1 PROBLEM SOLVING, ABSTRACTION, AND STEPWISE REFINEMENT

INTRODUCTION

Programming reduces to the art and science of problem solving. To be a good programmer, you must be a good problem solver. To be a good problem solver, you must attack a problem in a methodical way, from initial problem inspection and definition to final solution, testing, and documentation. In the beginning, when confronted with a programming problem, you will be tempted to get to the computer and start coding as soon as you get an idea of how to solve it. However, you *must* resist this temptation. Such an approach might work for simple problems, but will *not* work when you are confronted with the complex problems found in today's real world. A good carpenter might attempt to build a dog house without a plan, but would never attempt to build your "dream house" without a good set of blueprints.

In this chapter, you will learn about a systematic method that will make you a good problem solver, and therefore a good programmer—I call it the ***programmer's algorithm***. In particular, you will study the steps required to solve just about any programming problem using a *top-down structured* approach. You will be introduced to the concept of ***abstraction***, which allows problems to be viewed in general terms without agonizing over the implementation details required by a computer language. From an initial abstract solution, you will refine the solution step-by-step until it reaches a level that can be coded directly into a computer program. Make sure you understand this material and work the related problems at the end of the chapter. As you become more experienced in programming, you will find that the "secret" to successful programming is good planning through abstract analysis and stepwise refinement, which results in top-down structured software designs. Such designs are supported by languages like C++. In the chapters that follow, you will build on this knowledge to create workable C++ programs.

1-1 THE PROGRAMMER'S ALGORITHM

Before we look at the programmer's algorithm, it might be helpful to define what is meant by an algorithm. In technical terms, it is as follows:

> An ***algorithm*** is a series of step-by-step instructions that produces a solution to a problem.

Algorithms are not unique to the computer industry. Any set of instructions, such as those you might find in a recipe or a kit assembly guide, can be considered an algorithm. The programmer's algorithm is a recipe for you, the programmer, to follow when developing programs. The algorithm is as follows:

THE PROGRAMMER'S ALGORITHM

- Define the problem.
- Plan the problem solution.
- Code the program.
- Test and debug the program.
- Document the program.

Defining the Problem

You might suggest that defining the problem is an obvious step in solving any problem. However, it often is the most overlooked step, especially in computer programming. The lack of good problem definition often results in "spinning your wheels," especially in more complex computer programming applications.

Think of a typical computer programming problem, such as controlling the inventory of a large department store. What must be considered as part of the problem definition? The first consideration probably is what you want to get out of the system. Will the output information consist of printed inventory reports or, in addition, will the system automatically generate product orders based on sales? Must any information generated by a customer transaction be saved permanently on disk, or can it be discarded? What type of data is the output information to consist of? Is it numerical data, character data, or both? How must the output data be formatted? All of these questions must be answered in order to define the output requirements.

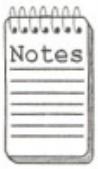
Careful consideration of the output requirements usually leads to deciding what must be put into the system in order to obtain the desired system output. For instance, in our department store inventory example, a desired output would be most likely a summary of customer transactions. How are these transactions to be entered into the system? Is the data to be obtained from a keyboard, or is product information to be entered automatically via an optical character recognition (OCR) system that reads the bar code on the product price tags? Does the input

consist of all numerical data, character data, or a combination of both? What is the format of the data?

The next consideration is processing. Will most of the customer processing be done at the cash register terminal, or will it be handled by a central store computer? What about credit card verification and inventory records? Will this processing be done by a local microcomputer, a minicomputer located within the store, or a central mainframe computer located in a different part of the country? What kind of programs will be written to do the processing, and who will write them? What sort of calculations and decisions must be made on the data within individual programs to achieve the desired output?

All of these questions must be answered when defining any computer programming problem. In summary, you could say that problem definition must consider the application requirements of output, input, and processing. The department store inventory problem clearly requires precise definition. However, even with small application programs, you must still consider the type of output, input, and processing that the problem requires.

When defining a problem, look for the nouns and verbs within a problem statement. The nouns often suggest input and output information, and the verbs suggest processing steps. The application will always dictate the problem definition. I will discuss problem definition further, as we begin to develop computer programs to solve real problems.



PROBLEM-SOLVING TIP

Look for the nouns and verbs within a problem statement; they often provide clues to the required output, input, and processing. The nouns suggest output and input, and the verbs suggest processing steps.

Planning the Solution

The planning stage associated with any problem is probably the most important part of the solution, and computer programming is no exception. Imagine trying to build a house without a good set of blueprints. The results could be catastrophic! The same is true of trying to develop computer software without a good plan. When developing computer software, the planning stage is implemented using a collection of algorithms. As you already know, an algorithm is a series of step-by-step instructions that produce results to solve problems. When planning computer programs, algorithms are used to outline the solution steps using English-like

statements, called ***pseudocode***, that require less precision than a formal programming language. A good pseudocode algorithm should be independent of, but easily translated into, *any* formal programming language.

