**Chapter 9**

**Introduction to High-Level Language Programming**

**A Guide to this Instructor’s Manual:**

We have designed this Instructor’s Manual to supplement and enhance your teaching experience through classroom activities and a cohesive chapter summary.

This document is organized chronologically, using the same headings that you see in the textbook. Under the headings you will find: lecture notes that summarize the section, Teaching Tips, Class Discussion Topics, and Additional Projects and Resources. Pay special attention to teaching tips and activities geared towards quizzing your students and enhancing their critical thinking skills.

In addition to this Instructor’s Manual, our Instructor’s Resources also contain PowerPoint Presentations, Test Banks, and other supplements to aid in your teaching experience.

|  |
| --- |
| **At a Glance** |

**Instructor’s Manual Table of Contents**

* Overview
* Learning Objectives
* Teaching Tips and Quick Quizzes
* Class Discussion Topics
* Additional Projects
* Additional Resources
* Key Terms
* Solutions to End-of-Chapter Exercises

|  |
| --- |
| Lecture Notes |

**Overview**

Chapter 9 introduces the concept of high-level programming languages. These languages are much closer to pseudocode and allow the programmer to ignore details about moving values around in memory. Code produced in high-level languages can be run on multiple computers using compilers or interpreters. Procedural languages share a common philosophy but differ in the details of syntax and semantics. Ada, C++, C#, Java, and Python are all examples of procedural languages. The chapter illustrates the similarities and differences through two detailed examples and provides a table that compares the five languages for a wide range of language features. The chapter introduces the software development life cycle needed to make large-scale software development feasible. It introduces the stages of the life cycle and explains what is done for each part. It describes the waterfall model of development and compares it to agile software development.

# **Learning Objectives**

* 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

# **Teaching Tips**

**9.1 The Language Progression**

1. Note the existence of online chapters that focus on each of the languages discussed in this chapter. Think about choosing one or more of these chapters to augment the material here. It is also important to decide from the start if you are going to use all five languages or choose one or two to discuss.
2. Discuss the motivation for developing high-level programming languages, based on the need for easier programming, and assembly language’s disadvantages. These include manual management of data movement, breaking each task into tiny subtasks, machine specific code, and statements far from natural language (or pseudocode).
3. We expect the following of high-level programming languages: the programmer does not need to manage the movement of data; the programmer can take a macroscopic view of tasks; programs are portable from machine to machine; and programming statements are closer to natural language and use standard mathematical notation. Introduce the term **third-generation languages** as an alternative description of high-level languages.
4. Translating high-level languages to machine code (remind them of the term **object code**) is more complex. Discuss how compilers work, translating **source code** to either assembly or an assembly-like language, and the use of assemblers and the **linker** to integrate **code libraries** into the **executable module**.
5. Show students Figure 9.1 as an example of how the code gets back to the machine.

**Quick Quiz 1**

1. (True or false) A compiler translates high-level source code into assembly language.

Answer: True

1. Name two disadvantages of assembly language that high-level languages are meant to overcome.

Answer: The programmer has to write instructions that control the movement of data among memory and registers, the programmer has to break tasks into tiny subtasks, code written in assembly language cannot be used on other types of machines, or instructions are very far from natural language or normal mathematical notation.

1. The resulting object code created by the linker is often called the \_\_\_\_\_\_\_.

Answer: executable module

**9.2 A Family of Languages**

1. Introduce the term **procedural languages** (**imperative languages**), and describe its underlying philosophy about describing algorithms. Talk about the differences in **syntax** and **semantics** in programming languages, while pointing out their similar approach.
2. Note the five languages used as examples are Ada, C++, C#, Java, and Python. If you are going to spend time introducing one of these languages in more detail, using the online chapters, this is a good place to do that.

**9.3 Two Examples in Five-Part Harmony**

1. Go over the pseudocode description of the favorite number algorithm. Go over the examples in each of the figures (9.3 through 9.7) and discuss the differences with students. Make a list of things to look at, drawing both from the pseudocode they have already seen and the features of assembly language. Items on the list might include: comments, creating variables, arithmetic operations, input and output, and marking the beginning and ending of blocks. Emphasize that students are not expected to understand every bit of the programs.

|  |  |
| --- | --- |
| ***Teaching Tip*** | It is helpful for students to see the same code in two different languages side by side. If you have the option to project two different screens, you can put up two programs at once and mark the same feature in each (input operations, for example). Otherwise, consider an activity where students are given pairs of programs on paper and asked to mark the same features. |

1. Remind students of the data cleanup algorithm. Because it is more complex, take time to work through examples again. Use the pseudocode as the basis for noting things to look for in the actual code: control structures like the while loop and if statements, how the n data items are represented, and how references to “the item at position left” become code.
2. The programs are longer; consider dividing the programs into sections that have a common purpose, like “setting up data,” “getting input values,” and so forth.

