Chapter 1

Problem solving and program design

1. Problems, solutions, and ways to express them
2. The branch control structure
3. The loop control structure
4. Modular design and object-oriented design
5. Problem solving through top-down design and stepwise refinement
6. Abstractions, functions, and recursion

Introduction

We learn computer science and computer programming today as a kind of *problem solving*: the aim is not beauty or knowledge, but practical solutions; ways to "get a job done."

How can we automate a task that is tedious when done by hand? How can we summarize business information that no one ever summarized before? How can we decide whether to allocate resources to a particular project? These are the kinds of questions businesses and other organizations face. They are *information processing* problems.

Such problems arise at a more and more rapid pace today, in a globally competitive economy, as information technology periodically doubles the data-processing power at the fingertips of managers, administrators, professionals, and everyone else. The power of information tends to reorganize the way we perform many tasks. Taking advantage of opportunities, and meeting the competition, forces us to face new problems and to try to solve them.

Computers and computer software are often sold as "solutions." If you have an information-related problem, then your first step is to see if software you own can solve it. Then you may look to acquire new software that does the job.

Software developers, and enterprises large enough to consider writing their own software, are in a position to hire people who will use the skills presented in this text. If you are looking for work in a growing industry, then it may serve you to learn those problem-solving skills and offer them to a high bidder.

Decades of experience in software development have taught that writing programs is more like constructing buildings than like writing poetry. The poet may find inspiration one evening and write a beautiful work of art. The software developer has more bases to touch. Some of the steps may seem tedious.

- Before even thinking about what to write in a computer program, the developer must be sure of what the problem is, precisely. What is the input? What is the output? How should they relate?
- Like a building, a complex piece of software must be *designed*. The design must be verified. Only after this stage is it cost effective to code the program in a language.

Experience has shown that the most efficient way to write programs is first to design them and then to write the program code. A written design is a useful form of documentation as well. Thinking before coding is known to save effort.

The first issues we address in this chapter are how to specify a problem, how to break a solution down into its component parts, and how to put that design for a solution down on paper. The abstract step-by-step plan for a solution is called an algorithm. All your study in computer science will involve work with algorithms. The three chief notations

historically have been *pseudocode*, *flowcharts*, and *module hierarchy charts*. As object-oriented technology has spread, *class diagrams* have become part of the documentation tool set as well.

Once a set of steps in a problem solution have been chosen, it is possible to arrange them in many different ways. Software developers have found, however, that a mere three such orderings, or *control structures*, are sufficient to solve any solvable problem, and that a design limited to these structures is immensely easier to understand than any other. This chapter will discuss the *sequence*, *branch*, and *loop* control structures and how to work with them in flowcharts and pseudocode.

Breaking down a problem in order to solve it also involves the construction of software *modules*. A section of this chapter will present the concepts of *functional decomposition* and *object-oriented design*. These differ chiefly in whether *operations* or *kinds of data item* are the primary focus of a particular software design.

At the right is a diagram of a *hierarchy*, in which some items have a supervisory or parent-like relationship to others, which are below them. We can build programs using such a hierarchical structure.

A typical software-development problem is too complex for any programmer or software designer to be able to hold all the details to its solution in mind at one time. Studies have indicated that a human being can concentrate on only about seven things at a time, at best. We overcome this obstacle in daily life by putting some details out of our minds temporarily. We deal with them later and concentrate for the moment on the big picture or on a single small part of it.

That's one way we can successfully approach software problem solving. We can look at a problem as a whole and break it down into subproblems. Then we can solve each of the subproblems, one at a time. If a subproblem is itself too complex, then we can break it down, and so on.

Most software development is done by *teams*, which may divide up the work into smaller parts. One member of the team may work with one part of the problem, another with another part. The team as a whole makes this division and puts the solutions together.

Breaking down a complex problem into subproblems is called *modular design*, or modular decomposition. Each sub-problem is solved by building a component to perform a single task—a procedural *module*. If a module is complex, we may break it into sub-modules.

A recent alternative to procedural decomposition of a problem is object-oriented design. Here we build our design around the categories of data *objects* found in our problem description. We started presenting objects in Chapter 1 and will continue this discussion throughout the text.

A second category of divide-and-conquer strategy is the *recursive* solution—a solution to a big problem expressed in part as a solution to a smaller instance of the same problem. Recursion, like modularity and object orientation, is a theme that occurs throughout the study of computer science. We will introduce it briefly as part of the discussion of the mathematical foundations of computer science.

1. Problems, solutions, and ways to express them

Key concepts:
- problem solving
- problem specification
- algorithm
- pseudocode
- flowchart
- program design

Programming does not begin with writing code in a programming language; it begins with understanding a problem that is to be solved. Problem solving includes asking any necessary questions about the problem to be sure that the user's needs are understood. Design is broken down into two stages:

- functional specification, in which *what* the program should do is planned, and
- architectural design, which determines *how*, internally, the program should accomplish its objectives.

At each stage of analysis and design, a software team may review its work.

Understand the problem

The first step in software development is to specify the problem to be solved. This may be more demanding than it sounds. Software development professionals can tell plenty of stories of how they demonstrated a program, or an early prototype, and found that the client's first comment was to request a change or addition to the program's original purpose.

The specification stage is also called *analysis*. The skills required have more to do with understanding a business or organization than using a programming language. The more thorough the analysis and specification are, the less effort will be needed at a later stage to re-specify the task that the program is to perform. Professionals who specialize in this work are called *systems analysts.*

Key components of a program's specification are its *input,* its *output,* and the relationship between them. Software developers have long prepared sample input/output as part of program specification. Most of our programming exercises include sample I/O. As software has become more complex, the specification may include entire screens, such as forms with which to get input or windows to display results.

Note that the specification phase does not address the critical question of *how* to arrive at the desired output from a given set of user input. That issue is left for the design phase.