> ***Pseudocode*** is an informal set of English-like statements that are generally accepted within the computer industry to denote common computer programming operations. Pseudocode statements are used to describe the steps in a computer algorithm.

Coding the Program

Coding the program should be one of the simplest tasks in the whole programming process, provided you have done a good job of defining the problem and planning its solution. Coding involves the actual writing of the program in a formal programming language. The computer language you use will be determined by the nature of the problem, the programming languages available to you, and the limits of the computer system. Once a language is chosen, the program is written, or coded, by translating your algorithm steps into the formal language code.

I should caution you, however, that coding is really a mechanical process and should be considered secondary to algorithm development. In the future, computers will generate their own program code from well-constructed algorithms. Research in the field of artificial intelligence has resulted in "code-generation" software. The thing to keep in mind is that computers might someday generate their own programming code from algorithms, but it takes the creativity and common sense of a human being to plan the solution and develop the algorithm.

Testing and Debugging the Program

You will soon find out that it is a rare and joyous occasion when a coded program actually "runs" the first time without any errors. Of course, good problem definition and planning will avoid many program mistakes, or "bugs." However, there always are a few bugs that manage to go undetected, regardless of how much planning you do. Getting rid of the program bugs (*debugging*) often is the most time-consuming job in the whole programming process. Industrial statistics show that often over 50 percent of a programmer's time is often spent on program debugging.

There is no absolute correct procedure for debugging a program, but a systematic approach can help make the process easier. The basic steps of debugging are as follows:

- Realizing that you have an error.
- Locating and determining the cause of the error.
- Fixing the error.

First of all, you have to realize that you have an error. Sometimes, this is obvious when your computer freezes up or crashes. At other times, the program might work fine until certain unexpected information is entered by someone using the program. The most subtle errors occur when the program is running fine and the results look correct, but when you examine the results closely, they are not quite right.

The next step in the debugging process is locating and determining the cause of the errors—sometimes the most difficult part of debugging. This is where a good programming tool, called a ***debugger***, comes into play.

Fixing the error is the final step in debugging. Your knowledge of the C++ language, this book, C++ on-line help, a C++ debugger, and your C++ reference manuals are all valuable tools in fixing the error and removing the "bug" from your program.

When programming in C++, there are four things that you can do to test and debug your program: ***desk-check*** the program, ***compile/link*** the program, ***run*** the program, and ***debug*** the program.

Desk-Checking the Program

Desk-checking a program is similar to proofreading a letter or manuscript. The idea is to trace through the program mentally to make sure that the program logic is workable. You must consider various input possibilities, and write down any results generated during program execution. In particular, try to determine what the program will do with unusual data by considering input possibilities that "shouldn't" happen. Always keep Murphy's law in mind when desk-checking a program: If a given condition can't or shouldn't happen, it will!

For example, suppose a program requires the user to enter a value whose square root must be found. Of course, the user "shouldn't" enter a negative value, because the square root of a negative number is imaginary. However, what will the program do if he or she does? Another input possibility that should always be

considered is an input of zero, especially when used as part of an arithmetic operation, such as division.

When you first begin programming, you will be tempted to skip the desk-checking phase, because you can't wait to run the program once it is written. However, as you gain experience, you soon will realize the time-saving value of desk-checking.

Compiling and Linking the Program

At this point, you are ready to enter the program into the computer system. Once entered, the program must be compiled, or translated, into machine code. Fortunately, the compiler is designed to check for certain program errors. These usually are ***syntax*** errors that you have made when coding the program. A syntax error is any violation of the rules of the programming language, such as using a period instead of a semicolon. There might also be type errors. A ***type error*** occurs when you attempt to mix different types of data, such as numeric and character data. It is like trying to add apples to oranges.

> A ***syntax error*** is any violation of the rules of the programming language, and a ***type error*** occurs when you attempt to mix different types of data.

During the compiling process, many C++ compilers will generate error and warning messages as well as position the display monitor cursor to the point in the program where the error was detected. The program will not compile beyond the point of the error until it is corrected. Once an error is corrected, you must attempt to compile the program again. If other errors are detected, you must correct them, recompile the program, and so on, until the entire program is successfully compiled.

After the program is successfully compiled, it must be linked to other routines that might be required for its execution. Linking errors will occur when such routines are not available or cannot be located in the designated system directory.

> A ***link*** error will occur when any required routines cannot be located by the compiler.

Running the Program

Once the program has been compiled and linked, you must execute, or run, it. However, just because the program has been compiled and linked successfully doesn't mean that it will run successfully under all possible conditions. Common bugs that occur at this stage include ***logic errors*** and ***run-time*** errors. These are the most difficult kinds of errors to detect. A logic error will occur when a loop tells the computer to repeat an operation, but does not tell it when to stop repeating. This is called an ***infinite loop***. Such a bug will not cause an error message to be generated, because the computer is simply doing what it was told to do. The program execution must be stopped and debugged before it can run successfully.

