Topic: 2.4.1 Programming

Why Programming?

You may already have used software, perhaps for word processing or spreadsheets to solve problems. Perhaps now you are curious to learn how programmers write software. A *program* is a set of step-by-step instructions that directs the computer to do the tasks you want it to do and produce the results you want.

There are at least three good reasons for learning programming:

- Programming helps you understand computers. The computer is only a tool. If you learn how to write simple programs, you will gain more knowledge about how a computer works.
- Writing a few simple programs increases your confidence level. Many people find great personal satisfaction in creating a set of instructions that solve a problem.
- Learning programming lets you find out quickly whether you like programming and whether you have the analytical turn of mind programmers need. Even if you decide that programming is not for you, understanding the process certainly will increase your appreciation of what programmers and computers can do.

A set of rules that provides a way of telling a computer what operations to perform is called a programming language. There is not, however, just one programming language; there are many. In this chapter you will learn about controlling a computer through the process of programming. You may even discover that you might want to become a programmer.

An important point before we proceed: You will not be a programmer when you finish reading this chapter or even when you finish reading the final chapter. Programming proficiency takes practice and training beyond the scope of this book. However, you will become acquainted with how programmers develop solutions to a variety of problems.

What is it that Programmers Do?
In general, the programmer's job is to convert problem solutions into instructions for the computer. That is, the programmer prepares the instructions of a computer program and runs those instructions on the computer, tests the program to see if it is working properly, and makes corrections to the program. The programmer also writes a report on the program. These activities are all done for the purpose of helping a user fill a need, such as paying employees, billing customers, or admitting students to college.

The programming activities just described could be done, perhaps, as solo activities, but a programmer typically interacts with a variety of people.

For example, if a program is part of a system of several programs, the programmer coordinates with other programmers to make sure that the programs fit together well. If you were a programmer, you might also have coordination meetings with users, managers, systems analysts, and with peers who evaluate your work-just as you evaluate theirs.

Topic: 2.4.1 Programming

Let us turn to the programming process.

The Programming Process
Developing a program involves steps similar to any problem-solving task. There are five main ingredients in the programming process:

