Main content

Chapter

9

Introduction to High-Level Language Programming

Change font sizeMain content

Chapter Contents

Chapter Introduction

r studying this chapter, you will be able to:

- Explain the advantages of high-level programming languages over assembly language
- Describe the general process of translation from high-level source code to object code
- Name the five procedural programming languages used in the examples of this chapter
- Explain the favorite number and data cleanup examples for each programming language
- Explain why the software development life cycle is necessary for creating large software programs
- List the steps in the software development life cycle, explain the purpose of each, and describe the products of each
- Explain how agile software development differs from the traditional waterfall model

9.1 The Language Progression

As of the end of Chapter 8, we have a complete and workable computer system. We have created a virtual environment in which we imagine that we are communicating directly with the computer, even though we are using a language (assembly language) that is far more suited to human communication than is (binary) machine language. We have learned about the system software needed to create and support this virtual environment, including the assembler that translates assembly language programs into machine language, as well as the operating system that accepts requests to load and execute a program and coordinates and manages the other software tools needed to accomplish this task. Our system also includes the network technologies and protocols that extend the virtual world across our campus, throughout our office building, and around the world, and we are aware of the need for protection against the security threats to which we are exposed as our virtual world widens.

But this puts us somewhat ahead of our story. In Chapter 6, we talked about the progression from machine language to assembly language, but today, using computers to solve problems usually involves writing programs in a high-level programming language. This section continues our discussion of the progression of programming languages from assembly language (where we left off in our language story) to high-level languages.

9.1.1 Where Do We Stand and What Do We Want?

At the end of Chapter 6, we were back in the “early days” of computing—the 1950s—when assembly language had just come into existence. As a step up from machine language, it was considered a reasonable programming environment because the people writing computer programs were for the most part technically oriented. Many had backgrounds in engineering or (as it would later be called) computer science. They were familiar with the inner workings of a computer and were accustomed to dealing with difficult problems steeped in mathematical notation, so the tedious precision of assembly language programming did not deter them. Also, because assembly language is so closely tied to machine language, assembly language programmers could see the kinds of processor activity that assembly language instructions would generate. By being sufficiently clever in their choice of instructions, they could often shave a small amount off the execution time or amount of memory that their programs required. For example, the following sequence of assembly language instructions:

```
        LOAD     X
        ADD      ONE
        STORE    X
        .        .
        .        .
        .        .
ONE:    .DATA    1
```

could be replaced by the single instruction

```
INCREMENT X
```

This is not the sort of performance improvement obtained by changing from a sequential search algorithm to a binary search algorithm. Instead, it is a fine-tuning improvement that may save a few millionths of a second, or even a few seconds if these instructions occur inside a loop that is executed many times. But remember that in this era, people did not have powerful personal computers at their disposal. Programmers were competing for the resources of a mainframe computer, and although these computers were physically large and very expensive, they did not have the processing speed or memory capacity of even the most modest of today’s systems. Conserving machine resources, even in tiny amounts, was an important part of computer programming.

Over the next few decades, however, computer usage became widespread, permeating society to a degree that would probably not have been believed in the 1950s. “Nontechie” types needed to write programs too, and they demanded a more user-friendly programming environment. This was provided through the use of high-level programming languages (which we talk about in this chapter and the next) and also through evolving operating systems and other system software (which were discussed in Chapter 6). In turn, these high-level languages opened the door to new programmers. Also during this period, incredible technological strides made machines so powerful that conserving resources was no longer the critical issue it once was, and the overhead of execution time occasioned by the use of high-level programming languages became acceptable.

Let's review some of the aspects of assembly language programming that made people look for better alternatives. Suppose our task is to add two integers. In the assembly language of Chapter 6, the following instructions would have to be included after the instructions that assign legitimate data values to *B* and C.

```
            LOAD        B
            ADD         C
            STORE       A
            .           .
            .           .
            .           .
A:          .DATA       0
B:          .DATA       0
C:          .DATA       0
```

The three .DATA statements reserve storage for signed integers, generate the binary representation of the integer value 0 to occupy those storage locations initially, and ensure that the labels *A, B*, and *C* are bound to those memory locations. But at this point in the execution of our assembly language program, the initial 0 values for *B* and *C* have been written over by legitimate data values. The LOAD statement then copies the current contents of the memory location labeled *B* into the ALU register *R*, the ADD statement adds the current contents of the memory location labeled *C* to what is currently in register *R*, and the STORE instruction copies the contents of *R* (which is now *B* + *C*) into the memory location labeled *A*.

To perform a simple arithmetic task, we had to manage all the data movement of the numbers to be combined as well as the resulting answer. This is a microscopic view of a task—we'd like to be able to say something like "add *B* and *C*, and call the result *A*," or better yet, something like "." But each assembly language statement corresponds to at most one machine language statement (you may recall from Chapter 6 that the pseudo-op .DATA statements do not generate any machine language instructions). Therefore, individual assembly language statements, though easier to read, can be no more powerful than the operations of the underlying instruction set. For the same reason, assembly language programs are machine specific. An assembly language statement that runs on machine X is nothing but a slightly "humanized" variant of the machine language statement for X, and it will not execute on a machine Y that has a different instruction set. Indeed, machine Y's assembler won't know what to do with such a statement. Finally, assembly language instructions are rather stilted. STORE A does not sound much like the sort of language we customarily speak, though STORE is certainly more expressive than its binary machine language counterpart. To summarize, assembly language has the following disadvantages:

- The programmer must "manually" manage the movement of data items between and among memory locations and registers (although such data items can be assigned mnemonic names).
- The programmer must take a microscopic view of a task, breaking it down into tiny subtasks—ADD, COMPARE, INCREMENT—that direct what is going on in individual memory locations and registers.
- An assembly language program is machine specific.
- Statements are not natural-language-like (although operations are given mnemonic code words as an improvement over a string of bits).