> A ***logic error*** occurs when the compiler does what you tell it to do, but is not doing what you meant it to do. A ***run-time*** error occurs when the program attempts to perform an illegal operation as defined by the laws of mathematics or the particular compiler in use.

A run-time error occurs when the program attempts to perform an illegal operation, as defined by the laws of mathematics or the particular compiler in use. Two common mathematical run-time errors are division by zero and attempting to take the square root of a negative number. A common error imposed by the compiler is an integer value out of range. Most C++ compilers limit integers to a range of –32,768 to +32,767. Unpredictable results can occur if an integer value exceeds this range.

Sometimes, the program is automatically aborted and an error message is displayed when a run-time error occurs. Other times, the program seems to execute properly, but generates incorrect results, commonly called *garbage*. Again, you should consult your compiler reference manual to determine the exact nature of the problem. The error must be located and corrected before another attempt is made to run the program.

Using a Debugger

One of the most important programming tools that you can have is a debugger. A debugger provides a microscopic view of what is going on in your program. Many C++ compilers include a built-in, or *integrated*, debugger that allows you to

single-step program statements and view the execution results in the CPU and memory.

DEBUGGING TIP

A word from experience: Always go about debugging your programs in a systematic, commonsense manner. Don't be tempted to change something just because you "hope" it will work and don't know what else to do. Use your resources to isolate and correct the problem. Such resources include your algorithm, a program listing, your integrated C++ debugger, your reference manuals, this textbook, and your instructor, just to mention a few. Logic and run-time errors usually are the result of a serious flaw in your program. They will not go away and cannot be corrected by blindly making changes to your program. One good way to locate errors is to have your program print out preliminary results as well as messages that tell when a particular part of the program is running.

Documentation

This final step in the programmer's algorithm often is overlooked, but it probably is one of the more important steps, especially in commercial programming. Documentation is easy if you have done a good job of defining the problem, planning the solution, coding, testing, and debugging the program. The final program documentation is simply the recorded result of these programming steps. At a minimum, good documentation should include the following:

- A narrative description of the problem definition, which includes the type of input, output, and processing employed by the program.
- An algorithm.
- A program listing that includes a clear commenting scheme. Commenting within the program is an important part of the overall documentation process. Each program should include comments at the beginning to explain what it does, any special algorithms that are employed, and a summary of the problem definition. In addition, the name of the programmer and the date the program was written and last modified should be included.
- Samples of input and output data.

- Testing and debugging results.
- User instructions.

The documentation must be neat and well-organized. It must be easily understood by you as well as any other person who might have a need to use or modify your program in the future. What good is an ingenious program if no one can determine what it does, how to use it, or how to maintain it?

One final point: Documentation should always be an ongoing process. Whenever you work with the program or modify it, make sure the documentation is updated to reflect your experiences and modifications.

Quick Check

1. English-like statements that require less precision than a formal programming language are called _____.
2. What questions must be answered when defining a computer programming problem?
3. What can be done to test and debug a program?
4. Why is commenting important within a program?

1-2 PROBLEM SOLVING USING ALGORITHMS

In the previous section, you learned that an algorithm is a sequence of step-by-step instructions that will produce a solution to a problem.

For instance, consider the following series of instructions:

Apply to wet hair.
Gently massage lather through hair.
Rinse, keeping lather out of eyes.
Repeat.

Look familiar? Of course, this is a series of instructions that might be found on the back of a shampoo bottle. But does it fit the technical definition of an

algorithm? In other words, does it produce a result? You might say "yes," but look closer. The algorithm requires that you keep repeating the procedure an infinite number of times, so theoretically you would never stop shampooing your hair! A good computer algorithm must terminate in a finite amount of time. The repeat instruction could be altered easily to make the shampooing algorithm technically correct:

Repeat until hair is clean.

Now the shampooing process can be terminated. Of course, you must be the one to decide when your hair is clean.

The foregoing shampoo analogy might seem a bit trivial. You probably are thinking that any intelligent person would not keep on repeating the shampooing process an infinite number of times, right? This obviously is the case when we humans are executing the algorithm, because we have some commonsense judgment. But what about a computer? Most computers do exactly what they are told to do via the computer program. As a result, a computer would repeat the original shampooing algorithm over and over an infinite number of times. This is why the algorithms that you write for computer programs must be precise.

Now, let's develop an algorithm for a process that is common to all of us—mailing a letter. Think of the steps that are involved in this simple process. You must first address an envelope, fold the letter, insert the letter in the envelope, and seal the envelope. Next, you need a stamp. If you don't have a stamp, you have to buy one. Once a stamp is obtained, you must place it on the envelope and mail the letter. The following algorithm summarizes the steps in this process:

Obtain an envelope.
Address the envelope.
Fold the letter.
Insert the letter in the envelope.
Seal the envelope.
If you don't have a stamp, then buy one.
Place the stamp on the envelope.
Mail the letter.