1. Defining the problem
2. Planning the solution
3. Coding the program
4. Testing the program
5. Documenting the program

1. Defining the Problem

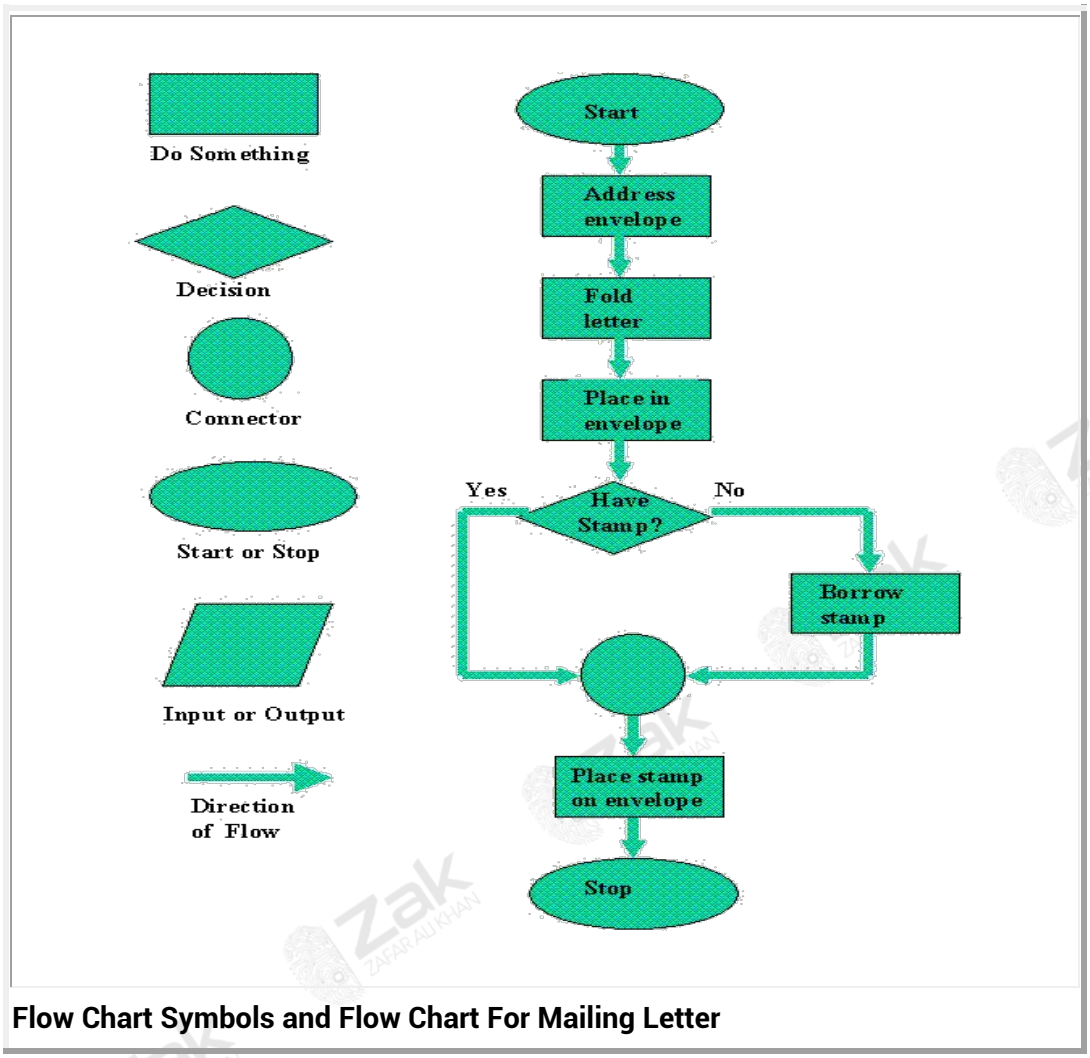
Suppose that, as a programmer, you are contacted because your services are needed. You meet with users from the client organization to analyze the problem, or you meet with a systems analyst who outlines the project. Specifically, the task of defining the problem consists of identifying what it is you know (input-given data), and what it is you want to obtain (output-the result). Eventually, you produce a written agreement that, among other things, specifies the kind of input, processing, and output required. This is not a simple process.

Two common ways of planning the solution to a problem are to draw a flowchart and to write pseudocode, or possibly both. Essentially, a flowchart is a pictorial representation of a step-by-step solution to a problem. It consists of arrows representing the direction the program takes and boxes and other symbols representing actions. It is a map of what your program is going to do and how it is going to do it. The American National Standards Institute (ANSI) has developed a standard set of flowchart symbols. Figure 1 shows the symbols and how they might be used in a simple flowchart of a common everyday act-preparing a letter for mailing.

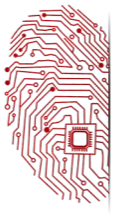
Pseudocode is an English-like nonstandard language that lets you state your solution with more precision than you can in plain English but with less precision than is required when using a formal programming language. Pseudocode permits you to focus on the program logic without having to be concerned just yet about the precise syntax of a particular programming language. However, pseudocode is not executable on the computer. We will illustrate these later in this chapter, when we focus on language examples.

Topic: 2.4.1 Programming

2. Planning the Solution



Flow Chart Symbols and Flow Chart For Mailing Letter



Topic: 2.4.1 Programming

3. **Coding the Program**

As the programmer, your next step is to code the program-that is, to express your solution in a programming language. You will translate the logic from the flowchart or pseudocode-or some other tool-to a programming language. As we have already noted, a programming language is a set of rules that provides a way of instructing the computer what operations to perform. There are many programming languages: BASIC, COBOL, Pascal, FORTRAN, and C are some examples. You may find yourself working with one or more of these. We will discuss the different types of languages in detail later in this chapter.

Although programming languages operate grammatically, somewhat like the English language, they are much more precise. To get your program to work, you have to follow exactly the rules-the syntax-of the language you are using. Of course, using the language correctly is no guarantee that your program will work, any more than speaking grammatically correct English means you know what you are talking about. The point is that correct use of the language is the required first step. Then your coded program must be keyed, probably using a terminal or personal computer, in a form the computer can understand.