**9.4 Feature Analysis**

1. The content for this section exists primarily in Figure 9.15. The table covers six pages. Every pair of pages covers one set of language features: three example languages on one page and two on the next. The first two sets of features pertain mostly to basic syntax; the last set describes deeper features such as scope, parameter passing, and object-oriented programming.
2. Pick and choose carefully those features from the table that you want to talk about. Then ask students to look back at the code examples from the previous section and generate the values from the table themselves rather than simply presenting the material from the table.

**9.5 Meeting Expectations**

1. This section revisits the expectations introduced at the start of the chapter and examines how these example languages meet each expectation.
2. *The programmer need not manage the details of the movement of data*: Point out how high-level languages treat variables, both declaration and assignment statements, and arithmetic.
3. *The programmer can take a macroscopic view of tasks*: Point out how similar to pseudocode these languages are. Arithmetic uses standard notation (for the most part), and loops and conditionals are very far from assembly language.
4. *Programs written in a high-level language will be portable rather than machine specific*: Discuss the variety of translation models that permit high-level languages to be portable. These include a separate compiler for each machine that translates to that machine’s assembly, a single compiler that translates to a machine-independent bytecode that is then either interpreted or compiled by a small program on the user’s machine, or an interpreter designed for the user’s machine.
5. *Programming statements in a high-level language will be closer to natural language and use standard mathematical notation*: It is fairly obvious that high-level languages succeed in that, subject to the constraint that the programs must be able to be compiled.

**9.6 The Big Picture: Software Engineering**

1. Introduce the term **software development life cycle**. Go over the breakdown of time spent in various phases of development, and discuss the difference in scale between the kinds of programs students have seen so far (both pseudocode and the actual code in the earlier section) and real-world professional software. Introduce the term **software engineering**, and explain why it is used in the professional world. Note that students who major in computer science can go on to become software engineers.

|  |  |
| --- | --- |
| ***Teaching Tip*** | Consider picking a single, small example to illustrate the software development life cycle. With the students, work through each phase, showing what the various documents would look like for this example. |

1. Introduce the term **feasibility study**, and use a concrete example to illustrate the cost–benefit analysis of such a study. Note the product of this phase: a feasibility document.
2. Introduce the term **problem specification**, and emphasize the detailed nature of the document it produces. Talk in detail about data, inputs, and error handling.
3. Introduce the terms **program design phase** and **divide-and-conquer** strategy or **top-down decomposition**. Illustrate the design process with a small example. Note the product of this phase: a program design document. In an object-oriented programming approach objects are identified, together with their data and subtasks they perform.
4. Briefly touch on the term **object-oriented programming**, and discuss again the idea that it is based around identified objects, their data, and the subtasks they must perform.
5. Introduce the term **coding** for writing the program from the design. Introduce the term **debugging** for the initial testing to make the code correct. Describe the three types of errors: **syntax errors**, **runtime errors**, and **logic errors**.
6. Introduce the term **empirical testing** and the idea of a **test suite** for the formal testing that follows each code writer’s individual debugging. Emphasize the creation and storage of sets of tests that are intended to thoroughly exercise a program. Testing occurs at multiple levels: **unit testing**, **integration testing**, and **regression testing**. Introduce the term **program verification**, and describe its use for critical systems like medical equipment or airplane controls. Introduce the term **benchmarking,** and emphasize that its main purpose is to check the time taken by various parts of the program.
7. Discuss the wide range of documentation that provides a written record of the development of a project (note the importance of clear writing for software developers). Introduce the terms **internal documentation**, **external documentation**, **technical documentation**, and **user documentation**.
8. Introduce the term **program maintenance**, and describe the activities that fall within it, including fixing errors, updating for new hardware or operating systems, and adding new features based on user demand.
9. Describe the need for tools to support software development. Introduce the term **integrated development environment (IDE)**, and describe the various programs that make up a typical IDE. Introduce the term **prototypes**; many IDES support the creation of GUI prototypes. Describe tools for version control and document tracking.
10. Introduce the idea of the **integrated development environment**, or **IDE**. Discuss the benefits of using an IDE to code multiple aspects of a program. Discuss how they can help with **protoypes**, and go over **rapid prototyping** as a way to help alleviate miscommunications between the user and the programmer early in development.
11. Introduce **version control systems** and how **code repositories** can help with versioning of software as new updates are coded and released.
12. Compare the **waterfall model**, described previously, with the new **agile software development** approach. Note the key difference is the emphasis on many passes through the sequence of specification, design, coding, and testing, with small increments made at each point. Note that **pair programming** is often used even with beginning programmers.

