OVERVIEW OF PROGRAMMING AND PROBLEM SOLVING.

1.1 Overview of Programming

1.1.1 Introduction

A computer is a programmable electronic device that can store, retrieve, and process data.

What a brief definition for something that has, in just a few decades, changed the way of life in industrialized societies! Computers touch all areas of our lives: paying bills, driving cars, using the telephone, shopping. In fact, it might be easier to list those areas of our lives in which we do not use computers. You are probably most familiar with computers through the use of games, word processors, Web browsers, and other programs.

In this section we describe some general principles that you can use to design and write programs. These principles are not particular to Visual Basic. They apply no matter what programming language you are using.

1.1.2 What is Programming?

> *The Analytical Engine has no pretensions whatever to originate anything. It can do whatever we know how to order it to perform. It can follow analysis; but it has no power of anticipating any analytical relations or truths. Its province is to assist us in making available what we are already acquainted with. ADA AUGUSTA, COUNTESS OF LOVELACE (1815–1852)*

Much of human behaviour and thought is characterized by logical sequences of actions applied to objects. Since infancy, you have been learning how to act, how to do things; and you have learned to expect certain behaviour from other people.

A lot of what you do every day you do automatically. Much of what you do unconsciously you once had to learn. Watch how a baby concentrates on putting one foot before the other while learning to walk. Then watch a group of three-year-olds playing tag.

On a broader scale, mathematics never could have been developed without logical sequences of steps for manipulating symbols to solve problems and prove theorems. Mass production would never have worked without operations taking place on component parts in a certain order. Our whole civilization is based on the order of things and actions.

We create order, both consciously and unconsciously, through a process called **programming. Programming is planning or scheduling the performance of a task or an event.**

Notice that the key word in the definition of computer is data. Computers manipulate data. When you write a program (a plan) for a computer, you **specify the properties of the data and the operations that can be applied to it.** Those operations are then combined as necessary to solve a problem.

Data is information in a form the computer can use, for example, numbers and letters. **Information** is any knowledge that can be communicated, including abstract ideas and concepts such as “the Earth is round.”

Data comes in many different forms: letters, words, integer numbers, real numbers, dates, times, coordinates on a map, and so on. Virtually any kind of information can be represented as data, or as a combination of data and operations on it. Each kind of data in the computer is said to have a specific data type. For example, if we say that two data items are of type Integer, we know now they are represented in memory and that we can apply arithmetic operations to them. A **data type** is simply specification of how information is represented in the computer as data and the set of operations that can be applied to it

Computer programming is the process of specifying the data types and the operations for a computer to apply to data in order to solve a problem. **Computer program** is simply a data type specifications and instructions for carrying out operations that are used by a computer to solve a problem

Just as a concert program lists the pieces to be performed and the order in which the players perform them, a computer program lists the types of data that are to be used and the sequence of steps the computer performs on them.

The computer allows us to do tasks more efficiently, quickly, and accurately than we could by hand—if we could do them by hand at all. In order for this powerful machine to be a useful tool, it must be programmed. That is, we must specify what we want done and how. We do this through programming.

1.2. Program Design

A computer is not intelligent. It **cannot** analyze a problem and come up with a solution. A **human** (the programmer) **must** analyze the problem, develop the instructions for solving the problem, and then have the computer carry out the instructions. What’s the advantage of using a computer if it cannot solve problems? Once we have written a solution for the computer, the computer can repeat the solution very quickly and consistently, again and again. The computer frees people from repetitive and boring tasks.

Designing a program is often a difficult task. There is no complete set of rules, no algorithm to tell you how to write programs. **Program design is a creative process**. Still, there is the outline of a plan to follow. The outline is given below.