An algorithm is a plan for a solution

To accomplish an information-processing task, we usually need to break a problem down into smaller sub-problems. This is because a problem worth solving is usually of some complexity. We simplify it by divide-and-conquer tactics. To solve the problem, "Display the sum of two numbers input by the user," our breakdown might be, in words:

1. Get two numbers from the user.
2. Add them, saving the sum.
3. Display this sum.

Or, as a diagram (Figure Chapter 1 -1).

The next step is to plan a design for a solution. Only after the design stage do professional programmers code their programs.

The preceding solution is not written in program code; it is still in an abstract form, in the form of an *algorithm*.

An algorithm is a precise plan to solve a problem or complete a task in a finite number of steps.

The first way we expressed the preceding algorithm is called *pseudocode*. It uses text and is precise but informal, not following a particular vocabulary, set of grammar rules, or style. Pseudocode is written in a natural language, as opposed to a programming language.

In pseudocode, we may use operators, such as the equal sign or the arithmetic operators (+, ×, –, ÷) of mathematics. One mathematical operator you may not have seen is the left arrow (←). We will use it to assign a value to a variable, as in *sum* ← *a* + *b*. You may wish to pronounce the arrow as "gets," as in "*sum* gets *a* + *b*."

The second standard way to express algorithms is called a *flowchart*. A flowchart is a diagram that pictures the flow of control, or the motion of activity, from one step to another. It is composed of geometric shapes, text, and arrows. The geometric shapes broadly define what type of instruction or instruction sequence is being executed. Text within each shape provides more specific information. The arrows indicate the direction of movement from one instruction or operation to the next command to be executed.

Each flowchart in this chapter will use some combination of the following four shapes or elements:

Element	Meaning
	Beginning or end of program or instruction sequence
	Input or output
	Decision
	Other processing
	Flow of control

Figure Chapter 1-1: Simple algorithm to add

Historical note

The word "algo-rithm" comes to us from the name of a great ninth-century mathematician, Mohammed ibn Musa Al-Khowa-rizmi. His work, written in Arabic, came to have in-fluence in Europe, where his algorith-mic method of calculating com-peted successfully with the use of the abacus. He is credited with inventing algebra, named after the title of a book he wrote. His mis-fortune was that algorithms came into their full use only with computer programming, more than a thousand years after he died!

Flowcharts were once seen as an indispensable tool for software designers and programmers. As programs have become larger and more complex, other tools have come to the fore. Here, we use flowcharts as a tool to teach control structures. They provide a picture of the jumps from step to step that occur at the level of machine code. You will see few flowcharts after this chapter.

Benefits of a design approach

We could, of course, sit down at a computer and write a program without ever consciously developing an algorithm, writing pseudocode, or drawing a flowchart. We could type what came into our minds and then try to fix it if it didn't work. That is called "hacking". (Some unlawful activities, such as computer break-ins and deliberate virus propagation, also go by the same name.) If you wish to hack code in the sense of programming without working from a design, then we wish you happy hacking. A warning: the time spent correcting errors in such programs is almost always enormously more than the time necessary to draw up a workable plan to start with.

A slogan that has been promoted among programmers is, "Test, then code." It means to verify your algorithm in your head or on paper before you code it in a programming language.

It is universally accepted in the software industry that *design* is a crucial step in efficiently solving a problem. Though it is an extra step that can be skipped if we wish to jump right away into coding a program, even the coding can be greatly speeded up if the code is based on a well-thought-out design. A common approach to problem solving and software development is expressed as follows:

Repeat until problem is solved:
- Determine problem specification
- Design algorithm
- Code program
- Test program
- Debug program

Of course, this software development process may not always require repetition of all steps. For example, it may be necessary to test and debug repeatedly, without changing program specifications and without redesigning the solution.

A flowchart to describe the same process might be like the one in Figure Chapter 1 -2.

Each phase is necessary. A poor understanding of what the program needs to input and output invalidates all the rest of the work. Design precedes efficient coding. Every program must be tested. If it is found not to work correctly, the errors (bugs) found must be corrected. Parts of this process must be repeated if later steps reveal flaws in earlier ones.

Later in this text, we will discuss in more detail the techniques of breaking down a problem and of *engineering* the solution. The development of software was once an art of individuals, but today it is so complex that most successful programming is done by use of science, mathematics, and teamwork.

Figure Chapter 1 -2: Phases of software development

2. The branch control structure

Key concepts:
- branch (decision)
- jump
- nested branch

It is easy to visualize the solution to a problem if that solution is just a series of steps carried out one after the other without variation or repetition. We could picture a straight road with no forks and no circles; only a series of highway signs or toll booths. That would describe a program written for our model processor using only the non-jump instructions.

But most problems can't be solved in such a linear manner. Here is one: *Accept from the user an integer, the user's age, and tell the user whether he or she is qualified to vote, assuming that the legal voting age is 18.*

A sequence of steps that always execute will not solve this problem; on some input, certain steps should be followed, while on other input, different steps should be taken.

Here is a solution:

```
If age is at least 18
    Say "OK"
```

We could diagram our solution as in Figure Chapter 1 -3.

The branch control structure is also known as the decision or selection structure. It always involves a yes/no test and, based on that test, a decision or selection of which branch to take out of two possible paths.

Often we will want to provide two alternative actions, rather than only the alternative of a certain action or none at all. The pseudocode below and the flowchart in Figure Chapter 1 -4 represent a design to solve the problem of displaying the greater of two user-input values.

Figure Chapter 1 -3. One-way branch

```
Prompt for integers a and b
If a > b
    display a
otherwise
    display b
```

The branch control structure admits only two possible courses of action. What if our problem requires three or more alternatives? We resort to using two branch structures, one after the other. To find the oldest of three people, for example, we may make two comparisons, each of which yields a decision:

```
Input age1, age2, age3
If age1 > age2
    oldest ← age1
otherwise
    oldest ← age2
If age3 > oldest
    oldest ← age3
Display oldest
```

Figure Chapter 1 -4. Two-way branch

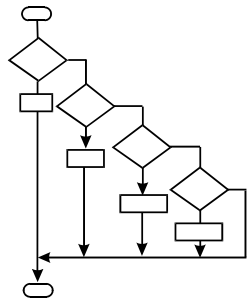


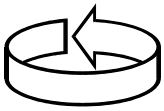
Figure Chapter 1 -5: Multiway branch

Here we identify four algebra-type variables, *age1*, *age2*, *age3*, and *oldest.* We assign values to these variables using the left-facing arrow.

