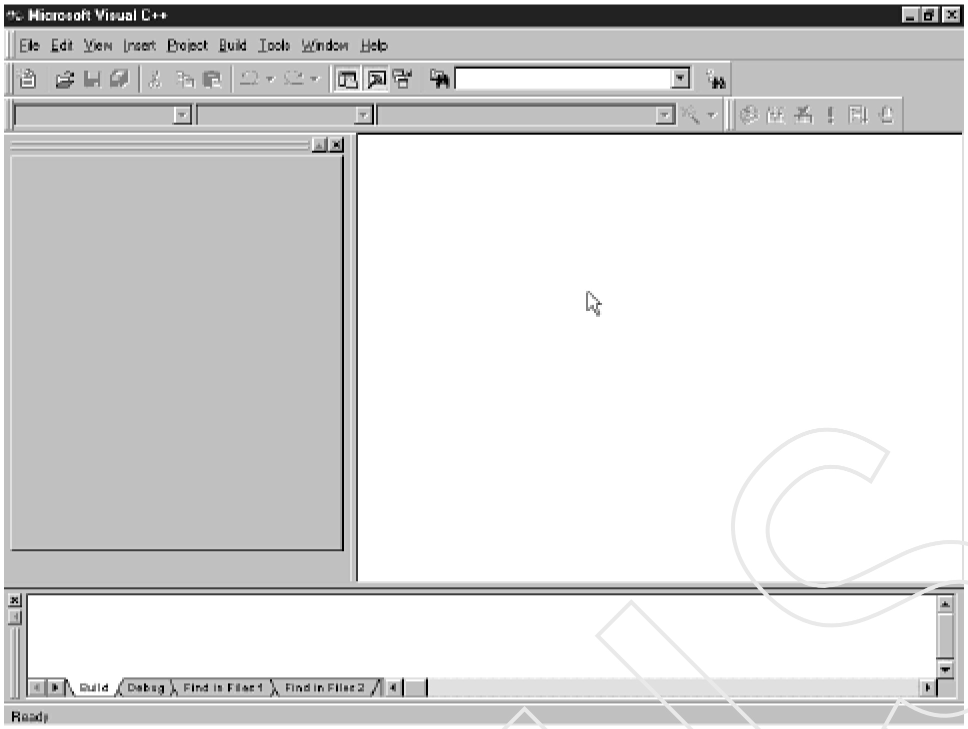
The Workspace
When you start Visual C++ for the first time, an area on the left side of Developer Studio looks like it is taking up a lot of real estate and providing little to show for it. This area is known as the workspace, and it is your key to navigating the various pieces and parts of your development projects. The workspace allows you to view the parts of your application in three different ways:

Class View allows you to navigate and manipulate your source code on a C++ class level.

Resource View allows you to find and edit each of the various resources in your application, including dialog window designs, icons, and menus.

File View allows you to view and navigate all the files that make up your appli-cation.

FIGURE . The Visual C++ opening screen.



## The Output Pane

The Output pane might not be visible when you start Visual C++ for the first time. After you compile your first application, it appears at the bottom of the Developer Studio environment and remains open until you choose to close it. The Output pane is where Developer Studio provides any information that it needs to give you; where you see all the compiler progress statements, warnings, and error messages; and where the Visual C++ debugger displays all the variables with their current values as you step through your code. After you close the Output pane, it reopens itself when Visual C++ has any message that it needs to display for you.

### The Editor Area

The area on the right side of the Developer Studio environment is the editor area. This is the area where you perform all your editing when using Visual C++, where the code editor windows display when you edit C++ source code, and where the window painter displays when you design a dialog box. The editor area is even where the icon painter displays when you design the icons for use in your applications. The editor area is basically the entire Developer Studio area that is not otherwise occupied by panes, menus, or toolbars.

### Menu Bars

The first time you run Visual C++, three toolbars display just below the menu bar. Many other toolbars are available in Visual C++, and you can customize and create your own toolbars to accommodate how you best work. The three toolbars that are initially open are the following:

The Standard toolbar contains most of the standard tools for opening and saving files, cutting, copying, pasting, and a variety of other commands that you are likely to find useful.

The WizardBar toolbar enables you to perform a number of Class Wizard actions without opening the Class Wizard.

The Build minibar provides you with the build and run commands that you are most likely to use as you develop and test your applications. The full Build toolbar also lets you switch between multiple build configurations (such as between the Debug and Release build configurations).
Rearranging the Developer Studio Environment
The Developer Studio provides two easy ways to rearrange your development environment. The first is by right-clicking your mouse over the toolbar area. This action opens the pop-up menu shown in Figure 1.2, allowing you to turn on and off various toolbars and panes.