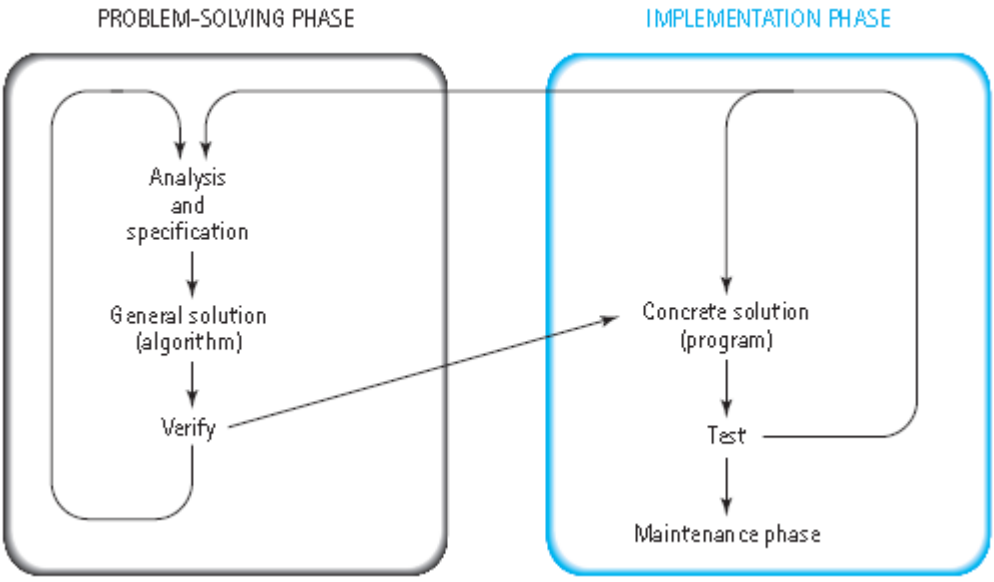

Fig 1.0: Program Design Process

As indicated there, the entire program- design process can be divided into two phases, the **problem-solving phase** and the **implementation phase**. The result of the problem-solving phase is an algorithm, expressed in English, for solving the problem. To produce a program in a programming language such as Visual Basic, the algorithm is translated into the programming language. Producing the final program from the algorithm is called the implementation phase.

1.2.1 Problem-Solving Phase

- **Analysis and Specification**.

  It is important that you understand (define) the problem and what the solution must do. The first step in a programming task is to be certain that the task that you want your program to do is completely and precisely specified. Do not take this step lightly. If you do not know exactly what you want as the output of your program, you may be surprised at what your program produces. Be certain that you know what the input to the program will be and exactly what information is supposed to be in the output, as well as what form that information should be in. For example, if the program is a bank accounting program, you must know not only the interest rate, but also whether interest is to be compounded annually, monthly, daily, or whatever.

- **General Solution (Algorithm).**

  Specify the required data types and the logical sequences of steps that solve the problem. Many novice programmers do not understand the need to design an algorithm before writing a program in a programming language, such as Visual Basic, and so they try to short-circuit the process by omitting the problem-solving phase entirely, or by reducing it to just the problem definition part. This seems reasonable. Why not “go for the mark” and save time? The answer is that it does not save time!

Experience has shown that the two-phase process will produce a correctly working program faster. The two-phase process simplifies the algorithm design phase by isolating it from the detailed rules of a programming language such as Visual Basic. The result is that the algorithm design process becomes much less intricate and much less prone to error. For even a modest-size program, it can represent the difference between a half day of careful work and several frustrating days of looking for mistakes in a poorly understood program.

- Verify. Follow the steps exactly to see if the solution really does solve the problem.

An **Algorithm** is an instruction for solving a problem or sub- problem in a finite amount of time using a finite amount of data.

1.2.2 Implementation Phase

1. Concrete Solution (Program). Translate the algorithm (the general solution) into a programming language.

2. Test. Have the computer follow the instructions. Then manually check the results. If you find errors, analyze the program and the algorithm to determine the source of the errors, and then make corrections.

Once a program has been written, it enters a third phase: maintenance.

1.2.3 Maintenance Phase

1. Use. Use the program.

2. Maintain. Modify the program to meet changing requirements or to correct any errors that show up while using it.

The programmer begins the programming process by analyzing the problem, breaking it into manageable pieces, and developing a general solution for each piece called an **algorithm**. The solutions to the pieces are collected together to form a program that solves the original problem. Understanding and analyzing a problem take up much more time than Figure 1.0 implies. They are the heart of the programming process.