The branch control structure may also be nested; that is, a branch may be put inside another branch. Figure Chapter 1 -5 is a flowchart for a branch nested to four levels.

3. The loop control structure

Key concepts:
- loop
- top tested
- bottom tested
- sentinel value
- structured design
- unstructured design
- trace
- repetition
- iteration



Many computer programs process large quantities of data in a repetitive way. At least one instruction sequence within such a program will be repeated for each data value or each set of associated data values to be processed. For example, a payroll program must repeat the same operations for each employee whose pay is to be computed. To repeat an instruction or a group of instructions as often as required for all of the data to be processed, a *loop structure* is needed. The loop is also known as the repetition or iteration control structure.

An easy example is drawing a square:

edge ← 100
Do four times:
  Draw a line segment of length *edge*
  Turn 90° to the right

Consider this problem: *Compute and display the sum of four input values.*
Pseudocode for a solution to the problem might look like this:

1. Set *sum* to 0.
2. Set *counter* to 4.
3. Repeat until *counter* = 0:
   a. Prompt for input value
   b. Add input value to *sum*
   c. Decrement *counter* by 1
4. Display *sum*.

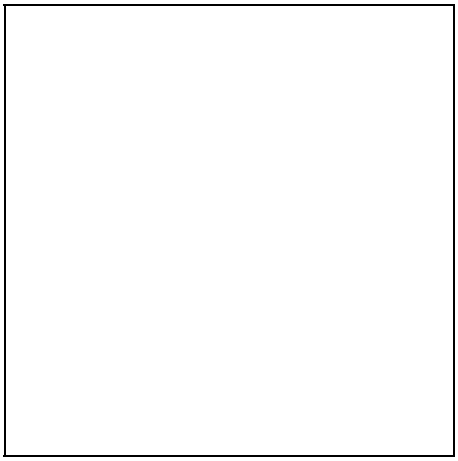


Figure Chapter 1 -6: Counted loop to input four input values and compute the sum.

The loop occurs in step 3. The steps *a*, *b*, and *c* under step 3 are together called the *body* of the loop. The flowchart in Figure Chapter 1 -6 diagrams the same solution.

An essential component of every loop is a test to determine whether to repeat the loop another time or to exit the loop. We will discuss three categories of loop, categorized by the ways to exit them:

- counter controlled
- sentinel tested
- general exit tested

In the first method, a counter variable controls the loop so that it iterates a predetermined number of times. In the second method, the loop is exited when a special value, called a *sentinel value*, is encountered. For example, a sentinel value of zero might be used to indicate the end of input of a list of integers. A loop whose exit test involves other than a counter or a sentinel value is in the third category.

The decision whether to repeat the loop body or not may be made either before or after executing the instructions making up the body of the loop. If the decision is made at the beginning of the loop, it is a *top-tested* loop. If the decision to repeat or not is made after executing the body of the loop, it is a *bottom-tested* loop. It is possible for the body of a top-tested loop not to be executed at all. The body of a bottom-tested loop, on the other hand, will always be executed at least once. Counted loops are top tested.

3.1. A counted-loop example

Consider the problem of finding a two-character pattern in a series of characters. Let us suppose we wish to know whether we have made the error of typing two commas in a row while entering text. The text is stored in a series of characters denoted by $input_1$, $input_2$, $input_3$, and so forth, up to n input characters. Pseudocode for one solution is as follows:

```
found ← false
for each value of i from 1 to n – 1
    if input_i is a comma
        if input_i+1 is a comma
            found ← true
If found is true
    display "Two consecutive commas found"
otherwise
    display "Two consecutive commas not found"
```

If all the temporary information used in an algorithm, such as variables, is hard to keep in mind at once, it is useful to trace its execution with pencil and paper. Often a table is useful.

The solution above could be traced by a table that lists some variables and their values, given input of "OK,,then":

i	***found***	***input $_i$***	***input $_{i+1}$***
1	false	O	K
2	false	K	,
3	false	,	,
4	true	,	t
5	true	t	h
6	true	h	e
7	true	e	n

The seven steps above are a trace of the duplicate-comma-finding algorithm as applied to the input string, "OK,,then." At each step, we wrote in the value of each variable at the *beginning* of the loop body. The trace establishes that the algorithm reports that "OK,,then" does have two consecutive commas.

Problems

1. Chapter 1 presented algorithms for converting between decimal and binary numerals and for adding and subtracting binary numerals. These algorithms involve counted loops. Write pseudocode or a flowchart for (a) binary-to-decimal conversion; (b) decimal-to-binary conversion; (c) binary addition; (d) binary subtraction
2. What is the output of this algorithm on input "OK,,then"? How does the text of the pseudocode compare to the pseudocode in the example in this subsection?

```
for each value of i from 1 to n – 1
    if input_i is a comma and input_i+1 is a comma
        display "Two consecutive commas found"
    otherwise
        display "Two consecutive commas not found"
```

3.2. A sentinel-controlled loop: getting input until the user asks to quit

Let's consider the same problem with one variation: instead of adding a fixed number of input values, our program should get input and add it until the user enters a special value, zero. Like the solution for the previous problem, the solution needs a loop; but it should not be a counted one. With this problem, it is impossible to say ahead of time how many iterations will occur. A flowchart of the solution is to the right.

The solution uses a *sentinel value*, zero, to tell when to exit from the loop. If the user enters a zero, that value is treated specially and terminates the loop. If the first input value is zero, how many times will the body of this bottom-tested loop execute?