**9.7 Conclusion**

1. Go over the various topics covered in this chapter with the students. If you focused on a couple of programming languages, discuss with them which one they preferred and why.
2. If necessary, briefly go over the software life cycle again.
3. Talk with students about the benefits of using high-level programming and what advances may still be in store later down the line.

# **Class Discussion Topics**

1. What features of the different languages you’ve seen do you like best? Are all those features in the same language? Discuss why there are so many different languages, all with the same underlying approach. Why isn’t there just one “right” language?
2. Compare the software development process with processes suggested for writing papers. Are there parallels; are there significant differences?
3. If you were designing a language and wanted it to be more like natural language, how would you choose to describe conditional statements and loops? How would your form differ from those of the languages you’ve studied?

# **Additional Projects**

1. Take a sample program (provided by your instructor) and try to convert it into the kind of pseudocode used in your textbook. What parts of the program are unnecessary when describing an algorithm in pseudocode? What parts stay more or less the same?
2. Given a problem to solve (provided by your instructor), work through the waterfall model of the software development life cycle. Perform an informal feasibility study, producing a feasibility document a paragraph or two long. Then refine the problem statement to be a problem specification document. Design pseudocode for your program design document. If you have been learning to program in a particular language, then complete the process by coding and testing your program.
3. Take programs that solve the same problem but written in two different languages described in this chapter. For every 2–3 lines of each program, list the lines that correspond in the other program. Are there any lines that have no corresponding lines?
4. If you worked through one of the online chapters for a particular programming language, briefly look over one of the other languages and see if it appears more or less complex than the one you initially studied. With just this brief glimpse into the new language, which one do you prefer and why?

# **Additional Resources**

1. A website describing the philosophy and approach of agile software development: <http://www.agile-process.org/>
2. An article comparing C# and Java: <http://msdn.microsoft.com/en-us/library/ms836794.aspx>
3. A tutorial about why and how version control systems work: <http://betterexplained.com/articles/a-visual-guide-to-version-control/>
4. A quick comparison between C++ and C#: https://msdn.microsoft.com/en-us/library/yyaad03b(v=vs.71).aspx

**Key Terms**

* **Agile software development**: An approach to software development that emphasizes a flexible and ready response to meet a shifting target.
* **Code library**: A collection of thoroughly tested object code for various useful tasks.
* **Code repository**: All the various versions and details about when, why, and by whom changes were made to a program.
* **Coding**: The process of translating the detailed designs into computer code.
* **Debugging**: The process of locating and correcting program errors.
* **Divide-and-conquer strategy**: A program design strategy in which 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 total job.
* **Empirical testing**: Designing a special set of test cases and running the program using these test data.
* **Executable module**: The resulting object code after a linker inserts requested code from code libraries.
* **External documentation**: Any materials assembled to clarify the program’s design and implementation.
* **Feasibility study**: A step in the software development life cycle that evaluates a proposed project and compares the costs and benefits of various solutions.
* **Imperative language**: Same as procedural language.
* **Integrated development environment (IDE)**: A collection of programs that support software development, such as debuggers, editors, toolkits, and libraries, that lets programmers perform several tasks within the shell of a single application.
* **Integration testing**: After unit testing, integration testing is done to see that the modules communicate the necessary data between and among themselves and that all modules work together smoothly.
* **Internal documentation**: Documentation that is part of the program code itself.
* **Linker**: A piece of system software that inserts requested object code from code libraries into the object code for the requesting program.
* **Logic error**: An error in the algorithm used to solve a problem.
* **Object code**: Machine language instructions.
* **Object-oriented programming**: A programming language that uses identified objects, together with their data and the subtasks they must perform.
* **Open source programming language**: A programming language that is open to the world to help develop and build.
* **Pair programming**: Involves two programmers at a single workstation. At any given point in time, one is writing code and the other is actively observing, watching for possible errors but also thinking about the overall approach.
* **Problem specification**: A step in the software development life cycle that involves developing a clear, concise, and unambiguous statement of the exact problem the software is to solve.
* **Procedural language**: A program written in a procedural language consists of sequences of statements that manipulate data items.
* **Program design phase**: A step in the software development life cycle that plans the structure of the software to be written.
* **Program maintenance**: The process of adapting an existing software product due to errors, new system requirements, or changing user needs.
* **Program verification**: Used to prove that if the input data to a program satisfies certain conditions, then, after the program has been run on these data, the output data satisfies certain other conditions.
* **Prototype**: Sample graphical user interfaces (GUIs) created early in the software development process in order to get user input.
* **Rapid prototyping**: The fast creation of prototypes using IDEs.
* **Regression testing**: If anything is changed on an already-tested module, regression testing is done to be sure that this change has not introduced a new error into code that was previously correct.
* **Runtime error**: An error that occurs when the program is run using certain sets of data that result in some illegal operation, such as dividing by zero.
* **Semantics**: The meaning of correctly written programming statements.
* **Software development life cycle**: The overall sequence of steps needed to complete a large-scale software project.
* **Software engineering**: Large-scale software development.
* **Source code**: High-level language instructions.
* **Syntax**: The rules for exactly how programming statements must be written; the grammatical structure of a programming language.
* **Syntax error**: An error that occurs because a program statement fails to follow the correct rules of syntax.
* **Technical documentation**: Documentation that enables programmers who later have to modify the program to understand the code.
* **Test suite**: A special set of test cases designed to run a program for the purpose of empirical testing.
* **Third-generation language**: Another name for high-level programming language as opposed to machine language (first generation) or assembly language (second generation).
* **Top-down decomposition**: A program design strategy in which 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 total job.
* **Unit testing**: Testing each module of code as it is completed.
* **User documentation**: Documentation that helps users run the program.
* **Version control system**: A method of managing a project as changes are made.
* **Waterfall model**: The traditional model of software development that follows a series of steps.