We would like to overcome these deficiencies, and *high-level programming languages* were created to do just that. Thus, we have the following expectations of a program written in a high-level language:

- The programmer need not manage the details of the movement of data items within memory or pay any attention to exactly where those items are stored.
- The programmer can take a macroscopic view of tasks, thinking at a higher level of problem solving (add *B* and *C*, and call the result *A*). The "primitive operations" used as building blocks in algorithm construction (see Chapter 1) can be larger.
- Programs are portable rather than machine specific.
- Programming statements are closer to natural language and use standard mathematical notation.

High-level programming languages are often called **third-generation languages**, reflecting the progression from machine language (first generation) to assembly language (second generation) to high-level language. They are another step along the continuum shown in Chapter 6, Figure 6.3. This also suggests what by now you have suspected: We've reached another layer of abstraction, another virtual environment designed to further distance us from the low-level hardware components of the machine.

9.1.2 Getting Back to Binary

There is a price to pay for this higher level of abstraction. When we moved from machine language to assembly language, we needed a piece of system software—an assembler—to translate assembly language instructions into binary machine language instructions (**object code**). This was necessary because the computer itself—that is, the collection of electronic devices—can respond only to machine language instructions. Now that we have moved up another layer with regard to the language in which we communicate, we need a different type of translator to convert our high-level language instructions into machine language instructions. This new type of translator is called a *compiler*. Rather than doing the entire translation job right down to object code, a compiler often translates high-level language instructions (**source code**) only into assembly language (the hard part of the translation). It then turns the final (simple) translation job over to an assembler that finishes the task by converting the assembly language into binary, exactly as we showed in Chapter 6.

Some tasks (e.g., sorting and searching) need to be performed often, as part of solving other problems. The code for such a useful task can be written as a group of high-level language instructions and thoroughly tested to be sure it is correct. Then the object code for the task can be stored in a **code library**. A program can just request that a copy of this object code be included along with its own object code. A piece of system software called a **linker** inserts requested object code from code libraries into the object code for the requesting program. The resulting object code is often called an **executable module**. Thus, a high-level program might go through the transitions shown in Figure 9.1. Compare this with Figure 6.4.

Figure 9.1 Transitions of a high-level language program

The work of the compiler is discussed in more detail in Chapter 11. Let us note here, however, that the compiler has a much tougher job than the assembler. An assembler has a one-for-one translation task because each assembly language instruction corresponds to at most one machine language instruction. A single high-level programming language instruction, on the other hand—precisely because a high-level language is more expressive than assembly language—can “explode” into many assembly language instructions.

Change font size Main content

Chapter Contents

9.2 A Family of Languages

Most of today’s popular high-level programming languages fall into the same philosophical family; they are **procedural languages** (also called **imperative languages**). A program written in a procedural language consists of sequences of statements that manipulate data items. The programmer’s task is to devise the appropriate step-by-step sequence of “imperative commands”—instructions in the programming language—that, when carried out by the computer, accomplish the desired task.

Procedural languages follow directly from the Von Neumann computer architecture described in Chapter 5, an architecture characterized by sequential fetch-decode-execute cycles. A random access memory stores and fetches values to and from memory cells. Thus, it makes sense to design a language whose most fundamental operations are storing and retrieving data values. For example,

Even though a high-level programming language allows the programmer to think of memory locations in abstract rather than physical terms, the programmer is still directing, via program instructions, every change in the value of a memory location.

Ada, C++, C#, Java, and Python Online Chapters

To explore programming through the lens of a particular programming language, and to get a sense of what programming in a high-level language is like, try one or more of the online language chapters found on the companion site for this text (www.cengage.com) and in the MindTap. Each chapter includes language-specific exercises and practice problems.

The languages we have chosen to discuss from this procedural language family are Ada, C++, C#, Java, and Python. These languages differ in the rules (the **syntax**) for exactly how statements must be written and in the meaning (**semantics**) of correctly written statements. Rather than fill up pages and pages of this book with the details of each of these languages, we've created online chapters for you to investigate the language(s) of your choice (or your instructor's choice) in much more detail than you will see here. See the Special Interest Box for information on accessing these online chapters.

Change font sizeMain content

Chapter Contents

9.3Two Examples in Five-Part Harmony

At this point you might (or might not) have studied one or more of the online chapters for Ada, C++, C#, Java, or Python. In either case, you might be interested to see how these languages are similar and how they differ. In this section, we'll look at two sample problems and their solutions in each of the five languages. Don't be overly concerned about the details; just try to get the "big picture" in each of these solutions.

Change font sizeMain content

Chapter Contents

9.3.1Favorite Number

Our first problem is trivially simple. Nonetheless, it will allow you to observe some of the significant syntactic differences in these five languages. A pseudocode version is shown in Figure 9.2.

Figure 9.2

Pseudocode algorithm for favorite number

```
Get value for the user's favorite number, n
Increase n by 1
Print a message and the new value of n
```

Next, we show this same algorithm implemented in Ada (Figure 9.3), C++ (Figure 9.4), C# (Figure 9.5), Java (Figure 9.6), and Python (Figure 9.7). The program code in each figure is easily recognizable as a formalized version of the pseudocode—it uses some mechanism to get the user's favorite number, then sets the value of *n* to *n* + 1, and finally writes the output. The syntax, however, varies with the language. In particular, each language has its own way of reading input (from the keyboard), performing a computation, and writing output (to the screen). There's also a variation in the amount of "startup" code required to get to the actual algorithm implementation part. Each language has a notation (--, //, or #) that denotes a program comment, and each language has its own set of special "punctuation marks." For example, four of the five languages (Python being the exception) require a semicolon to terminate an executable program statement.