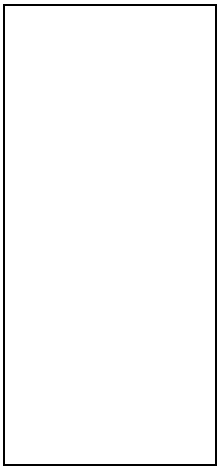


Figure Chapter 1 -7: The sentinel value is zero

Sentinel-controlled loops have a danger: the user may assume that a certain value is normal data, rather than a sentinel, or the program may test an item as a sentinel even if it were entered as normal data. Consider a program that reads a file of student records. The data-entry operator has been instructed to enter a special value of 99 for year of graduation as a sentinel to indicate end-of-file, and the program that reads the file uses that sentinel. No record for any student will be read, then, starting with the first student with a 1999 year of graduation! (This is one instance of the so-called "Year 2000" problem.)

3.3. A business problem

Let's consider a business matter: How much money will we end up paying back to the bank if we borrow $10,000 for four years at 10% interest? For simplicity, we'll assume there is only one payment, at the end of the loan term.

Since interest accrues repeatedly, the solution to this problem requires a loop. The data items we will need are the principal borrowed, the interest rate, the current year (relative to the date of the loan) and the amount owed to the bank.

We choose the tool of pseudocode to sketch our design of a solution to the problem:

```
amt owed ←10,000
interest rate ←0.1
year ←0
while year < 4
    add (interest rate times amt owed) to amt owed
    add 1 to year
```

It may be advisable to test our design by calculating results by hand to see if they are reasonable. We trace this algorithm below:

Interest rate 0.1

Year	Amt. owed	Interest	New amt. owed
0	10,000	1,000	11,000
1	11,000	1,100	12,100
2	12.100	1,210	13,310
3	13,310	1,331	14,641
4	14,641		

It's reasonable to consider paying $14,641 back after four years on a loan of $10,000 at 10% interest. We consider our design to be ready to implement.

3.4. A general-exit-tested loop

Perhaps you noticed that the double-comma-seeking algorithm mentioned before could have been improved. It could have been designed to terminate instantly upon finding two

commas in consecutive positions of the input. That would prevent wasting time looking for commas after the presence of double commas was already determined.

Such a modification would require a third kind of exit test, which puts the solution in a miscellaneous category of general-exit-tested loops. It could be written in pseudocode as follows:

```
i ← 1
found ← false
while not found and i < n – 1
    if input_i is a comma
        if input_{i+1} is a comma
            found ← true
    i ← i + 1
```

The exit test here checks two conditions: that *found* is not true and that the counter *i* has not reached the next-to-last character of the input. The word *and* indicates logical conjunction, the operation performed on bits by the AND gate at the hardware level.

3.5. Structured design in problem solving

Notice that our flowcharts each have exactly one beginning and one endpoint. This is true as well of every *component* of our flowcharts. Flowcharts with this feature are called *structured* flowcharts. We will encourage the use of structured design, reflected in structured flowcharts, as opposed to unstructured design. The flowcharts in and depart from the guideline of one-way-in, one-way-out.

The flowchart in Figure Chapter 1 -8 is an instance of an unstructured way of expressing an algorithm that would involve the use of the go-to concept in pseudocode, as below:

```
A. If x = 3 then go to step C
B. If y < 1 then
C.      Display (“y < 1”) and
D.      Go to step F
E. Display “x = 3”
F. Display “Done”
```

Figure Chapter 1 -8: An unstructured flowchart with two entrances to process D

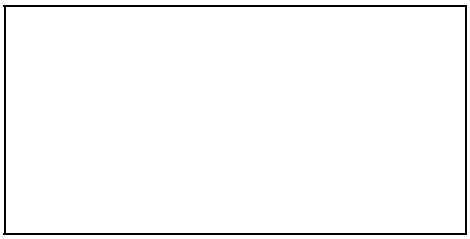
While it is possible to write a correctly-executing computer program from unstructured flowcharts or pseudocode, software professionals have found it difficult to understand the code and design of such programs, the more so as programs become larger. Practically no one writes commercial software today by unstructured techniques. If you are skeptical, we invite you to find out by experience.

A flowchart, or a piece of pseudocode, may be well organized or not. If we make no rules about the forms of our flowcharts, or if we often use the phrase “Go to...” in pseudocode, then we are practicing *unstructured design*. This is a close cousin to hacking. People who’ve practiced unstructured and structured design are likely to prefer the structured variety. It is much easier to understand algorithms expressed entirely as sequences, branches and loops.

One advantage of sequences, branches, and loops is that they have *one* entry point and *one* exit point. Therefore, we can replace any one of these branches, or loops, or sequences, with a rectangle, making it easier to understand the whole algorithm. The general rule is: avoid unstructured jumps. Flowcharts are best understood if they are easily decomposable into structured sub-flowcharts. Pseudocode is best written free of the phrase "go to".

While unstructured flowcharts, like plates of spaghetti, have an unlimited variety of internal structures, it can be proven that any flowchart that is possible to draw can be simplified into a structured one using the three basic control structures, the sequence, the branch, and the loop, nesting these structures within each other if necessary.

Figure Chapter 1 -9: An unstructured flowchart with two exits.



An unstructured design can be like a tangled hose

It is widely agreed that structured designs and structured flowcharts are understandable to humans, while an unstructured design is often impossible to follow above the scale of few isolated steps. Trying to understand one can be like unravelling a hundred feet of loose kite string.

The concepts of structured programming were developed in the 1960s and 1970s, long after computers had become widespread.

4. Modular design and object-oriented design

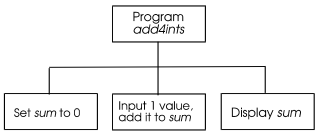
Key concepts:

- module
- object
- object-oriented programming
- window
- inheritance

4.1. Divide and conquer

As we prepare more and more complex program designs, we will want to break down our problems into simpler sub-problems. To solve a sub-problem, we will design a *module*, or a program component. This is a simple case of the more general strategy, "Divide and conquer."

Some modules correspond to tasks that our problem solution requires us to carry out. In this case, our design may be documented as a module hierarchy, where modules that use other modules appear above them in the hierarchy.