One more note here: Programmers usually use a text editor, which is somewhat like a word processing program, to create a file that contains the program. However, as a beginner, you will probably want to write your program code on paper first.

4. **Testing the Program**

some experts insist that a well-designed program can be written correctly the first time. In fact, they assert that there are mathematical ways to prove that a program is correct. However, the imperfections of the world are still with us, so most programmers get used to the idea that their newly written programs probably have a few errors. This is a bit discouraging at first, since programmers tend to be precise, careful, detail-oriented people who take pride in their work. Still, there are many opportunities to introduce mistakes into programs, and you, just as those who have gone before you, will probably find several of them.
Eventually, after coding the program, you must prepare to test it on the computer. This step involves three phases:

a. **Desk-checking.** This phase, similar to proofreading, is sometimes avoided by the programmer who is looking for a shortcut and is eager to run the program on the computer once it is written. However, with careful desk-checking you may discover several errors and possibly save yourself time in the long run. In desk-checking you simply sit down and mentally trace, or check, the logic of the program to attempt to ensure that it is error-free and workable.
Many organizations take this phase a step further with a walkthrough, a process in which a group of programmers (your peers) review your program and offer suggestions in a collegial way.

b. **Translating.** A translator is a program that (1) checks the syntax of your program to make sure the programming language was used correctly, giving you all the syntax-error messages, called diagnostics, and (2) then translates your program into a form the computer can understand. A by-product of the process is that the translator tells you if you have improperly used the programming language in some way. These types of mistakes are called syntax errors. The translator produces descriptive error messages. For instance, if in FORTRAN you mistakenly write N=2 *(I+J))-which has two closing parentheses instead of one-you will get a message that says, "UNMATCHED PARENTHESES." (Different translators may provide different wording for error messages.) Programs are most commonly translated by a *compiler*. A compiler translates your entire program at one time. The translation involves your original program, called a source module, which is transformed by a compiler into an object module. Prewritten programs from a system library may be added during the link/load phase, which results in a load module. The load module can then be executed by the computer.

c. **Debugging.** A term used extensively in programming, debugging means detecting, locating, and correcting bugs (mistakes), usually by running the program. These bugs are logic errors, such as telling a computer to repeat an operation but not telling it how to stop repeating. In this phase you run the program using test data that you devise. You must plan the test data carefully to make sure you test every part of the program.

5. Documenting the program

Documenting is an ongoing, necessary process, although, as many programmers are, you may be eager to pursue more exciting computer-centered activities. Documentation is a written detailed description of the programming cycle and specific facts about the program. Typical program documentation materials include the origin and nature of the problem, a brief narrative description of the program, logic tools such as flowcharts and pseudocode, data-record descriptions, program listings, and testing results. Comments in the program itself are also considered an essential part of documentation. Many programmers document as they code. In a broader sense, program documentation can be part of the documentation for an entire system.