Figure 9.3Ada program for favorite number

Figure 9.4 **C++ program for favorite number**

Figure 9.5 **C# program for favorite number**

Figure 9.6 **Java program for favorite number**

Figure 9.7 **Python program for favorite number**

9.3.2 Data Cleanup (Again)

Now that you've seen a bare-bones sample for each language, let's implement a solution to a considerably more interesting problem. In Chapter 3, we discussed several algorithms to solve the data cleanup problem. In this problem, the input is a set of integer data values (answers to a particular question on a survey, for example) that may contain 0s, although 0s are considered invalid data. The output is to be a clean data set where the 0s have been eliminated. Figure 9.8 is a copy of Figure 3.16. It shows the pseudocode for the converging-pointers data cleanup algorithm, the most time- and space-efficient of the three data cleanup algorithms from Chapter 3.

Figure 9.8 **The converging-pointers algorithm for data cleanup**

Our pseudocode does not specify the details of how to "get values." In the favorite number example, the single input value was entered at the keyboard. The survey data, however, is probably already stored in an electronic file. It might have been collected via an online survey that captured the responses or via paper forms that have been scanned to capture the data in digital form. Designing our programs to read input data from a file, however, is a bit more than we want to get into, so we'll again assume the input data is typed in at the keyboard.

The pseudocode algorithm of Figure 9.8 is implemented in Ada (Figure 9.9), C++ (Figure 9.10), C# (Figure 9.11), Java (Figure 9.12), and Python (Figure 9.13).

Figure 9.9 **Ada converging-pointers algorithm**

Figure 9.10 **C++ converging-pointers algorithm**

Figure 9.11C# converging-pointers algorithm

Figure 9.12Java converging-pointers algorithm

Figure 9.13Python converging-pointers algorithm

As with the previous, simpler example, you can see that each program follows the outline of the pseudocode algorithm. Each language supports if statements and while loops. The extent of the while loop is denoted by curly braces {} in three of the languages, by *loop … end loop* in Ada, and (although this is less evident) by a colon and indentation in Python. There are several different ways of creating the memory space to hold the list of data values. And, as we saw before, each language does I/O (from keyboard to screen) using different syntax, and requires different “startup” code. But the output of each version looks like Figure 9.14, where boldface indicates user input.

Figure 9.14Output from the various data cleanup implementations

Each of the five languages supports many more programming features than are shown in these simple examples. Consult the online language chapters for more in-depth programming concepts (including functions, parameter passing, object-oriented programming, and graphical programming) supported by each of these languages.

Main content

Chapter Contents

9.4Feature Analysis

If you have studied one (or more) of the online chapters for Ada, C++, C#, Java, or Python, then the “features” of that programming language will be familiar to you. You can compare them with the features of the other languages by scanning Figure 9.15 starting. If you haven’t studied any of these languages in detail, the figure will still give you a brief reference on each of them. Figure 9.15 compares only the features that are included in the online chapter for each language, so it should not be viewed as a comprehensive list of features for any of these languages.*

Figure 9.15Feature analysis of five high-level languages (examples in blue)

9.5Meeting Expectations

At the beginning of this chapter, we gave four expectations for programs written in a high-level programming language. Now that we have been introduced to the essentials of writing programs in such a language, it is time to see how well these expectations have been met.

1. *The programmer need not manage the details of the movement of data items within memory or pay any attention to exactly where those items are stored*. The programmer's only responsibilities are to declare (or in the case of Python, create) all constants and variables the program will use. This involves selecting identifiers—that is, symbolic names—to represent the various data items and indicating the data type of each, either in the declaration statement or, in the case of Python, in an assignment statement. The identifiers can be descriptive names that meaningfully relate the data to the problem being solved. Data values are moved as necessary within memory by program instructions that simply reference these identifiers, without the programmer knowing which specific memory locations contain which values, or what value currently exists in an ALU register. The concepts of memory address and movement between memory and the ALU, along with the effort of generating constant data values, have disappeared.
2. *The programmer can take a macroscopic view of tasks, thinking at a higher level of problem solving*. Instead of moving data values here and there and carefully orchestrating the limited operations available at the machine language or assembly language level, the programmer can, for example, write the formula to compute the circumference of a circle given its radius. The details of how the instruction is carried out—how the data values are moved about and exactly how the multiplication of real number values is done—are handled elsewhere. Compare the power of conditional and looping instructions—which are tools for algorithmic problem solving and resemble the operations with which we constructed algorithms in pseudocode—with the assembly language instructions LOAD, STORE, JUMP, and so on, which are tools for data and memory management.
3. *Programs written in a high-level language will be portable rather than machine specific*. Program developers use a variety of approaches to make their programs portable to different platforms. For programs written in most high-level languages, the program developer runs through the entire translation process to produce an executable module (complete object code) as shown in Figure 9.1, and it is the executable module that is sold to the user, who runs it on his or her own machine. The program developer doesn't usually give the user the source code to the program, for a multitude of reasons. First, the program developer does not want to give away the secrets of how the program works by revealing the code to someone who could make a tiny modification and then sell this "new" program. Second, the program developer wants to prevent the user from being able to change the code, rendering a perfectly good program useless, and then complaining that the software is defective. And finally, if the program developer

distributes the source code, then all users must have their own translators to get the executable module needed to run on their own machines.

The developer can compile the program on any kind of machine as long as there is a compiler on that machine for the language in which the program is written. However, there must be a compiler for each (high-level language, machine-type) pair. If the program is written in C++, for example, and the program developer wants to sell his or her program to be used on a variety of computers, he or she needs to compile the same program on a Windows PC using a C++ compiler for the PC, on a Mac using a C++ compiler for the Mac, and so on, to produce all the various object code versions. The program itself is independent of the details of each particular computer's machine language because each compiler takes care of the translation. This is the "portability" we seek from high-level language programs.