A module hierarchy may be of any depth. The modules higher up in the chart make use of the ones lower down. A module directly above another one, and connected to it, uses the one below it directly. The relationship between two modules may be indirect if a third module is connected between them.

The program design would very likely use a sequence of pseudocode for each module. We will discuss modular design in more detail in Chapter 8.

4.2. Object-oriented design

Another way to break down a problem is into the categories of entities (persons, things, events) that appear in the problem domain. Each entity has certain attributes and certain behaviors.

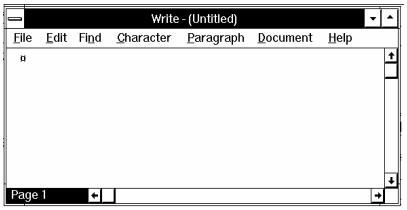
For relatively simple problems, a design built around pseudocode and module dependencies, that is, around *what happens*, will be effective. For very complex problems, the solution must focus on entities, their attributes, and their behaviors.

Let's take an example from real life. A computer is a thing, not an action. But if you turn it on, it goes into action—a disk spins, characters appear on the screen. The behavior is built into our notion of the object. If it didn't compute, it wouldn't be a computer.

Computer programs today model or represent things in the real world by defining data items with built-in behavior. A window on the screen in a word-processor program, for example, models a piece of paper on which you might type. When you select the window by clicking your mouse in it, the window comes to the front on the screen and lets you type words. The words appear in the window.

The window has more behavior built in than that. If you click in the lower-right corner and drag your mouse, the window changes size, letting you define the position of its lower-right corner with the mouse. If you click and drag in the title bar at the top, the window may be moved around the screen. If you click twice in the upper-left corner, the window disappears. The latter three behaviors are characteristic of all windows in the Windows, OS/2, and Macintosh environments.

A window on the screen is a data item, but its behavior is built into it. That is, program code is associated with each window data item.



A window in the Windows Write word processor

In the terminology of computer programming, a data item that is defined partly in terms of its behavior is called an *object*. One of the aims of this text is to present a relatively new way of writing software, called *object-oriented programming*. Our second C++ program example, in the next chapter, will have an object in it. Chapter 9 will show you how to define classes of objects in C++.

To design a simple program that adds two numbers, it is not necessary to define objects. Assembler languages don't supply tools to support working with data items as objects. But, to design a larger program, such as one that creates windows on the screen, it will be useful to make objects part of our planning. Thus, the flowchart, pseudocode and modular-breakdown techniques that we have used in this chapter in discussing design will be augmented later by other ways to describe our software models of the real world. Objects and categories of objects (classes) will become part of our design strategy.

A window object, for example, could be presented as a list of features. Some of them are data attributes and others are behaviors:

A window has:	
attributes	• horizontal and vertical location on screen
	• width, height
behaviors	• ability to be moved by user
	• ability to be resized
	• ability to be closed

The design of *asm.exe*, the processor simulator program presented in Appendix A, makes heavy use of objects. To *asm.exe*, an assembler program is an object with a name, a series of lines corresponding to RAM cells, a size (number of lines), and a screen view consisting of rectangles showing the contents of its RAM cells, the program counter, the instruction register, the accumulator, and the input/output area. The behavior of an assembler program includes loading itself from disk, executing itself, and updating its screen view.

Like each window in Windows, each rectangle on the screen in *asm.exe* has data properties and behaviors: title, contents, screen location, size, background color, color of contents. Its main form of behavior is to draw itself, either in normal colors or highlighted.

In the simulator application, the three processor registers, the RAM cells, and the input/output box are all specialized kinds of screen rectangles. Each special type has not only the attributes and behaviors of screen rectangles in general, but also particular attributes and behaviors characteristic of the special type. In a similar way, the word-processor window in Windows Write is a specialized version of the universal window

that is seen in the Windows environment. The specialized types *inherit* the features of the general types.

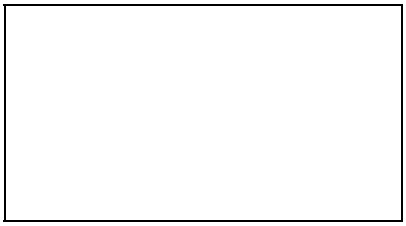


Figure Chapter 1 -10: A class diagram showing inheritance and containment relationships

The class diagram in Figure Chapter 1 -10 signifies that a data type, *supervisors*, models a category of people who are all employees; thus, a supervisor is a kind of employee. This relationship is an *inheritance* one.

A different kind of relationship exists between employees and addresses. The address of an employee may be an object, having a street and a city, and this object is a component of the employee object. The relationship between the classes *employees* and *addresses* is a *containment* relationship.

Software developers today use objects more and more to solve problems. The problems are increasingly complex and often require accurately modelling some aspect of the real world. The trend to ward reengineering business processes means that software developers must solve complex problems quickly by reusing program code and building on models that have been constructed earlier. Object-oriented programming and object-oriented design have joined earlier techniques at the center of the programmer's and software engineer's toolbox.

Lab activity
Open a file directory window on your computer and describe some of the properties (data attributes) and behaviors of the window. Which of them are the same as the attributes of all screen windows, as described in this section?

5. Problem solving through top-down design and stepwise refinement

Key concepts:
- top-down design
- stepwise refinement
- desk checking

We defined an algorithm above as a precise plan to solve a problem or complete a task in a finite number of steps. Algorithms were presented using flowcharts and pseudocode. Flowcharts have the advantage of being pictorial. Pseudocode may be terse; it has the advantage over C++ of being closer to English. Pseudocode allows you to show the logic required to solve the problem without being concerned about the syntax of a specific programming language such as C++.