Does this sequence of instructions fit our definition of a good algorithm? In other words, does the sequence of instructions produce a result in a finite amount of time? Yes, assuming that each operation can be understood and carried out by the person mailing the letter. This brings up two additional characteristics of good

algorithms: Each operation within the algorithm must be ***well-defined*** and ***effective***. By well-defined, I mean that each of the steps must be clearly understood by people in the computer industry. By effective, I mean that some means must exist in order to carry out the operation. In other words, the person mailing the letter must be able to perform each of the algorithm steps. In the case of a computer program algorithm, the compiler must have the means of executing each operation in the algorithm.

In summary, a good computer algorithm must possess the following three attributes:

1. Employ well-defined instructions that are generally understood by people in the computer industry.
2. Employ instructions that can be carried out effectively by the compiler executing the algorithm.
3. Produce a solution to the problem in a finite amount of time.

In order to write computer program algorithms, we need to establish a set of well-defined, effective operations. The set of pseudocode operations listed in Table 1-1 will make up our algorithmic language. We will use these operations from now on, whenever we write computer algorithms.

TABLE 1-1 PSEUDOCODE OPERATIONS USED IN THIS TEXT

Sequence	**Decision**	**Iteration**
Add (+)	If/Then	While
Calculate	If/Else	Do/While
Decrement	Switch/Case	For
Divide (/)		
Increment		
Multiply (*)		
Print		
Read		
Set or assign (=)		
Square		
Subtract (−)		
Write		

Notice that the operations in Table 1-1 are grouped into three major categories: ***sequence***, ***decision***, and ***iteration***. These categories are called ***control structures***. The sequence control structure includes those operations that produce a single action or result. Only a partial list of sequence operations is provided here. This list will be expanded as additional operations are needed. As its name implies, the decision control structure includes the operations that allow the computer to make decisions. Finally, the iteration control structure includes those operations that are used for looping, or repeating, operations within the algorithm. Many of the operations listed in Table 1-1 are self-explanatory. Those that are not will be discussed in detail as we begin to develop more complex algorithms.

Quick Check

1. Why is it important to develop an algorithm prior to coding a program?
2. What are the three major categories of algorithmic-language operations?
3. List three decision operations.
4. List three iteration operations.

1-3 PROBLEM ABSTRACTION AND STEPWISE REFINEMENT

At this time, we need to introduce a very important concept in programming—***abstraction***. Abstraction allows us to view a problem in general terms, without worrying about the details. As a result, abstraction provides for generalization in problem solving.

> ***Abstraction*** provides for generalization in problem solving by allowing you to view a problem in general terms, without worrying about the details of the problem solution.

You might have heard the old saying "You can't see the forest for the trees." This means that it is very easy to get lost within the trees of the forest without seeing the big picture of the entire forest. This saying also applies to problem

solving and programming. When starting out to solve a program, you need to get the "big picture" first. Once you have the big picture of the problem solution, the "forest," you can gradually *refine* the solution by providing more detail until you have a solution, the "trees," that is easily coded in a computer language. The process of gradually adding more detail to a general problem solution is called ***stepwise refinement***.

> ***Stepwise refinement*** is the process of gradually adding detail to a general problem solution until it can be easily coded in a computer language.

As an example, consider the problem of designing your own "dream house." Would you begin by drafting out the detailed plans of the house? Not likely, because you would most likely get lost in detail. A better approach would be to first make a general perspective artist's rendition of the house. Then make a general floor plan diagram, and finally make detailed drawings of the house construction that could be followed by the builders.

The concepts of problem abstraction and stepwise refinement allow you to *divide-and-conquer* the problem and solve it from the *top down*. This strategy has been proven to conquer all types of problems, especially programming problems. In programming, we generate a general problem solution, or algorithm, and gradually refine it, producing more detailed algorithms, until we get to a level that can be easily coded using a programming language. This idea is illustrated in Figure 1-1.

In summary, when attacking a problem, always start with the big picture and begin with a very general, or *abstract*, model of the solution. This allows you to concentrate on the problem at hand without getting lost in the "trees" by worrying about the implementation details of a particular programming language. You then gradually refine the solution until you reach a level that can be easily coded using a structured programming language, like C++. The *Problem Solving in Action* examples that follow illustrate this process.