Even the availability of the appropriate compiler may not guarantee that a program developed on one type of machine can be compiled on a different type of machine. Each programming language has a certain core of instructions that are considered standard. Any respectable compiler for that language must support that core. In fact, national and international standards groups such as ANSI (American National Standards Institute) and ISO (International Organization for Standardization), which develop standards for an incredible number of things, also develop standards for programming languages. Compilers are thus built to support "ANSI-standard language X." However, there are often useful features or types of instructions that are not considered a standard part of the language and that some compilers support and some do not. If a program is written to take advantage of some of these extra features that are available on a particular compiler—often referred to as extensions, or "bells and whistles"—the program might not work with a different compiler that does not support these extensions. The price for using nonstandard features is the risk of sacrificing portability.

The standardization process (for anything, including a programming language) is necessarily a slow one because it seeks to satisfy the interests of a number of groups, such as consumers, industry, and government. If official standardization comes too late, it must bow to what may have become a de facto standard by common usage. If standardization is imposed too early, it may thwart the development of new ideas or technology.

Newer languages such as Java and C# were developed specifically to run on a variety of hardware platforms without the need for a separate compiler for each type of machine. A compiler for Java or C# translates the source code program into very low-level code (called *bytecode* in Java and *Microsoft Intermediate Language* in C#). The resulting programs are not machine-language code for any real machine, but rather for an "idealized" virtual machine. The machine language for this virtual machine can easily be translated into any specific machine language. The program developer only needs to do one compilation to produce low-level virtual machine code and then distribute this virtual machine code to the various users. The final translation/execution of this virtual code into the machine language of a particular user's machine is done by a small piece of software on the user's machine (a Java bytecode interpreter for Java or a Just In Time compiler for C#).

The Python language takes a still different approach to portability. A Python program is *interpreted* rather than compiled, which means that it is translated from source code into object code every time it is executed. As a consequence, each user's machine has to have a Python interpreter, but such an interpreter is available for virtually every operating system, and is small, quick, and free. Python is an **open source programming language**. The open source philosophy is based on the belief that better software is developed if a large group of individuals can examine and modify the code; essentially, "the more eyes, the merrier." The open source movement has attracted many skilled and dedicated people who, usually working for free, are motivated by the goals of producing high-quality software and working cooperatively with like-minded individuals. In the spirit of open source code development, Python developers are happy to send their source code to users. (Other well-known open source programming projects include the online encyclopedia Wikipedia and the Mozilla Firefox browser.)

4. *Programming statements in a high-level language will be closer to natural language and will use standard mathematical notation*. High-level languages provide us with statements that give natural implementations of pseudocode instructions such as "while condition do something …" or "if condition do something…." Although pseudocode is still somewhat stilted, it is nonetheless close to natural language. We can also use standard mathematical notation such as $A + B$ and $A > B$.

Change font sizeMain content

Chapter Contents

9.6The Big Picture: Software Engineering

Because any high-level language program ultimately must be translated by a compiler or interpreter, there are very stringent syntax rules about punctuation, use of keywords, and so on for each program statement. If something about a program statement cannot be understood by the compiler, then the compiler cannot translate the program into machine language; if the compiler cannot translate the program, you get an error message and the program's instructions cannot be executed. For novice programmers this can happen many times during the development of new software. This obstacle leads beginning programming students to conclude that the bulk of the effort in the software development process should be devoted to implementation—that is, restating an algorithm in terms of computer code and ridding that code of all syntax errors so that it finally executes.

In fact, implementation represents a relatively small part of the **software development life cycle**—the overall sequence of steps needed to complete a large-scale software project. Studies have shown that on big projects (system software such as operating systems or compilers, for example, or large applications such as a program to manage an investment company's portfolio), the initial implementation of the program may occupy only 10–20% of the total time spent by programmers and designers. About 25–40% of their time is spent on problem specification and program design—important planning steps that must be completed prior to implementation. Another 40–65% is spent on the tasks that follow implementation—reviewing, modifying, fixing, and improving the original code and writing finished documentation. Although there is no universal agreement on the exact sequence of steps in the software development life cycle, Figure 9.16 summarizes one possible breakdown. We'll discuss each of these steps shortly.

Figure 9.16

Steps in the software development life cycle

Before Implementation

Feasibility study
Problem specification
Program design
Algorithm selection or development, and analysis
Implementation

Coding
Debugging
After Implementation

Testing, verification, and benchmarking
Documentation
Maintenance

Beginning programming students might not see or appreciate the entire software development life cycle because the programming assignments usually solved in introductory classes are extremely and unrealistically small. This can create a skewed and misleading view of the software development process. It is somewhat akin to a civil engineering student building a match-stick bridge that is just a couple of inches long; a multitude of new problems must be addressed when that task is scaled up to a full-sized, real-life bridge.

Change font sizeMain content

Chapter Contents

9.6.1 Scaling Up

The programs that students write in an introductory course may have 50–100 lines of code. Even by the end of the course, programs are usually not longer than a few hundred lines. However, real-world programs are often orders of magnitude larger. Compilers or some operating systems contain many thousands of lines. Truly large software systems, such as the NASA Space Shuttle ground control system and the data management system of the U.S. Census Bureau, may require the development of millions of lines of code. To give you an idea of how very large that is, a printed listing of a 1-million-line program would be almost 17,000 pages long—about the size of 50 books. The difference in complexity between a million-line software package and a 100-line homework assignment is equivalent to the difference in scale between a 300-page novel and a single sentence!

Figure 9.17 categorizes software products in terms of size, the number of programmers needed for development, and the duration of the development effort. These numbers are very rough approximations, but they give you an idea of the size of some widely used software packages. Analogous building construction projects are also listed.