A widely used way to solve a software-development problem is the *top-down* approach. Top-down design means identifying the main, or top-level, steps of the solution first. Some or all of the main steps may be subsequently broken down into secondary steps or instruction sequences. Some of the secondary steps may be further refined into even lower-level steps. This successively more detailed development of the algorithm is called *stepwise refinement*. If the problem solution is complex, the final pseudocode expression of the algorithm might resemble a research paper outline:

I. (major step)
   A. (substep of I.)
   B. (substep of I.)
      1. (substep of I.B);
      2. (substep of I.B);
   C. (substep of I.)
II. (another major step)
   A. (substep of II.)
   B. (substep of II.)
III. (another major step)

To take an example outside of software development, no home builder would try to construct a house without a plan. The original plan may be drawn up by an architect. The house design will go to a construction company, which will break down the construction process into separate tasks and draw up a schedule. Some tasks in construction might be: lay foundation, build frame, lay floor, raise wall, install pipe, install wiring, install insulation, and plaster interior. Some steps, such as "lay floor," may be repeated for different parts of the house. Different teams of workers might carry out different tasks. Software construction, too, is divided into tasks carried out by teams.

Below is a seven-step methodology for developing successful software solutions. It is a slight variation on the one presented earlier. You are likely to need such an approach for larger projects. Beginning programmers may find even short exercises less time consuming if they use it.

Notice as you read through this sequence of steps that much preliminary work is done before we start coding a solution in C++.

A problem-solving methodology

1. Clearly *identify the problem*. We have seen some very creative and well-written programs that solve the wrong problem! Make sure that you and the person for whom you're writing the program agree on the problem to be solved.
2. Prepare a *sample of the output* your program is to generate.
3. Specify the necessary *input data* to produce the required output.
4. Using top-down design with stepwise refinement, *prepare an algorithm,* either in flowchart form or in pseudocode. (If your problem involves two or more different entities with attributes and behaviors, such as customers, employees, or transactions, then your design step should include some attention to these entities and their characteristics—see Chapter 9.)
5. Using pencil and paper, *test your algorithm with sample data.* Use sample data that will test all parts of your algorithm. If your pencil-and-paper testing reveals 'bugs' (errors in logic), revise the algorithm. (You may be in this revision loop for some time if the problem is particularly complicated.)
6. *Code the solution.* That is, write a C++ program based on your algorithm.
7. *Test your program.* Careful work in steps 4 and 5 will pay off with fewer errors at program testing time. Again, as in step 5, use sample data that will test every part of your program.

After Step 7, you may need to repeat some or all of the steps—coding, design, or even program specifications—until you are fully satisfied with the results.

5.1. Case study: a payroll application

Here is a simple example that uses the above seven-step problem-solving method.

1. Identify the problem.

The problem description, or specifications, is to compute the gross weekly pay for an hourly employee, based on regular hourly pay for hours up to 40 and time-and-a-half for overtime. If the slightest doubt exists in the mind of the software developer about the problem to be solved, he or she must consult with the customer for clarification. The first step is also an appropriate occasion to suggest additional specifications that the customer may not be aware are feasible or desirable.

2. Prepare sample output

The output sample should include reasonable values, including the calculated value, gross pay. Although our problem specification only calls for working with one employee's hours and pay, we plan to format our output as if it were a table, with labeled columns:

Name	Rate	Hours	Gross pay
----	----	-----	---------
Samuel B. Jones	8.75	45	415.63

3. Determine necessary input data

To arrive at our specified output, we need to prompt the user for the employee's name, from the employee roster; hourly pay rate, also from the roster; and number of hours worked, from the time card.

4. Design an algorithm.

Using the top-down design principle, pseudocode of an initial version of the algorithm might be:

1. Input *name, pay_rate, hours_worked*
2. Compute *gross_pay*, including overtime
3. Show *name, pay_rate, hours_worked, gross_pay* with headings

The first draft is only a skeleton. Shall we fill in some details? Using stepwise refinement, we provide more substance to our algorithm:

1. Input raw data
   A. Prompt for name; input name
   B. Prompt for rate; input hourly rate
   C. Prompt for hours; input hours worked
2. Compute gross pay
   A. Compute regular pay
   B. Compute overtime pay
   C. Add overtime pay and regular pay to get gross pay
3. Show payroll report
   A. Show headings
   B. Display dashes under headings
   C. Show name, rate, hours, gross pay to 2 decimal places

We look over our latest draft of the algorithm, and discover that the logic for computing regular pay and overtime is not indicated. In fact, we realize that our as yet rather limited programming tools require that we input hours as two separate entries, regular hours and overtime hours. Here is a third draft of the algorithm. Notice that Parts 1 and 2 of the second version have been considerably revised:

1. Input raw data
   A. Prompt for name; input name
   B. Prompt for rate; input hourly rate
   C. Prompt for regular hours. The user should be prompted to input only 40 if hours worked were over 40.
   D. Prompt for overtime hours. User should be prompted to input hours worked over 40 if work included overtime, otherwise to input 0.
2. Compute gross pay:
   gross pay ← pay rate × (regular hours + 1.5 × overtime hours)
3. Display payroll report
   A. Show headings
   B. Show dashes under headings
   C. Show name, rate, hours, gross pay (to 2 decimal places)

5. Test the algorithm with sample data

In order to test all parts of our algorithm, we will need to use two sets of sample data, one with no overtime hours and another set with overtime hours. Since this is to be a paper-and-pencil test, we should choose sample data that will make for easy computation. The first set of test data will be for Les Toil, who worked at $10.00 per hour for 40 hours.

The portion of our algorithm that requires testing is the computation part. Here are the inputs involved in computing gross pay:

Rate: 10.00
Regular hours: 40
Overtime hours: 0

and here is the computation:

gross = 10.00 × (40 + 1.5 × 0)
= 10.00 × (40 + 0)
= 10.00 × 40
= 400.00

The algorithm produces a correct result for the first set of data.

The second set of test data, which must test the overtime computation, will be for Morey Work, who worked at $10.00 per hour for 50 hours.

Here are the inputs used to compute gross pay:

Rate: 10.00
Regular hours: 40
Overtime hours: 10

Here is the computation:

gross = 10.00 × (40 + 1.5 × 10)
= 10.00 × (40 + 15)
= 10.00 × 55
= 550.00

The algorithm produces a correct result for the second set of data.