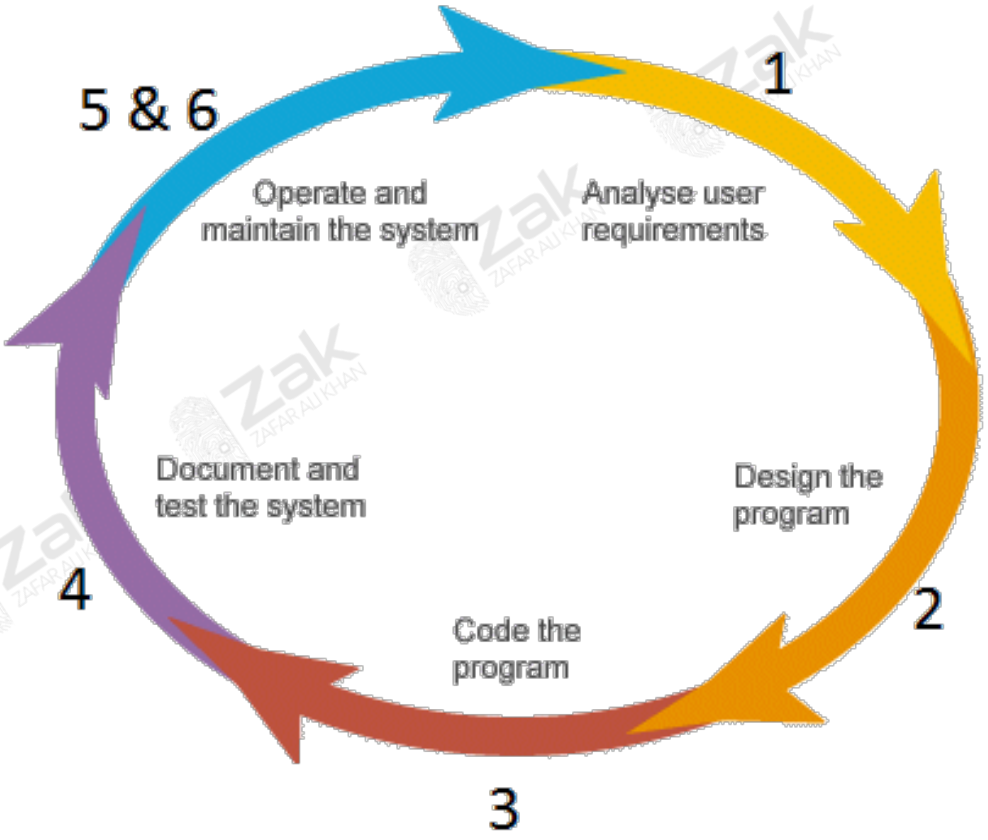
The wise programmer continues to document the program throughout its design, development, and testing. Documentation is needed to supplement human memory and to help organize program planning. Also, documentation is critical to communicate with others who have an interest in the program, especially other programmers who may be part of a programming team. And, since turnover is high in the computer industry, written documentation is needed so that those who come after you can make any necessary modifications in the program or track down any errors that you missed.

Topic: 2.4.1 Programming

Program development life cycle (PDLC)

When programmers build software applications, they just do not sit down and start writing code. Instead, they follow an organized plan, or **methodology**, that breaks the process into a series of tasks. There are many application development methodologies just as there are many programming languages. There different methodologies, however, tend to be variations of what is called the program development life cycle (PDLC).

The **program development life cycle (PDLC)** is an outline of each of the steps used to build software applications. Similarly to the way the system development life cycle (SDLC) guides the systems analyst through development of an information system, the program development life cycle is a tool used to guide computer programmers through the development of an application. The program development lifecycle consists of six steps.

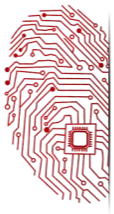


03-111-222-ZAK
OlevelComputer
AlevelComputer
@zakonweb
zak@zakonweb.com
www.zakonweb.com

Topic: 2.4.1 Programming

The Six Steps of PDLC

STEP	PROCEDURE	DESCRIPTION
1	**Analyze the problem**	Precisely define the problem to be solved, and write program specifications – descriptions of the program's inputs, processing, outputs, and user interface.
2	**Design the program**	Develop a detailed logic plan using a tool such as pseudocode, flowcharts, object structure diagrams, or event diagrams to group the program's activities into modules; devise a method of solution or algorithm for each module; and test the solution algorithms.
3	**Code the program**	Translate the design into an application using a programming language or application development tool by creating the user interface and writing code; include internal documentation – comments and remarks within the code that explain the purpose of code statements.
4	**Test and debug the program**	Test the program, finding and correcting errors (debugging) until it is error free and contains enough safeguards to ensure the desired results.
5	**Formalize the solution**	Review and, if necessary, revise internal documentation; formalize and complete the end-user (external) documentation.
6	**Maintain the program**	Provide education and support to end users; correct any unanticipated errors that emerge and identify user-requested modifications (enhancements). Once errors or enhancements are identified, the program development life cycle begins again at Step 1.



Topic: 2.4.1 Programming

How to expose faults in programs, and how to avoid them??