Figure 9.17

Size categories of software products

Category	Typical Number of People	Typical Duration	Product Size in Lines of Code	Examples	Building Analogy
Trivial	1	1–2 weeks	< 500	Student homework assignments	Small home improvement
Small	1–3	A few weeks or months	500–2,000	Student team projects, advanced course assignments	Adding on a room
Medium	2–5	A few months to 1 year	2,000–10,000	Research projects, simple production software such as assemblers, editors, recreational and educational software	Single-family house
Large	5–25	1–3 years	10,000–100,000	Most current applications—word processors, spreadsheets, operating systems for small computers, compilers, iPhone apps	Small shopping mall
Very Large	25–100	3–5 years	100,000–1 M	Airline reservations systems, inventory control systems for multinational companies	Large office building
Extremely Large	> 100	> 5 years	> 1 M	Large-scale real-time operating systems, advanced military work, international telecommunications networks	Massive skyscraper

Virtually all software products developed for the marketplace are neither trivial nor small. Most fall instead into either the Medium or the Large category of Figure 9.17. The Very Large and Extremely Large categories are enormous intellectual enterprises. It would be impossible to develop correct and maintainable software systems of that size without extensive planning and design, just as it is impossible to build a 50-story skyscraper without paying a great deal of attention to initial project planning and project management. Neither endeavor can be carried out by a single individual; a team development effort is essential in building software, just as in constructing buildings. Such projects also entail estimation of costs and budgets, personnel management, and scheduling issues, which are typical concerns for large engineering projects, and therefore the term **software engineering** is often applied to these large-scale software development projects.

Change font sizeMain content

Chapter Contents

9.6.2 The Software Development Life Cycle

Each step in the software development life cycle, as shown in Figure 9.16 and described in the following paragraphs, has its own purpose and activities. Each should also result in a written document that reflects past decisions and guides future actions. Keep in mind that every major software project is developed as a team effort, and these documents help keep various members of the team informed and working toward a common goal.

1. *The feasibility study*—The **feasibility study** evaluates a proposed project and compares the costs and benefits of various solutions. One choice might be to buy a new computer system for this project. Even though the cost of computer hardware has dropped dramatically, computers are still significant purchases. In addition to the costs of the machine itself, there may be costs for peripherals such as 3-D printers or telecommunications links. The costs of software (purchased or produced in-house); equipment maintenance; salary for developers or consultants, technical support people, and data entry clerks—these must all be factored in, as must the costs incurred in training new users on the system. The overall cost of using a computer to solve a problem can be much higher than expected, and it can also be more than the value of the information produced. Other options should also be considered. Thus, the feasibility study should address the following question:

Vital Statistics for Real Code

The Windows operating system was created by Microsoft Corporation. Development of this system (originally called the Interface Manager) began in 1981. Subsequently renamed Microsoft Windows, the system was not released until November 1985, after 55 person-years of effort. Since then, there have been a number of evolutions. Each new version provided more and more services with a resulting expansion in the number of lines of source code (SLOC). Here is some sample data, including, for comparison, information from Chapter 6 (Special Interest Box, “Now That’s Big”) on Mac OS X Tiger 10.4, and on the Linux kernel version 4.1 operating system. (Google’s Android operating system, common on many mobile phones, is based on the Linux kernel.)

Name	Release Date	Estimated SLOC (in millions)
Windows 3.1	1992	2.5
Windows 95	1995	8.0
Mac OS X Tiger 10.4	2005	86.0
Windows 7	2009	39.5
Windows 10	2015	50
Linux kernel v. 4.1	2015	20

Operating systems are not the only software packages that fall into the “massive skyscraper” (Extremely Large) category of Figure 9.17. Here are three other examples:

Name	Estimated SLOC (in millions)
Facebook	61
Software in a modern high-end automobile	100
Google Software Services (web search engine, Gmail, YouTube, etc.)	2000

This creeping code growth is a little more striking when viewed visually:

The complexity of software projects of this magnitude is breathtaking. Not only technical capabilities, but also planning, teamwork, communication, and management skills are vital to successful completion.

What are the relative costs and benefits of the following choices?

- Buying a new computer system and writing or buying software
- Writing new software for an existing computer system
- Purchasing the needed resources from a “cloud computing” provider (see Chapter 7)
- Outsourcing the work to a contractor
- Revising the current manual process for solving this problem
- Cutting back the scope of the project to better align it with existing resources
- Cancelling the project entirely and doing without the information that would be generated

At the end of the feasibility study, a *feasibility document* expresses the resulting recommendations. The creation of this document can be a very complex process involving considerations that are the provinces of business, law, management, economics, psychology, and accounting as well as computer science. The purpose of the feasibility study is to make all project stakeholders aware of the costs, risks, and benefits of various development paths as a guide to deciding which approach to use.

2. *Problem specification*—If it is determined that the project is feasible and will benefit from a computer solution, and that software development is to go forward, we move on to the problem specification phase. **Problem specification** involves developing a clear, concise, and unambiguous statement of the exact problem the software is to solve. Because the original problem statement used in the feasibility study is written in a natural language, such as English, it may be unclear, incomplete, or even internally contradictory. This rough initial problem statement must be transformed into a complete, accurate, and consistent problem specification that describes the actions this software should take for every possible input it may encounter. During the problem specification phase, the software developers and their "customers"—those who are commissioning the software and will be its eventual users—must resolve each and every inconsistency, ambiguity, and gap. It is much easier and cheaper to make changes at this stage than to make changes in actual code months down the road. Consider how much more practical it is to change your mind when looking at the blueprints of your new home than after the foundation has been dug and the walls have started to go up.

   The *problem specification document* expresses the final and complete problem specification and guides the software developers in all subsequent decisions. The specification document describes exactly how a program is to behave in all circumstances—not only in the majority of cases, but even under the most unusual and unexpected conditions. It contains a description of the data to be input to the program, the desired results to be computed, and instructions on how these results are to be displayed as output. It may also include limitations on the time allotted to produce those computations or on the amount of memory the program requires.

   Once agreed to by the developer and the customer, the problem specification document becomes essentially a legal contract describing what the developer promises to provide and what the customer agrees to accept. Like a contract, it usually includes a delivery schedule and a price, and it is signed by both the customer and the developer.