As indicated in fig 1.0, testing takes place in both phases. Before the program is written, the algorithm is tested, and if the algorithm is found to be deficient, then the algorithm is redesigned. That implementation testing is performed by mentally going through the algorithm and executing the steps yourself. On large algorithms this will require a pencil and paper. The Visual Basic program is tested by compiling it and running it on some sample input data. The compiler will give error messages for certain kinds of errors. To find other types of errors, you must somehow check to see whether the output is correct.

1.3 Algorithm

If our definitions of a computer program and an algorithm look similar, it is because a program is simply an algorithm that has been written for a computer. An **algorithm** is a written or verbal description of a logical sequence of actions applied to objects. We use algorithms every day. Recipes, instructions, and directions, method, procedure, and routine **are all examples** of algorithms that are not programs.

When learning your first programming language it is easy to get the impression that the hard part of solving a problem on a computer is translating your ideas into the specific language that will be fed into the computer. This definitely is not the case. The most difficult part of solving a problem on a computer is discovering the method of solution. After you come up with a method of solution, it is routine to translate your method into the required language, be it Visual Basic or some other programming language. It is therefore helpful to temporarily ignore the programming language and to concentrate instead on formulating the steps of the solution and writing them down in plain English, as if the instructions were to be given to a human being rather than a computer. A sequence of instructions expressed in this way is frequently referred to as an algorithm.

The instructions may be expressed in a programming language or a human language. Our algorithms will be expressed in English and in the programming language Visual Basic. A computer program is simply an algorithm expressed in a language that a computer can understand. Thus, the term algorithm is more general than the term program. However, when we say that a sequence of instructions is an algorithm, we usually mean that the instructions are expressed in English, since if they were expressed in a programming language we would use the more specific term program. An example may help to clarify the concept.

When you start your car, you follow a step-by-step procedure. The algorithm might look something like this:

1. Insert the key.
2. Make sure the transmission is in Park (or Neutral).
3. Turn the key to the start position.
4. If the engine starts within six seconds, release the key to the ignition position.
5. If the engine doesn’t start in six seconds, release the key and gas pedal, wait ten seconds, and repeat Steps 3 through 5, but not more than five times.
6. If the car doesn’t start, call the mechanic.

Without the phrase “but not more than five times” in Step 5, you could be trying to start the car forever. Why? Because if something is wrong with the car, repeating Steps 3 through 5 over and over will not start it. This kind of never-ending situation is called an infinite loop. If we leave the phrase “but not more than five times” out of Step 5, the procedure doesn’t fit our

definition of an algorithm. An algorithm must terminate in a finite amount of time for all possible conditions.

Suppose a programmer needs an algorithm to determine an employee's weekly wages. The algorithm reflects what would be done by hand:

1. Look up the employee's pay rate.
2. Determine the hours worked during the week.
3. If the number of hours worked is less than or equal to 40, multiply the hours by the pay rate to calculate regular wages.
4. If the number of hours worked is greater than 40, multiply 40 by the pay rate to calculate regular wages, and then multiply the difference between the hours worked and 40 by $1^{1}/_{2}$ times the pay rate to calculate overtime wages.
5. Add the regular wages to the overtime wages (if any) to determine total wages for the week.

The steps the computer follows are often the same steps you would use to do the calculations by hand. After developing a general solution, the programmer tests the algorithm, "walking through" each step manually with paper and pencil. If the algorithm doesn't work, the programmer repeats the problem-solving process, analyzing the problem again and coming up with another algorithm. Often the second algorithm is just a variation of the first. When the programmer is satisfied with the algorithm, he or she translates it into a programming language. We use the Visual Basic programming language in this course.