**USING THE APPLICATION WIZARD TO CREATE THE APPLICATION SHELL**

The AppWizard asks you a series of questions about what type of application you are building and what features and functionality you need. It uses this information to create a shell of an application that you can immediately compile and run. This shell provides you with the basic infrastructure that you need to build your application around. You will see how this works as you follow these steps:

**1.** In Step 1 of the AppWizard, specify that you want to create a Dialog-based application. Click Next at the bottom of the wizard.
**2.** In Step 2 of the AppWizard, the wizard asks you about a number of features that you can include in your application. You can uncheck the option for including support for ActiveX controls if you will not be using any ActiveX controls in your application. Because you won't be using any ActiveX controls in today's application, go ahead and uncheck this box.
**3.** In the field near the bottom of the wizard, delete the project name (Hello) and type in the title that you want to appear in the title bar of the main application window, such as **My First Visual C++ Application**. Click Next at the bottom of the wizard.
**4.** In Step 3 of the AppWizard, leave the defaults for including source file comments and using the MFC library as a DLL. Click Next at the bottom of the wizard to proceed to the final AppWizard step.
**5.** The final step of the AppWizard shows you the C++ classes that the AppWizard will create for your application. Click Finish to let AppWizard generate your application shell.
**6.** Before AppWizard creates your application shell, it presents you with a list of what it is going to put into the application shell, as shown in Figure 1.7, based on the options you selected when going through the AppWizard. Click OK and AppWizard generates your application.

Chapter 5

# Development of efficient programs

## What is Debugging?

Debugging is the process of locating and fixing errors (known as bugs), in a computer program, or hardware device. To debug a program or hardware device, you start with a known problem, isolate the source of the problem, and then fix it. When someone says they have debugged a program, or "removed the bugs" in a program, they imply that they have fixed the program, so that the bugs no longer exist in it. Debugging is a necessary process in almost any new software, or hardware development process, whether a commercial product, an enterprise, or personal application program. For complex products, debugging is done periodically throughout the development, and again during the customer beta test stages. Because most computer programs and many programmed hardware devices contain thousands of lines of code, almost any new product is likely to contain a few bugs. Invariably, the bugs in the functions that get the most use, are found and fixed first. An early version of a program that has lots of bugs is referred to as "buggy." Debugging tools help identify coding errors at various stages of development. Some programming language packages include a facility for checking the code for errors as it is being written.

## Bug management

It is common practice for software to be released with **known bugs** that are considered non-critical, that is, that do not affect most users' main experience with the product. While software products may, by definition, contain any number of **unknown bugs**, measurements during testing can provide an estimate of the number of likely bugs remaining; this becomes more reliable the longer a product is tested and developed ("if we had 200 bugs last week, we should have 100 this week"). Most big software projects maintain two lists of "known bugs"— those known to the software team, and those to be told to users. This is not dissimulation, but users are not concerned with the internal workings of the product. The second list informs users about bugs that are not fixed in the current release, or not fixed at all, and a workaround may be offered.

There are various reasons for not fixing bugs:

- The developers often don't have time or it is not economical to fix all non-severe bugs.
- The bug could be fixed in a new version or patch that is not yet released.
- The changes to the code required to fix the bug would be large, and would bring with them the chance of introducing other bugs into the system.
- Users may be relying on the undocumented, buggy behavior; it may introduce a breaking change.
- It's "not a bug". A misunderstanding has arisen between expected and provided behavior

Given the above, it is often considered impossible to write completely bug-free software of any real complexity. So bugs are categorized by severity, and low-severity non-critical bugs are tolerated, as they do not affect the proper operation of the system for most users. NASA's SATC managed to reduce the number of errors to fewer than 0.1 per 1000 lines of code (SLOC) but this was not felt to be feasible for any real world projects.

The severity of a bug is not the same as its importance for fixing, and the two should be measured and managed separately. On a Microsoft Windows system a blue screen of death is rather severe, but if it only occurs in extreme circumstances, especially if they are well diagnosed and avoidable, it may be less important to fix than an icon not representing its function well, which though purely aesthetic may confuse thousands of users every single day. This balance, of course, depends on many factors; expert users have different expectations from novices, a niche market is different from a general consumer market, and so on.

A school of thought popularized by Eric S. Raymond as Linus's Law says that popular open-source software has more chance of having few or no bugs than other software, because "given enough eyeballs, all bugs are shallow".[12] This assertion has been disputed, however: computer security specialist Elias Levy wrote that "it is easy to hide vulnerabilities in complex, little understood and undocumented source code," because, "even if people are reviewing the code, that doesn't mean they're qualified to do so."[13]

Like any other part of engineering management, bug management must be conducted carefully and intelligently because "what gets measured gets done"[14] and managing purely by bug counts can have unintended consequences. If, for example, developers are rewarded by the number of bugs they fix, they will naturally fix the easiest bugs first— leaving the hardest, and probably most risky or critical, to the last possible moment ("I only have one bug on my list but it says "Make sun rise in West"). If the management ethos is to reward the number of bugs fixed, then some developers may quickly write sloppy code knowing they can fix the bugs later and be rewarded for it, whereas careful, perhaps "slower" developers do not get rewarded for the bugs that were never there.