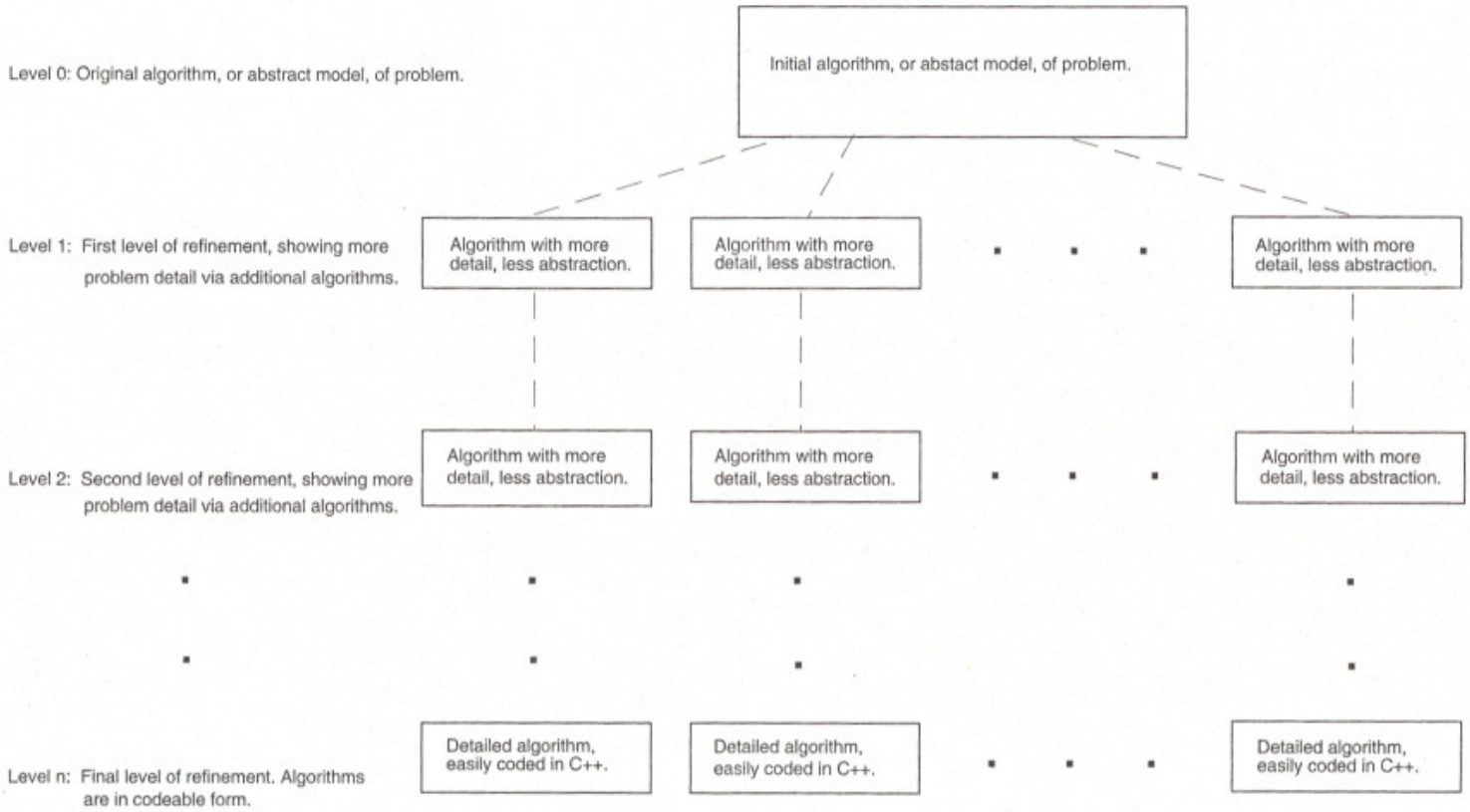


Figure 1-1 Problem solution begins with general, abstract model of the problem, which is stepwise refined, producing more and more detail, until a codeable level of algorithms is reached.

Quick Check

1. Explain why abstraction is important when solving problems.
2. Explain the process of stepwise refinement.
3. How do you know when you have reached a codeable level of an algorithm when using stepwise refinement?

PROBLEM SOLVING IN ACTION: PYTHAGOREAN THEOREM

Problem

Develop a set of algorithms to find the hypotenuse of a right triangle, given its two right angle sides, using the Pythagorean theorem depicted in Figure 1-2. Construct the final algorithms using the algorithmic instructions listed in Table 1-1.

$$H^2 = A^2 + B^2$$

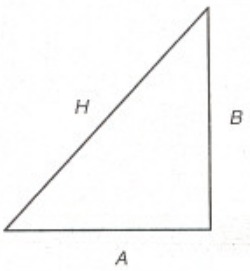


Figure 1-2 Solving the hypotenuse of a right triangle using the Pythagorean theorem.

Defining the Problem

When defining the problem, you must consider three things: *output, input*, and *processing* as related to the problem statement. Let's label the two sides A and B, and the hypotenuse H. The problem requires us to find the hypotenuse (H), given the two sides (A and B). So the output must be the hypotenuse (H). We will display the hypotenuse value on the system monitor. In order to obtain this output,

the two sides (A and B) must be received by the program. Let's assume that the user must enter these values via the system keyboard.