3. *Program design*—Now that it is clear what is to be done, the **program design phase** is the time to plan *how* it is to be done. In a traditional programming approach, the **divide-and-conquer** strategy (also called **top-down decomposition**) comes into play. Tasks are broken down into subtasks, which are broken down into sub-subtasks, and so on, until each piece is small enough to code comfortably. These pieces work together to accomplish the overall job. In an **object-oriented programming** approach, the appropriate objects are identified, together with their data and the subtasks they must perform. This allows classes to be designed with variables to store the data, and functions (also called *methods*) to carry out these subtasks. Objects from these classes cooperate to accomplish the total job.

   The larger the project, the more crucial it is to think of it in terms of smaller building blocks, or helpful classes, that are created separately and then properly assembled to solve the problem at hand. Although small programs of 50–100 lines can be thought of in one piece, 100,000-line programs cannot. However, it is possible to treat a 100,000 line program as a collection of about 1,000–2,000 smaller pieces, each containing about 50–100 lines.

The *program design document* breaks the problem into subtasks and sub-subtasks, or into various classes. Some of this design may be documented graphically, through structure charts or through class diagrams that give the properties and functions of each class. Modules that carry out subtasks in a traditional design, or that carry out some service that the objects of a class provide, are ultimately translated into separate sections of code. There must also be a complete specification of each module: what it is to do, what information it needs to know in order to do it, and what the rest of the program needs to get from it when it is done. This information must be sufficiently detailed that a programmer can use the description as a guide to writing code for the module in the language of choice.

Program design is one of the most creative and interesting parts of the overall software development life cycle. It is related to coding in roughly the same way that designing an airplane is related to riveting a wing.

4. *Algorithm selection or development, and analysis*—Once the various subtasks have been identified, algorithms must be found to carry them out. For example, one subtask may be to search a list of numbers for some particular value. In Chapters 2 and 3, we examined two different algorithms for searching—sequential search and binary search. If there is a choice of algorithms, the programmer must determine which is more suitable for this particular task and which is more efficient. It may also be the case that a totally new algorithm has to be developed from scratch. This, too, is a very creative process. Documentation of this phase includes a description of the algorithms chosen or developed, perhaps in pseudocode, an analysis of their efficiency (as we did in Chapter 3), and a rationale for their use.
5. *Coding*—**Coding** is the process of converting the detailed designs into computer code. If the design has been carefully developed, this should be a relatively routine job. Perhaps reusable code can be pulled from a program library, or a useful class can be employed. Coding is the step that usually comes to mind when people think of software development. However, as we have shown, a great deal of important preparatory work precedes the actual production of code. Inexperienced programmers may think that they will save time by skipping the earlier phases and getting right to coding. The opposite is usually true. In all but the most trivial of programs, tackling coding without first doing problem specification, program design, and algorithm selection or development ultimately leads to more time being spent and a poorer outcome. The coding phase also results in a written document, namely, the listing of the program code itself.
6. *Debugging*—**Debugging** is the process of locating and correcting program errors, and it can be a slow and expensive operation that requires as much effort as writing the program in the first place. Errors can occur because a program statement fails to follow the correct rules of syntax, which makes the statement unrecognizable by the compiler and results in an error. Though irritating, these **syntax errors** are accompanied by messages from the compiler that help to pinpoint the problem. Other errors, called **runtime errors**, occur only when the program is run using certain sets of data that result in some illegal operation, such as dividing by zero. The system software also provides messages to help detect the cause of runtime errors. The third, and most subtle, class of errors is **logic errors**. These are errors in the algorithm used to solve the

problem. Some incorrect steps that result in wrong answers are performed, but there are no error messages to help pinpoint the problem. Indeed, the first step in debugging a logic error is to notice that the answers are wrong. For example, if our algorithm calls for us to add 317 to the value of *A*, but we accidently translate this as:

our mistake will not produce an error message as the statement is syntactically correct and semantically meaningful. The only way we know there is an error is to notice that an incorrect value has been stored in *A*.

Debugging has always been one of the most frustrating, agonizing, and time-consuming steps in the programming process. Extensive time spent on debugging usually means that insufficient time was spent on carefully specifying, organizing, and structuring the solution. If the design is poor, then the resulting program is often a structural mess, with convoluted, hard-to-understand logic. On the other hand, devoting careful attention to the design phases can help reduce the amount of debugging and speed up the completion of the project.

Careful documentation of the debugging process includes notes on the problems found and on how the code was changed to solve them. This may prevent later changes from reintroducing old errors.

7. *Testing, verification, and benchmarking*—Even though a program produces correct answers for 1, 5, or even 1,000 data sets, how can we be sure that it is 100% correct and will work on all data? One approach, called **empirical testing**, is to design a special set of test cases, called a **test suite**, and run the program using this special test data. Test data that is carefully chosen to exercise all the different logic paths through a program can help uncover errors. In a conditional statement, for example, one set of data should make the Boolean expression true, so that one block of code is executed. Another set of data should make the same Boolean expression false, so that the other block of code is executed. The quantity of the test data per se does not matter; what matters is that the data covers all the various possibilities that could occur when the program is run. The goal of empirical testing is to make sure that every statement in the program has been executed at least once or, even better, multiple times. Having said that, we should note that in all but the most trivial of programs, it is not feasible to "test all the cases" because there are so many different possibilities. The best that can be said is that the more thorough the testing, the higher the level of our confidence that the program is correct.

   It's not a good plan to wait until the complete program is "finished" before testing takes place. In a program of any size, there are too many places where an error could occur, so the debugging process is extremely difficult at such a late stage. **Unit testing** takes place on each module (subtask code) as it is completed. As these tested modules are combined to work together, **integration testing** is done to see that the modules communicate the necessary data between and among themselves and that all modules work together smoothly. And if anything is changed on an already-tested module, **regression testing** is done to be sure that this change hasn't introduced a new error into code that was previously correct. Thus, testing should not be viewed as a separate and distinct phase of software development that occurs at the very end; instead, it is something that is going on simultaneously with the development of code.