**Software Testing** is an empirical investigation conducted to provide stakeholders with information about the quality of the product or service under test[1], with respect to the context in which it is intended to operate. Software Testing also provides an objective, independent view of the software to allow the business to appreciate and understand the risks at implementation of the software. Test techniques include, but are not limited to, the process of executing a program or application with the intent of finding software bugs. It can also be stated as the process of validating and verifying that a software program/application/product meets the business and technical requirements that guided its design and development, so that it works as expected and can be implemented with the same characteristics.

Software Testing, depending on the testing method employed, can be implemented at any time in the development process, however the most test effort is employed after the requirements have been defined and coding process has been completed.

**White box testing** (a.k.a. clear box testing, glass box testing, transparent box testing, translucent box testing or structural testing) uses an internal perspective of the system to design test cases based on internal structure. It requires programming skills to identify all paths through the software. The tester chooses test case inputs to exercise paths through the code and determines the appropriate outputs. In electrical hardware testing, every node in a circuit may be probed and measured; an example is in-circuit testing (ICT).

Since the tests are based on the actual implementation, if the implementation changes, the tests probably will need to change, too. For example ICT needs updates if component values change, and needs modified/new fixture if the circuit changes. This adds financial resistance to the change process, thus buggy products may stay buggy. Automated optical inspection (AOI) offers similar component level correctness checking without the cost of ICT fixtures, however changes still require test updates.

While white box testing is applicable at the unit, integration and system levels of the software testing process, it is typically applied to the unit. While it normally tests paths within a unit, it can also test paths between units during integration, and between subsystems during a system level test. Though this method of test design can uncover an overwhelming number of test cases, it might not detect unimplemented parts of the specification or missing requirements, but one can be sure that all paths through the test object are executed.

Typical white box test design techniques include:

- Control flow testing
- Data flow testing
- Branch Testing

**Black box testing** takes an external perspective of the test object to derive test cases. These tests can be functional or non-functional, though usually functional. The test designer selects valid and invalid inputs and determines the correct output. There is no knowledge of the test object's internal structure.

This method of test design is applicable to all levels of software testing: unit, integration, functional testing, system and acceptance. The higher the level, and hence the bigger and more complex the box, the more one is forced to use black box testing to simplify. While this method can uncover unimplemented parts of the specification, one cannot be sure that all existent paths are tested.

- *Alpha testing* is simulated or actual operational testing by potential users/customers or an independent test team at the developers' site. Alpha testing is often employed for off-the-shelf software as a form of internal acceptance testing, before the software goes to beta testing
- *Beta testing* comes after alpha testing. Versions of the software, known as beta versions, are released to a limited audience outside of the programming team. The software is released to groups of people so that further testing can ensure the product has few faults or bugs. Sometimes, beta versions are made available to the open public to increase the feedback field to a maximal number of future users

- *Gamma* Testing is a little-known informal phrase that refers derisively to the release of “buggy” products. It is not a term of art among testers, but rather an example of referential humar. Cynics have refers to all software releases as “gamma testing” since defects are found in almost all commercial, commodity and publicly available software eventually.

Finally, acceptance testing can be conducted by the end-user, customer, or client to validate whether or not to accept the product. Acceptance testing may be performed as part of the hand-off process between any two phases of development

**System testing** of software or hardware is testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. System testing falls within the scope of black box testing, and as such, should require no knowledge of the inner design of the code or logic. As a rule, system testing takes, as its input, all of the "integrated" software components that have successfully passed integration testing and also the software system itself integrated with any applicable hardware system(s). The purpose of integration testing is to detect any inconsistencies between the software units that are integrated together (called *assemblages*) or between any of the *assemblages* and the hardware. System testing is a more limiting type of testing; it seeks to detect defects both within the "inter-assemblages" and also within the system as a whole.

**Test-Cycle**

Although testing varies between organizations, there is a cycle to testing. Here is an example of typical test cycle

1) Requirements Analysis: Testing should begin in the requirements phase of the sdlc
2) Design Analysis: During the design phase, esters work with developers in determining what aspects of a design are testable and under what parameter those testers work.
3) Test Planning: Test strategy, Test Plan, Test Bed creation
4) Test Development: Test Procedures, Test Scenarios, Test cases, Test Scripts to use in testing software
5) Test Execution: Testers execute the software based on the plans and tests and report any errors found to the development team.
6) Test Reporting: Once testing is completed, testers generate metrics and make final reports on their efforts and whether or not the software tested is ready for release.
7) Retesting the defects.