Testing a program before coding it is called *desk checking.* It requires some extra work initially, but it will reduce the number of errors that occur when the coded solution is tested. Care and attention are musts in desk checking; it is easy to get seemingly correct

results by performing what we *think* the pseudocode says, rather than applying it literally, as a compiler will translate our program code. Sloppy desk checking reinforces errors rather than catching them.

Key concepts
- abstraction
- function
- computation
- computable function
- recurrence
- recursion

6. Abstractions, functions, and recursion

Computer science is concerned with the practical issues of data manipulation by hardware and writing programs to accomplish this data manipulation. But it also has a theoretical aspect that is of critical practical importance. It involves mathematics. Theoretical computer science not only *uses* mathematics, as physics, chemistry and other fields use certain parts of math; computer-science theory *is* a branch of mathematics, also known in part as discrete mathematics.

6.1. Computers and abstractions

Mathematics is a field of study that works entirely with abstractions, such as numbers and operations. Similarly, computers by their nature work with abstractions, because they are symbol-manipulating machines. Letters, numbers, and words are among the symbols they operate on.

The *process* of abstraction lets us set aside concrete details (e.g., what do we have three of?) and concentrate on the matter at hand (e.g., three of anything plus five of anything equals eight of that thing).

As we will discuss in detail, writing a subprogram module to accomplish part of a program specification, such as output, is procedural abstraction, and designing classes, such as window classes, is a form of data abstraction. In each case we focus initially on deciding how a software component will work and on naming it, but we temporarily leave aside implementation details that are not crucial to the user of the component.

6.2. Sets and functions

If you studied algebra using a mathematical approach (as opposed to a formula-memorizing approach), then you know about *sets*. Three well-known sets are the set of natural numbers (0, 1, 2 ...), the set of real numbers (numbers that can each be represented as a series of digits with a decimal point somewhere in the series), and the set of truth values, {*True, False*}.

In algebra, geometry, and trigonometry, you encountered *functions*. Perhaps you learned that a function is a set, too. It is a *mapping* from one set to another set, possibly to the same one. Every function consists of ordered pairs of values in such a way that a given value in the first set always maps to a unique determined value in the second one. Thus, given a certain value on the left (the function argument), a function is quite predictable and always returns the same value on the right. Two values on the left, however, may both have the same return value for a certain function.

One example of a function is the one that returns the ordinal value, or position, of a letter in a series of letters. The letters ‘A’, ‘B’, and ‘C’, for example, have the positions 1, 2, and 3 in the alphabet.

So our position function might be diagrammed as in Figure Chapter 1 - 11. We have the sets {‘A’, ‘B’, ‘C”}, {1, 2, 3}, and {(‘A’,1), (‘B’,2), (‘C’,3)}. The third set is a function, which we could call *index*:

index(‘A’) = 1
index(‘B’) = 2
index(‘C’) = 3

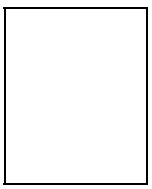


Figure Chapter 1 - 11. A function from letters to natural numbers

A similar mapping can define what we mean mathematically by oddness (Figure Chapter 1 -12):

odd(1) = *true*
odd(2) = *false*
odd(3) = *true*
odd(4) = *false*
...

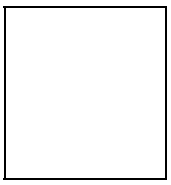


Figure Chapter 1 -12. A function from natural numbers to truth values

Notice that while none of our functions have two arrows coming out of the same argument set element, some may have two or more arrows pointing to the same return-value set element.

Finally, we have functions from numbers to numbers, such as the function that returns a value twice as large as its argument (input), illustrated in Figure Chapter 1 -13:

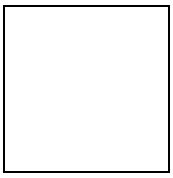


Figure Chapter 1 -13. A function from natural numbers to natural numbers

twice(1) = 2
twice(2) = 4
twice(3) = 6
twice(4) = 8
...

Any of the functions described above could be extended to map from or between large or even infinite sets.

6.3. Computable functions

In mathematics, a function is a passive, static abstraction. However, a very interesting set of functions is the category that can be computed. That is, for each *computable* function, there is one or more algorithm that starts with the argument value and step by step arrives at the return value. The number of steps must be finite for us to call the function computable.

Since you know that computer programs are supposed to be deterministic (predictable), it may make sense to you that every computer program that works correctly produces as its output the return value of a computable function, with the program's input corresponding to the function's argument. It may also make sense that for every computable function, it is possible to write a computer program whose output is the return value of the function. In fact, it is possible to write many such programs. Thus, we have a neat correspondence between the category of possible computer programs and a category of mathematical functions.

This is true whether we take a narrow view and consider only programs with numeric inputs and outputs, or a broad view, considering inputs and outputs that include text, graphics, mouse clicks, and so forth. Since all computer data is stored as bit patterns, and we have shown that any bit pattern corresponds to a number, therefore we could consider only that set of functions from numbers to numbers and still defend our claim that the class of computable mathematical functions has a correspondence to the set of all possible computer programs.

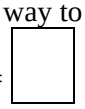
Now, when we ourselves compute functions by hand we use operators and other notation to specify the steps we take to get from the function's argument to its return value. For example, any two numbers have a sum, so we could define a mathematical function *sum*(*a*,*b*) whose arguments could be any natural numbers *a* and *b*. This function would return the sum of the two arguments. We may use the operator, +, to denote our use of the function, in this way: *a*+*b*.

Computing the sum, or +, function, may consist of looking up one or more values in an addition table and possibly taking other steps.