**Solutions to End-of-Chapter Exercises**

Each online language module has its own set of exercises. Chapter 9 in the text itself contains only Challenge Exercises.

**Challenge Work**

**1.** Students have the code in various languages for the array declaration and the input process from Section 9.3.2. After that it should not be too difficult to implement the pseudocode algorithm of Figure 3.1 in the chosen language.

**2.** This is a much more challenging problem. A C++ solution appears below. Other versions will be more or less similar.

//program to illustrate Caesar cypher

#include <iostream>

using namespace std;

void Encode(char message[], int max, int s)

{

int i; //counter for array position

i = 0;

while (i <= max)

{

message[i] = message[i] + s;

if (message[i] > 'Z')

{

message[i] = message[i] - 26;

}

i = i + 1;

}

}

void Decode(char message[], int max, int s)

{

int i; //counter for array position

i = 0;

while (i <= max)

{

message[i] = message[i] - s;

if (message[i] < 'A')

{

message[i] = message[i] + 26;

}

i = i + 1;

}

}

void WriteMessage(char message[], int max)

{

for (int i = 0; i <= max; i++)

{

cout << message[i];

}

cout << endl;

}

void main()

{

char message[10]; //array to store message word

char next; //next character read in

int i; //counter for array position

int max; //maximum array element used

int s; //shift amount

cout << "Enter your message. Use all uppercase " << endl;

cout << "letters, maximum 10; end with %" << endl;

i = 0;

cin >> next;

while (next != '%')

{

message[i] = next;

cin >> next;

i = i + 1;

}

if (i > 10) //message was too long

{

cout << "Message too long, run the program again" << endl;

}

else

{

max = i - 1;

cout << "The original message is " << endl;

WriteMessage(message, max);

cout << endl;

cout << "What is the shift amount for the cypher?" << endl;

cin >> s;

cout << endl;

Encode(message, max, s);

cout << "The encoded message is " << endl;

WriteMessage(message, max);

cout << endl;

Decode(message, max, s);

cout << "The decoded message is " << endl;

WriteMessage(message, max);

cout << endl;

}

}

**3.**  A major task is collecting the data from the user and filling the 2-D array. Each language has its own way to declare a 2-D array. For Floyd's algorithm itself, students should be able to translate the given pseudocode to their choice of language. The length of the shortest path from 2 to 4 is 4, using the path 2 to 3 to 4.

**4.** and **5**. The details of file I/O are language-specific, and students will need some help here. After that, they should be able to adapt their solution to Exercise 1 fairly easily. Code for Exercise 5 will be a bit more challenging.

**6.** Any links to get students started on any of these topics would be valuable.