The Pythagorean theorem states that the hypotenuse squared is equal to the sum of the squares of the two sides. In symbols,

$$H^2 = A^2 + B^2$$

This equation represents the processing that must be performed by the computer. In summary, the problem definition is as follows:

Output: The hypotenuse (H) of a right triangle displayed on the system monitor.

Input: The two sides (A and B) of a right triangle to be entered by the user via the system keyboard.

Processing: Employ the Pythagorean theorem: $H^2 = A^2 + B^2$.

Now that the problem has been defined in terms of output, input, and processing, it is time to plan the solution by developing the required algorithms.

Planning the Solution

We will begin with an abstract model of the problem. This will be our initial algorithm, which we will refer to as *main()*. At this level, we are only concerned about addressing the major operations required to solve the problem. These are derived directly from the problem definition. As a result, our initial algorithm is as follows:

Initial Algorithm

```
main()
BEGIN
   Obtain the two sides (A and B) of a right triangle from the user.
   Calculate the hypotenuse of the triangle using the Pythagorean theorem.
   Display the results on the system monitor.
END.
```

Notice that, at this level, you are not concerned with how to perform the foregoing operations in a computer language. You are only concerned about the major program operations, without agonizing over the language implementation details. This is problem abstraction!

The next step is to refine repeatedly the initial algorithm until we obtain one or more algorithms that can be coded directly in a computer language. This is a relatively simple problem, so we can employ the pseudocode operations listed in Table 1-1 at this first level of refinement. Three major operations are identified by the preceding algorithm: getting data from the user, calculating the hypotenuse, and displaying the results to the user. As a result, we will create three additional algorithms that implement these operations. First, getting the data from the user. We will call this algorithm *GetData()*.

First Level of Refinement

```
GetData()
BEGIN
   Write a program description message to the user.
   Write a user prompt message to enter the first side of the triangle (A).
   Read (A).
   Write a user prompt message to enter the second side of the triangle (B).
   Read (B).
END.
```

STYLE TIP

The *GetData()* algorithm illustrates some operations that result in good programming style. Notice that the first *Write* operation is to write a program description message to the person running the program—the user. It is good practice always to include such a message so that the user understands what the program will do. In addition, the second *Write* operation will display a message to tell the user to "Enter the first side of the triangle (*A*)." Without such a prompt, the user will not know what to do. You must write a user prompt message anytime the user must enter data via the keyboard. Such a message should tell the user what is to be entered and in what format the information is to be entered. (More about this later.)

In Table 1-1, you will find two sequence operations called ***Read*** and ***Write***. The *Read* operation is an input operation. We will assume that this operation will obtain data entered via the system keyboard. The *Write* operation is an output operation. We will assume that this operation causes information to be displayed on the system monitor. The *GetData()* algorithm uses the *Write* operation to display a ***prompt*** to the user and a corresponding *Read* operation to obtain the user input and assign it to the respective variable.

The next task is to develop an algorithm to calculate the hypotenuse of the triangle. We will call this algorithm *CalculateHypot()* and employ the required pseudocode operations from Table 1-1, as follows:

```
CalculateHypot()
BEGIN
   Square(A).
   Square(B).
   Assign (A^2 + B^2) to H^2.
   Assign square root of H^2 to H.
END.
```

The operations in this algorithm should be self-explanatory. Notice how the *Assign* operation works. For example, the statement *Assign square root of* H^2 *to* H sets the variable H to the value obtained from taking the square root of H^2. We could also express this operation as *Set* H^2 *to* $(A^2 + B^2)$. The *Assign* and *Set* operations are equivalent; however, the verb objects within the respective phrases are reversed.

The final task is to develop an algorithm to display the results. We will call this algorithm *DisplayResults()*. All we need here is a *Write* operation as follows:

```
DisplayResults()
BEGIN
   Write(H).
END.
```

That's all there is to it. The illustration in Figure 1-3 is called a ***structure diagram***, because it shows the overall structure of our problem solution. The structure diagram shows how the problem has been divided into a series of subproblems, whose collective solution will solve the initial problem. By using the above algorithms and structure diagram, this problem can be coded easily in any

structured language, such as C++. You have just witnessed a very simple example of structured top-down design using stepwise refinement.

When you begin coding in C++, you will translate the preceding algorithms directly into C++ code. Each of the algorithms will be coded as a C++ ***function***. Functions in C++ are subprograms designed to perform specific tasks, such as those performed by each of our algorithms. Thus, we will code a function called *GetData()* to get the data from the user, another function called *CalculateHypot()* to calculate the hypotenuse, and a third function called *DisplayResults()* to display the final results to the user. In addition, we will code a function called *main()* to sequentially ***call*** these functions as needed to produce the desired result.