1.3 Programming Language

A **programming language** is a simplified form of English (with math symbols) that adheres to a strict set of grammatical rules. English is far too complicated and ambiguous for today's computers to follow. Programming languages, because they limit vocabulary and grammar, are much simpler. Programming language is a set of rules, symbols, and special words used to construct a computer program

Although a programming language is simple in form, it is not always easy to use. Try giving someone directions to the nearest airport using a limited vocabulary of no more than 25 words, and you begin to see the problem. **Programming forces you to write very simple, exact instructions**.

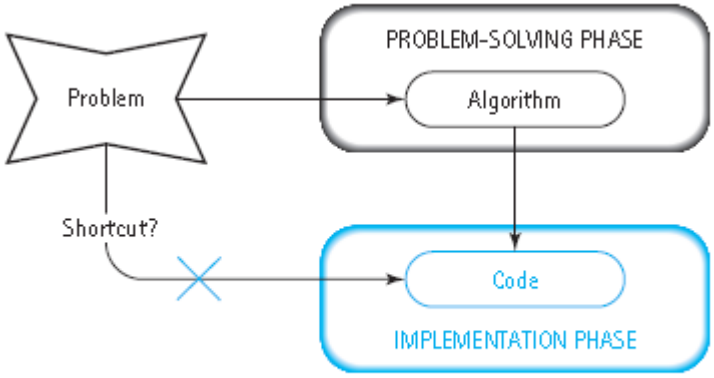

Figure 1.3 *Programming shortcut?*

Translating an algorithm into a programming language is called **coding** the algorithm. The products of the translation—the code for all the algorithms in the problem— are tested by collecting them into a program and running (executing) the program on the computer. If the program fails to produce the desired results, the programmer must debug it—that is, determine what is wrong and then modify the program, or even one or more of the algorithms, to fix it. **The combination of coding and testing the algorithms is called implementation**.

Code is a data type specifications and instructions for a computer that are written in a programming language Code is the product of translating an algorithm into a programming language. **The term code can refer to a complete program or to any portion of a program.**

There is no single way to implement an algorithm. For example, an algorithm can be translated into more than one programming language. Each translation produces a different implementation. Even when two people translate an algorithm into the same programming language, they are likely to come up with different implementations. Why? Because every programming language allows the programmer some flexibility in how an algorithm is translated. Given this flexibility, people adopt their own styles in writing programs, just as they do in writing short stories or essays. Once you have some programming experience, you can develop a style of your own.

Some people try to speed up the programming process by going directly from the problem definition to coding the program. A shortcut here is very tempting and at first seems to save a lot of time. However, for many reasons that will become obvious to you as this course progresses, this kind of shortcut actually takes more time and effort. Developing a general solution before you write a program helps you manage the problem, keep your thoughts straight, and avoid mistakes. If you don’t take the time at the beginning to think out and polish your algorithm, you spend a lot of extra time debugging and revising your program. So think first and code later.

Once a program has been put into use, it is often necessary to modify it. Modification may involve fixing an error that is discovered during the use of the program or changing the

program in response to changes in the user's requirements. Each time the program is modified, it is necessary to repeat the problem-solving and implementation phases for those aspects of the program that change. This phase of the programming process is known as **maintenance** and actually accounts for the majority of the effort expended on most programs. For example, a program that is implemented in a few months may need to be maintained over a period of many years. Thus it is a cost-effective investment of time to carefully develop the initial problem solution and program implementation. Together, the problem-solving, implementation, and maintenance phases constitute the program's life cycle.

In addition to solving the problem, implementing the algorithm, and maintaining the program, writing documentation is an important part of the programming process. Documentation includes written explanations of the problem being solved and the organization of the solution, comments embedded within the program itself, and user manuals that describe how to use the program. Many different people are likely to work on a program over a long period of time. Each of those people must be able to read and understand the code.

1.4 program Translators (How Is a Program Converted into a Form That a Computer Can Use)

In the computer, all data, whatever its form, is stored and used in binary codes, strings of 1s and 0s. Instructions and data are stored together in the computer's memory using these binary codes. If you were to look at the binary codes representing instructions and data in memory, you could not tell the difference between them; they are distinguished only by the manner in which the computer uses them. It is thus possible for the computer to process its own instructions as a form of data.