Errors in computer solutions are called bugs. They create two problems. One is that the error needs to be corrected; this is normally fairly straightforward because most errors are caused by silly mistakes. The second problem, however, is much more complicated, the errors have to be found before they can be corrected. Finding where the error is and identifying it, can be very difficult and there are a number of techniques available for solving such problems.

1. Translator diagnostics. Each of the commands that are in the original program is looked at separately by the computer translator to execute it. Each command will have a special word which says what sort of command it is. The translator looks at the special word in the command and then goes to its dictionary to look it up. The dictionary tells the translator program what the rules are for that particular special word. If the word has been typed in wrongly, the translator will not be able to find it in the dictionary and will know that something is wrong. If the word is there, but the rules governing how it should be used have not been followed properly, the translator will know that there is something wrong. Either way, the translator program knows that a mistake has been made, it knows where the mistake is and, often, it also knows what mistake has been made. A message detailing all this can be sent to the programmer to give hints as what to do. These messages are called translator diagnostics.

2. Debugging tools. Sometimes the program looks alright to the translator, but it still doesn't work properly. Debugging tools are part of the software which help the user to identify where the errors are. The techniques available include:

a) Cross-referencing. This software checks the program that has been written and finds places where particular variables have been used. This lets the programmer check to make sure that the same variable has not been used twice for different things.

b) Traces. A trace is where the program is run and the values of all the relevant variables are printed out, as are the individual instructions, as each instruction is executed. In this way, the values can be checked to see where they suddenly change or take on an unexpected value.

c) Variable dumps (check/watch). At specified parts of the program, the values of all the variables are displayed to enable the user to compare them with the expected results.

3. Desk checking is sometimes known as a dry run. The user works through the program instructions manually, keeping track of the values of the variables. Most computer programs require a very large number of instructions to be carried out, so it is usual to only dry run small segments of code that the programmer suspects of containing an error.

Topic: 2.4.1 Programming

4. Focus on the top-down design in section 2.1.c we discussed the splitting up of a problem into smaller and smaller parts, until each part was a manageable size. When the parts were combined they would produce a solution to the original problem. Because we are starting with a big problem and splitting it into smaller problems, this was called top-down design. When the program is written, each small program is written separately, allowing the small programs to be tested thoroughly before being combined. This is called bottom-up programming. The technique is particularly useful for testing the finished program because it is far easier to test a lot of small programs than it is to test one large one. One problem can arise, because the small programs have to be joined together, these joints have to be tested too to make sure that no silly mistakes have been made like using the same variable name for two different things in two parts of the program (tested by cross referencing).

5. Test strategies are important to establish before the start of testing to ensure that all the elements of a solution are tested, and that unnecessary duplication of tests is avoided.

6. (This section is with respect to VB6) If you reach a point in your code that calls another procedure (a function, subroutine, or the script associated with an object or applet), you can enter (*step into*) the procedure or run (*step over)* it and stop at the next line. At any point, you can jump to the end (*step out*) of the current procedure and carry on with the rest of the application.

You may want to step through your code and trace code execution because it may not always be obvious which statement is executed first. Use these techniques to trace the execution of code:

- **Break points** can be set within program code so that the program stops temporarily to check that it is operating correctly upto that point.
- **Step Into**: Traces through each line of code and steps into procedures. This allows you to view the effect of each statement on variables.
- **Step Over**: Executes each procedure as if it were a single statement. Use this instead of **Step Into** to step across procedure calls rather than into the called procedure.
- **Step Out**: Executes all remaining code in a procedure as if it were a single statement, and exits to the next statement in the procedure that caused the procedure to be called initially.
- **Run to Cursor**: Allows you to select a statement in your code where you want execution to stop. This allows you to "step over" sections of code (for example, large loops).

03-111-222-ZAK | OlevelComputer AlevelComputer | @zakonweb | zak@zakonweb.com | www.zakonweb.com

Topic: 2.4.1 Programming

To trace execution from the current statement

- From the **Debug** menu, choose **Step Into**, **Step Over**, **Step Out**, or **Run To Cursor**.

To trace execution from the beginning of the program

- From the **Debug** menu, choose **Step Into**, **Step Over**, **Step Out**, or **Run To Cursor**.