> A ***function*** in C++ is a subprogram designed to perform specific tasks, such as those performed by an algorithm.

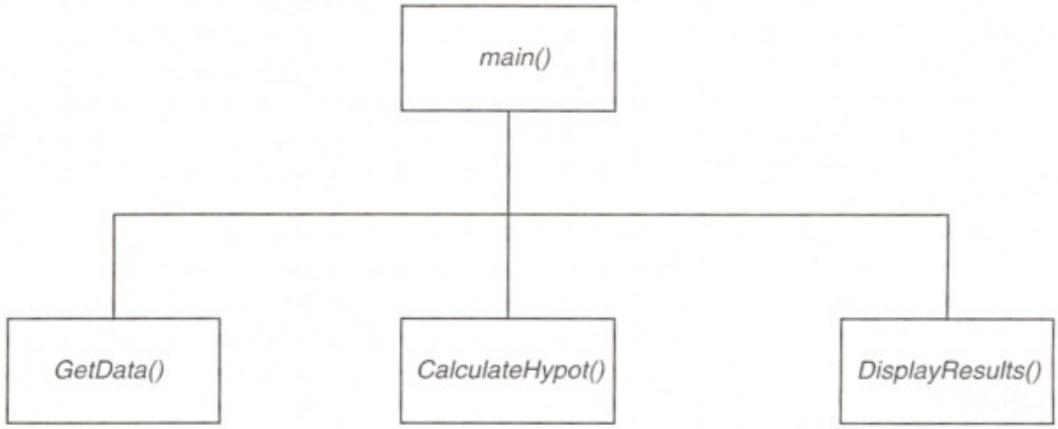


Figure 1-3 A structure diagram for the Pythagorean theorem problem shows the top-down structured design required to solve the problem using a structured language like C++.

PROBLEM SOLVING IN ACTION: SALES TAX

Problem

Develop a set of algorithms to calculate the amount of sales tax and the total cost of a sales item, including tax. Assume the sales tax rate is 7 percent and the user will enter the cost of the sales item.

Defining the Problem

Look for the nouns and verbs within the problem statement, as they often provide clues to the required output, input, and processing. The nouns suggest output and input, and the verbs suggest processing steps. The nouns relating to output and input are *sales tax*, *total cost*, and *cost*. The total cost is the required output, and the sales tax and item cost are needed as input to calculate the total cost of the item. However, the sales tax rate is given (7 percent), so the only data required for the algorithm is a user entry of the item cost.

The verb *calculate* requires us to process two things: the amount of sales tax, and the total cost of the item, including sales tax. Therefore, the processing must calculate the amount of sales tax and add this value to the item cost to obtain the total cost of the item. In summary, the problem definition in terms of output, input, and processing is as follows:

Output: The total cost of the sales item, including sales tax to be displayed on the system monitor.

Input: The cost of the sales item to be entered by the user on the system keyboard.

Processing: $Tax = 0.07 \times Cost$
$TotalCost = Cost + Tax$

Planning the Solution

Using the foregoing problem definition, we are now ready to write the initial algorithm as follows:

Initial Algorithm

```
main()
BEGIN
   Obtain the cost of the sales item from the user.
   Calculate the sales tax and total cost of the sales item.
   Display the total cost of the sales item, including sales tax.
END.
```

Again, we have divided the problem into three major tasks relating to input, processing, and output. The next task is to write a pseudocode algorithm for each task. We will refer to these algorithms as *GetData()*, *CalculateCost()*, and *DisplayResults()*. Due to the simplicity of this problem, we need only one level of refinement as follows:

First Level of Refinement

GetData()
BEGIN
  Write a program description message to the user.
  Write a user prompt to enter the cost of the item (*Cost*).
  Read (*Cost*).
END.

CalculateCost()
BEGIN
  Assign ($0.07 \times Cost$) to *Tax*.
  Assign ($Cost + Tax$) to *TotalCost*.
END.

DisplayCost()
BEGIN
  Write (*TotalCost*).
END.

Can you develop a structure diagram for this solution?

PROBLEM SOLVING IN ACTION: CREDIT CARD INTEREST

Problem

The interest charged on a credit card account depends on the remaining balance according to the following criteria: Interest charged is 18 percent up to $500 and 15 percent for any amount over $500. Develop the algorithms required to find the total amount of interest due on any given account balance.

Let's begin by defining the problem in terms of output, input, and processing.

Defining the Problem

Output: According to the problem statement, the obvious output must be the total amount of interest due. We will display this information on the system monitor.

Input: We will assume that the user will enter the account balance via the system keyboard.