When computers were first developed, the only programming language available was the primitive instruction set built into each machine, the **machine language, or machine code**. A Machine language is made up of binary- coded instructions that are used directly by the computer.

Even though most computers perform the same kinds of operations, their designers choose different sets of binary codes for each instruction. So the machine code for one family of computers is not the same as for another.

When programmers used machine language for programming, they had to enter the binary codes for the various instructions, a tedious process that was prone to error. Moreover, their programs were difficult to read and modify. In time, **assembly languages** were developed to make the programmer's job easier. Assembly language is a low-level programming language in which a mnemonic is used to represent each of the machine language instructions for a particular computer.

Instructions in an assembly language are in an easy-to-remember form called a mnemonic. Typical instructions for addition and subtraction might look like this:

Assembly Language	Machine Language
ADD	100101
SUB	010011

Although assembly language is easier for humans to work with, the computer cannot directly execute the instructions. Because a computer can process its own instructions as a form of data, it is possible to write a program to translate assembly language into machine code. Such a program is called an assembler.

Assembly language is a step in the right direction, but it still forces programmers to think in terms of individual machine instructions. Eventually, computer scientists developed high-level programming languages. These languages are easier to use than assembly languages or machine code because they are closer to English and other natural languages (see Figure 1.4).

A program called a compiler translates algorithms written in certain high-level lan- guages (Visual Basic, C++, Java, Pascal, and Ada, for example) into machine language. If you write a program in a high-level language, you can run it on any computer that has the appropriate compiler. This is possible because most high-level languages are standardized, which means that an official description of the language exists.

The text of an algorithm written in a high-level language is called source code. To the compiler, source code is just input data. It translates the source code into a machine language form called object code