   A second, and totally different, approach to confirming a program's correctness is to use mathematical logic. **Program verification** can be used to prove that if the input data to a program satisfies certain conditions, then, after the program has been run on this data, the output data satisfies certain other conditions. This process is a formal proof, much like the proof of a mathematical theorem that condition B follows from condition A. However, verification is not a magic wand that gives us blanket assurance that a program will behave exactly as we want; it only works if we know that the input conditions will indeed be true and that the output conditions indeed reflect what the program was supposed to produce. Furthermore, the program verification process can be difficult and time consuming. That's why program testing is used much more than formal program verification to reduce the risk of program errors.

In addition to correctness, the problem specification may require that the finished program meets certain performance characteristics such as the amount of time it takes to compute the results. ***Benchmarking*** the program means running it on many data sets to be sure its performance falls within those required limits. At the completion of testing (or verification) and benchmarking, we should have a correct and efficient program that is ready for delivery. Of course, all of the testing, verification, and benchmarking results are committed to paper (well, at least to a digital document) as evidence that the program meets its specifications.

8. *Documentation*—Program documentation is all of the written and online material that makes a program understandable. This includes **internal documentation**, which is part of the program code itself. Good internal documentation consists of choosing meaningful names for program identifiers, using plenty of comments to explain the code, and separating the program into short modules, each of which does one specific subtask. **External documentation** consists of any materials assembled to clarify the program's design and implementation. Although we have put this step rather late in the software development process, note that each preceding step produces some form of documentation. Program documentation goes on throughout the software development life cycle. The final, finished program documentation is written in two forms. **Technical documentation** enables programmers who later have to modify the program to understand the code. Such information as problem specification documents, design documents, structure charts or class diagrams, descriptions of algorithms, and program listings fall in this category. **User documentation** helps users run the program. Such documentation includes online tutorials, answers to frequently asked questions (FAQs), help systems that the user can bring up while the program is running, and perhaps (less and less often) written users' manuals.
9. *Maintenance*—Programs are not static entities that, once completed, never change. Because of the time and expense involved in developing software, successful programs are used for a very long time. It is not unusual for a program to still be in use 20 or 30 years after it was written. (Microsoft Word was first developed and released in 1983!) In fact, the typical life cycle for a medium- to large-sized software package is 1–3 years in development and 5–20 years in the marketplace. During this long period of use, errors may be uncovered, new hardware or system software may be purchased on which the program has to run, user needs may change, and the whims of the marketplace may fluctuate. The original program must be modified and brought out in new versions to meet these changing needs. **Program maintenance**, the process of adapting and improving an existing software product, may consume as much as 65% of the total software development life cycle budget. If the program has been well planned, carefully designed, well coded, thoroughly tested, and well documented, then program

maintenance is a much easier task. Indeed, it is with an eye to program maintenance (and to reducing its cost) that we stress the importance of these earlier steps.

Maintenance should not really be viewed as a separate step in the software development life cycle. Rather, it involves repetition of some or all of the steps previously described, from a feasibility study through implementation, testing, and updated documentation. Maintenance reflects the fact that the software development life cycle is truly a *cycle*, during which it may be necessary to redo earlier phases of development as our software changes, grows, and matures.

9.6.3 Modern Environments

Development Environments. Modern software development environments have had a great impact on the software development life cycle process. Regardless of the programming language used, most programmers now work within an **integrated development environment, or IDE**. The IDE lets the programmer perform a number of tasks within the shell of a single application program, rather than having to use a separate program for each task. Consider some of the system software tasks described in Section 6.2: Use a *text editor* to create a program; use a *file system* to store the program; use a *language translator* to translate the program to machine language; and if the program does not work correctly, use a *debugger* to help locate the errors.

A modern programming IDE provides a text editor, a file manager, a compiler, a linker and loader, and tools for debugging, all within this one integrated piece of software. This can significantly speed up program development.

Many IDEs enable programmers to quickly create sample graphical user interfaces (GUIs) called **prototypes** that can be shown to prospective users in the initial stages of software development. These prototypes do not have any functionality, but they can give users a good idea of what the final software package will look like—much like seeing a small-scale mock-up of a proposed new building before it is actually built. This **rapid prototyping** process allows any misunderstandings or miscommunications between user and programmer to be identified and corrected early in the development process.

Finally, there are software packages that track requirements from the initial specification through the design process to final code, to make sure that nothing gets lost along the way. These packages may also support graphical design of the various program elements, such as classes, and facilitate their translation into code.

Code Repositories. Software development is not a one-time thing. Even a single developer, for anything but a trivial piece of code, makes changes. And huge software projects involve teams of many people working on and interacting with various sections of code. There will be progressive versions as the project develops, hence the need for a **version control system** to manage the project as changes are made. Anyone who wants to make a change must be sure he or she is working on the latest version of the code. The user "checks out" a file from the version control system, edits it, and submits the revised file back to the version control system; in the meantime, while the file is "checked out," no one else can edit it. Alternatively, multiple users may simultaneously change a single file and the version control system warns of potential conflicts when the edited files are "merged." Also, there is a need for past versions to be maintained in some organized way so if some bug is found, it can be traced back through earlier versions to see when it first occurred. The version control system manages all of the details of this **code repository** (all the various versions and details about when, why, and by whom changes were made).

There are many version control systems, including Git, an open-source version control package created in 2005 by Linus Torvalds, the developer of the Linux operating system (who slyly named the system using British slang for a silly or stupid person). Git excels at managing large projects efficiently while being simple to use, even on your own personal machine. GitHub is a cloud-based storage facility for code repositories, which can be designated as private or public. It provides backup for files and, even in private repositories, can allow collaborators to see, use, and change these files, thereby enhancing distributed code development where programmers may be in different locations and different time zones. Public cloud-based code repositories, which anyone can view, use, and change, are particularly useful for open-source code development. As of April 2017, GitHub had almost 20 million users and hosted 57 million code repositories.