Processing: Here is an application in which a decision-making operation must be included in the algorithm. There are two possibilities as follows:

1. *If* the balance is less than or equal to \$500, *then* the interest is 18 percent of the balance, or

$$Interest = 0.18 \times Balance$$

2. *If* the balance is over \$500, *then* the interest is 18 percent of the first \$500 plus 15 percent of any amount over \$500. In the form of an equation,

$$Interest = (0.18 \times 500) + [0.15 \times (Balance - 500)]$$

Notice the use of the two **if/then** statements in these two possibilities.

Planning the Solution

Our problem solution begins with the initial algorithm we have been calling *main()*. Again, this algorithm will simply reflect the problem definition as follows:

Initial Algorithm

```
main()
BEGIN
   Obtain the account balance from the user.
   Calculate the interest on the account balance.
   Display the calculated interest.
END.
```

Due to the simplicity of the problem, only one level of refinement is needed, as follows:

First Level of Refinement

GetData()
BEGIN
  Write a program description message to the user.
  Write a user prompt to enter the account balance (*Balance*).
  Read (*Balance*).
END.

CalculateInterest()
BEGIN
  If *Balance* <= 500 Then
    Assign (0.18 × *Balance*) to *Interest.*
  If *Balance* > 500 Then
    Assign (0.18 × 500) + [0.15 × (*Balance* – 500)] to *Interest.*
END.

DisplayResults()
BEGIN
  Write(*Interest*).
END.

As you can see, the two decision-making operations stated in the problem definition have been incorporated into our *CalculateInterest()* algorithm. Notice the use of indentation to show which calculation goes with which **if/then** operation. The use of indentation is an important part of pseudocode, because it shows the algorithm structure at a glance.

How might you replace the two **if/then** operations in this algorithm with a single **if/else** operation? Think about it, as it will be left as a problem at the end of the chapter!

CHAPTER SUMMARY

The five major steps that must be performed when developing software are (1) define the problem, (2) plan the problem solution, (3) code the program, (4)

test and debug the program, and (5) document the program. When defining the problem, you must consider the output, input, and processing requirements of the application. Planning the problem solution requires that you specify the problem solution steps via an algorithm. An algorithm is a series of step-by-step instructions that provides a solution to the problem in a finite amount of time.

Abstraction and stepwise refinement are powerful tools for problem solving. Abstraction allows you to view the problem in general terms, without agonizing over the implementation details of a computer language. Stepwise refinement is applied to an initial abstract problem solution to develop gradually a set of related algorithms that can be directly coded using a structured computer language, such as C++.

Once a codeable algorithm is developed, it must be coded into some formal language that the computer system can understand. The language used in this text is C++. Once coded, the program must be tested and debugged through desk-checking, compiling, and execution. Finally, the entire programming process, from problem definition to testing and debugging, must be documented so that it can be easily understood by you or anyone else working with it.

Your C++ debugger is one of the best tools that you can use to deal with bugs that creep into your programs. It allows you to do source-level debugging and helps with the two hardest parts of debugging: finding the error and finding the cause of the error. It does this by allowing you to trace into your programs and their functions one step at a time. This slows down the program execution so that you can examine the contents of the individual data elements and program output at any given point in the program.

QUESTIONS AND PROBLEMS

Questions

1. Define an algorithm.
2. List the five steps of the programmer's algorithm.
3. What three things must be considered during the problem-definition phase of programming?
4. What tools are employed for planning the solutions to a programming problem?
5. Explain how problem abstraction aids in solving problems.
6. Explain the process of stepwise refinement.
7. The writing of a program is called _____.

8. State three things that you can do to test and debug your programs.
9. List the minimum items required for good documentation.
10. What three characteristics must a good computer algorithm possess?
11. The three major control structures of a structured programming language are _____, _____, and _____.
12. Explain why a single **if/then** operation in the credit card interest problem won't work. If you know the balance is not less than or equal to $500, the balance must be greater than $500, right? So, why can't the second **if/then** operation be deleted?

Problems

Least Difficult

1. Develop a set of algorithms to compute the sum, difference, product, and quotient of any two integers entered by the user.
2. Revise the solution you obtained in problem 1 to protect it from a divide-by-zero run-time error.

More Difficult

3. Develop a set of algorithms to read in an employee's total weekly hours worked and rate of pay. Determine the gross weekly pay using "time-and-a-half" for anything over 40 hours.
4. Revise the solution generated in the credit card problem to employ a single **if/else** operation in place of the two **if/then** operations.
5. A dimension on a part drawing indicates that the length of the part is 3.00 ± 0.25 inches. This means that the minimum acceptable length of the part is 3.0 – 0.25 = 2.75 inches and the maximum acceptable length of the part is 3.00 + 0.25 = 3.25 inches. Develop a set of algorithms that will display "ACCEPTABLE" if the part is within tolerance and "UNACCEPTABLE" if the part is out of tolerance. Also, show your problem definition in terms of output, input, and processing.
6. Employ Ohm's law to develop a set of algorithms to calculate voltage from current and resistor values entered by the user. Ohm's law states that voltage is equal to the product of current and resistance.