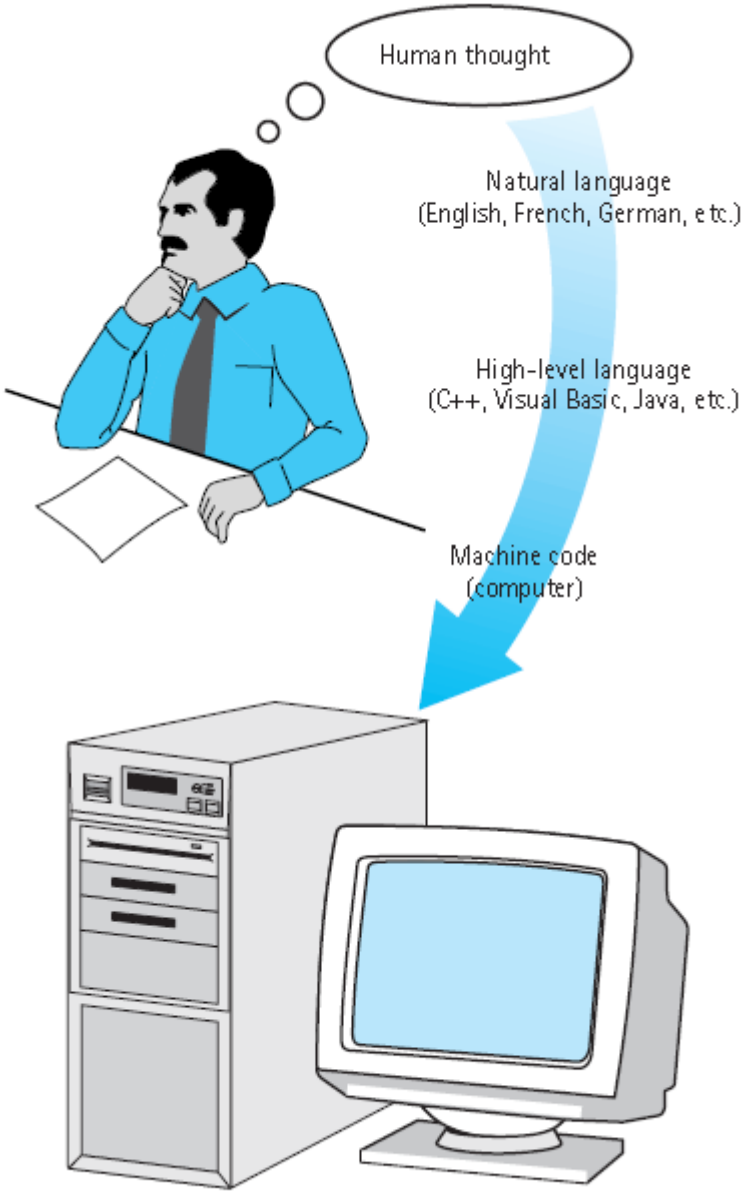

Figure 1.4 *Levels of abstraction*

A benefit of standardized high-level languages is that they allow you to write portable (or machine independent) code. As Figure 1.5 emphasizes, a single C++ program can be run on different machines, whereas a program written in assembly language or machine language is not portable from one computer to another. Because each computer family has its own machine language, a machine language program written for computer A may not run on computer B.

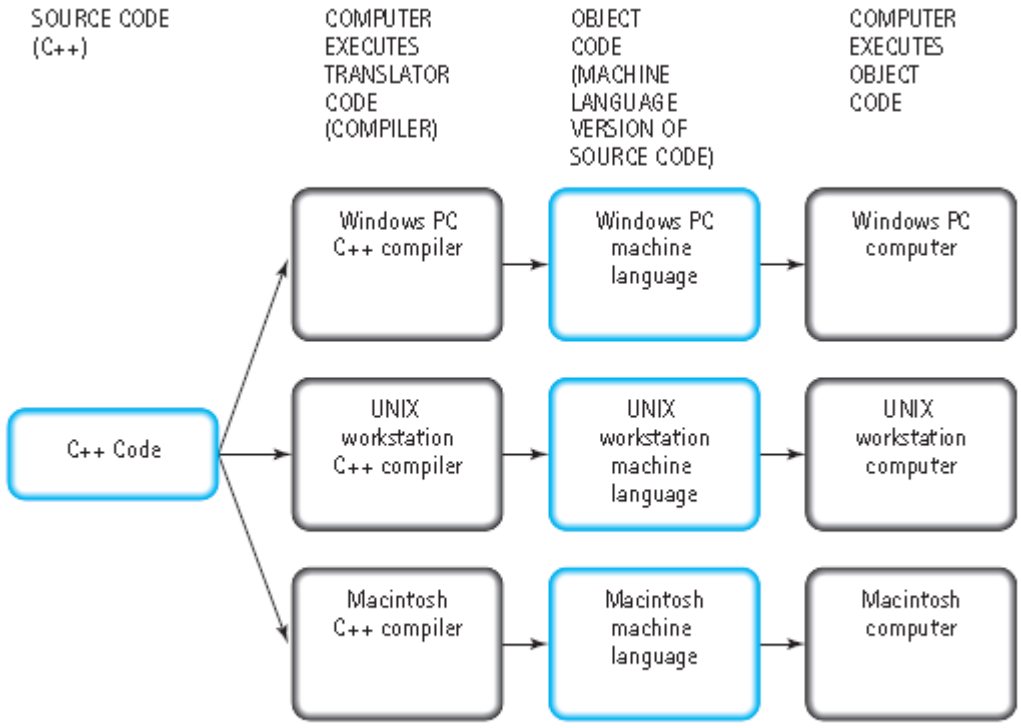

Figure 1.5 *High-level programming languages allow programs to be compiled on different systems.*

Visual Basic takes a somewhat different approach than we have described. Visual Basic programs are translated into a standard machine language called **Bytecode.**

However, there are no computers that actually use Bytecode as their machine language. In order for a computer to run Bytecode programs, it must have another program called the Common Language Run- time (CLR) that serves as a language interpreter for the program. Just as an interpreter of human languages listens to words spoken in one language and speaks a translation of them in a language that another person understands, the CLR reads the Bytecode machine language instructions and translates them into machine language operations that the particular computer executes. Interpretation is done as the program is running, one instruction at a time. It is not the same as compilation, which is a separate step that translates all of the instructions in a program prior to execution.