Change font sizeMain content

Chapter Contents

9.6.4 Agile Software Development

The software development process described in Section 9.6.2 is also known as the **waterfall model** of software development. Think of a series of waterfalls moving a river down and down through various plateaus. In the software development process, these plateaus would be the feasibility study, problem specification, program design, and so on—all the steps we outlined in Section 9.6.2. This imagery suggests that software development is a completely sequential process, that is, one step must be completed before moving on to the next, and that previous steps are never revisited. The waterfall model was originally developed to describe manufacturing processes, in which a step such as "cut sheet metal" definitely had to be completed (and never revisited) before "weld the seams."

Software development is much more fluid, and in fact there is a lot of wiggle room in the waterfall model. The program design phase may bring out issues that require revisiting the problem specification, the debugging phase obviously requires some amount of recoding, and testing and documentation are done throughout the process. Thus, the "pure" waterfall model of software development is seldom used in favor of a more flexible approach that is instead guided by the waterfall model.

Around 2001, a new software development model known as **agile software development** started to be used. The Merriam-Webster online dictionary defines the word *agile* as "marked by ready ability to move with quick easy grace; having a quick, resourceful and adaptable character." This suggests flexibility, nimbleness, and the ability to adjust to changes. Let's revisit a phrase or two from our discussion of the waterfall model: "The problem specification document expresses the final and complete problem specification and guides the software developers in all subsequent decisions....Once agreed to by the developer and the customer, this document becomes essentially a legal contract describing what the developer promises to provide and what the customer agrees to accept." Sounds like a done deal.

The whole agile software development philosophy is a recognition that problem specifications are never a "done deal," that there are bound to be changes, that the development team must be "agile" in its response to these changes, and that the customer should be involved in working with the development team throughout the design process. Agile software development is actually a whole suite of methods and processes that help to promote this flexible and ready response to meet a shifting and ever-changing target.

One of the principles of agile software development, pair programming has even made its way into some college computer science classes. **Pair programming** involves two programmers (students) at a single workstation. At any given point in time, one is writing code and the other is actively observing. The observer watches each line of code for possible errors but also is thinking about the overall approach, what problems may lie ahead, and possibly spotting improvements that could be made. The roles of the two individuals are switched frequently. The emphasis is on cooperation, not competition. Pair programming, if done well, actually produces better code more quickly than a single programmer, but of course there is the added cost of two people working together. So for any particular project, the relative costs and benefits of pair programming should be weighed.

Software Engineering Failures

It is easy to find numerous instances of software failures, with consequences ranging from temporary inconvenience to huge financial losses, or even to loss of life. Many of these failures can be attributed to a specific coding error, but usually a thorough investigation also reveals a much broader spectrum of errors.

The Mars Exploration Program, begun by NASA (National Aeronautics and Space Administration) in 1993, is a U.S. effort to understand and explore the planet Mars. As part of this program, the Mars Climate Orbiter was launched in December, 1998; its purpose was to study the climate and atmosphere of Mars. However, when the Mars Climate Orbiter entered the Martian atmosphere, it had evidently moved much closer to the planet's surface than its projected entry path; as a result, the propulsion system overheated and the engine burned out. It is believed that the $125 million vehicle was destroyed in the atmosphere.

According to the official NASA report, the "root cause" was an embarrassing mismatch between the English units (pound-secs) output by one software module and used as input to another software module that—by NASA software interface specifications—expected metric units (Newtons-secs). This second software module used the English unit quantities as if they were in metric units , and this discrepancy drove the trajectory down. According to the chairman of the Mars Climate Orbiter Mission Failure Investigation Board, "The failure review board has identified other significant factors that allowed this error to be born, and then let it linger and propagate to the point where it resulted in a major error in our understanding of the spacecraft's path as it approached Mars." The report went on to list the following contributing factors (quoted from the report), among others:
The systems engineering function within the project that is supposed to track and double-check all interconnected aspects of the mission was not robust enough, exacerbated by the first-time handover of a Mars-bound spacecraft from a group that constructed it and launched it to a new, multi-mission operations team.
Some communications channels among project engineering groups were too informal.

The small mission navigation team was oversubscribed and its work did not receive peer review by independent experts.
The process to verify and validate certain engineering requirements and technical interfaces between some project groups, and between the project and its prime mission contractor, was inadequate.
These are failures of the software engineering process that allowed a coding error to go undetected.

On a happier note, NASA's Mars Exploration Rover vehicle Opportunity landed on the surface of Mars on January 24, 2004, with an expected mission lifespan of 90 days. On January 24, 2017, 13 years after landing, Opportunity was still rolling around on Mars, collecting and transmitting scientific data!

9.7Conclusion

In this chapter, we have seen how the use of a high-level language overcomes many of the disadvantages of assembly language programming, creating a more comfortable and useful environment for the programmer. In a high-level language, the programmer need not manage the storage or movement of data values in memory. The programmer can think about the problem at a higher level, use program instructions that are both more powerful and more natural-language-like, and write a program that is much more portable among various hardware platforms. The online language chapters (on Ada, C++, C#, Java, and Python) spell out the mechanisms used by each language to give the programmer these more powerful problem-solving abilities. In this chapter, we've seen two small sample programs in each language plus a brief comparison of some of the features of these languages.

We also discussed the entire software development life cycle, noting that for large, real-world programs, software development must be a managed discipline. Coding is but a small part of the software development process.

The high-level languages we have investigated so far all belong to the procedural language family. In the next chapter, we'll look briefly at several additional procedural languages as well as other languages that take quite a different approach to problem solving.