The expression, $(2 + 5) \times 4$, with the operators + and ×, corresponds to the application of a product function to two arguments, one of which is the sum of 2 and 5:

$(2 + 5) \times 4 = product(sum(2,5), 4)$

Finally, the language of mathematics provides us with ways to express functions of different numbers of arguments if the argument values progress in a natural way from one to the other. Consider *sum*(1, 2, 3, 4) or *sum*(1, 2, ... *n*) for some natural number *n*; in other words, consider the sum of a series of consecutive numbers starting with 1. Mathematics has a way to express such functions:

sum(1, 2, 3, 4) = 

sum(1, 2, ..., *n*) = 

Here the symbol Σ (a Greek letter) is pronounced “sigma” or “summation.” It is used more or less as a super-powered plus sign.

Thus, mathematics gives us two ways to express the values of certain functions: the functional notation, with function names followed by arguments in parentheses, and operator notation. The operator notation often points to a step-by-step method for computing a function that is computable.

A *computation* is a finite sequence of concrete steps that begins with a computable function’s arguments and ends with its result or return value. For example,

 = *sum*(1, 2, 3, 4) = 1 + 2 + 3 + 4 = 3 + 3 + 4 = 6 + 4 = 10

The computation of this function took three steps, equal to the number of plus signs in the first expression with operators.

A computation is a particular series of operations on particular data values, whereas an algorithm is an abstract plan for computations on any of a wide variety of data values.

Whereas a function is a passive set of ordered pairs, a computation entails activity. A computer program or a subprogram computes a mathematical function. Later you will learn about C++ “functions”. These are subprograms, not mathematical functions.

6.4. Recursive mathematical functions

Some functions in mathematics are not computable. For example, we can imagine a function whose argument is the executable file for a computer program, and whose return value is *true* or *false*. This function take the value *true* if the argument program ever goes into an infinite loop (“hangs” or “freezes”, in your experience). It returns *false* if the argument program always terminates. Such a function would be highly useful implemented as a computer program, because with such a program we could certify whether the software we are buying, selling, or using is reliable in a crucial way.

Unfortunately, this function is uncomputable, regardless of what processor the executable file is designed to run on. The program we would like to write, to compute this function and earn perhaps billions of dollars, cannot be written.

How can we recognize computable functions, so as to avoid taking on impossible software-development tasks and focus on work with fruitful prospects? It turns out that every computable function can be expressed according to a certain form, the *recurrence*.

Here is a simple recurrence that assumes the operation “+” is defined:

$sum(a,b) = a + b$

It is too trivial to discuss.

Here is a more interesting one, which assumes that division is defined:

$$quotient(a,b) = \begin{cases} a \div b & \text{if } b \neq 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Notice that the definition on the right side of the recurrence is conditional; it depends on the value of *b*. This recurrence says that *quotient* is a function that has a return value of any arguments *a* and *b*, except where *b* is 0. To be undefined for certain argument values does not disqualify *quotient* from being a function.

Clearly both *sum* and *quotient* are computable, since we compute them all the time by hand or with calculators.

You may wonder what is recurring in the above recurrences. Nothing—but they follow a format in which something *may* recur, i.e., occur over and over.

Consider this:

$$sum(a,b) = \begin{cases} \text{undefined} & \text{if } a < 0 \\ \begin{cases} b & \text{if } a = 0 \\ sum(a-1, b+1) & \text{otherwise} \end{cases} \end{cases}$$

base cases

recursive case

Here we define the *sum* function conditionally on the value of *a*. In the case where *a* is negative, we choose to leave the return value undefined. (If we wrote a program to compute the function, it would show an error message.) If *a* is 0, our function expression *sum*(*a*,*b*) takes the value *b*. Thus, for example, *sum*(0,5) = 5 by our definition. OK? So much for the easy part.

In the case where *a* is positive, our function definition reuses itself, or *recurses*: *sum*(*a*–1,*b*+1) is returned. Let's step through the definition as it recurses with the argument values 2 and 1, to see how *sum*(2,1) comes out:

sum(2,1)	= *sum*(2–1,1+1)	by the recursive case
	= *sum*(1,2)	applying the – and + operators
	= *sum*(1–1,2+1)	by the recursive case
	= *sum*(0,3)	applying the operators
	= 3	by the base case

Our new definition of *sum* yields 2 + 1 = 3.

You might notice that as we apply and re-apply our definition of the *sum* function, the arguments *a* and *b* take on successive values where *a* decreases with each step of recursion and *b* increases. Thus whatever initial values *a* and *b* take, on some eventual recursive invocation of the definition, *a* will take the value 0 and the second line of the recurrence (*b*, if *a* = 0), the base case, will apply.

6.5. Recursive functions are computable

Why would we ever want to define a function *sum* in this way, when "*sum*(*a*,*b*) = *a* + *b*" seems simpler? Because in practice the addition operator is more complex to apply than would appear on the surface. For us, addition requires at least an addition table and possibly repeated one-digit additions and carries. It is similar for a computer. But adding 1 to (incrementing) a value or subtracting 1 from (decrementing) it is simpler; quite simple, at the hardware level. Mathematically, we could call finding the successor or predecessor of a natural number a *primitive* operation, defined simply by the very nature of natural numbers. The natural numbers are defined as 0 and the successors of natural numbers, where each natural number has exactly one immediate successor.

What is significant is that any function that we can define by a recurrence is computable if every component on the right side of the recurrence is computable. In other words, even if we don't choose to use recursion in every algorithm we use in a computer

program, nevertheless if it is possible to express as a recurrence the function our program is to compute (the problem it is to solve), we can be sure that it is also possible to write our desired program using some algorithm that computes the function. When we are solving a problem, it is useful to have a way of knowing that it can be solved.

Any loop can be specified using recursive pseudocode. Let's consider a very simple looping problem. What do you do to walk a distance of *n* steps? We could say, "Repeat *n* times: take one step," but another solution offers itself. Consider this algorithm:

Walk (*num-steps*)
If *num-steps* > 0
  Take a step
  Walk (*num-steps* – 1)

Here there are two possibilities. If the argument, or operand, or parameter, *num-steps*, is zero, then the algorithm will do nothing. But if it is one or higher, the algorithm reinvokes itself with a slightly smaller argument. That slightly smaller argument might be zero, or the reinvoked version of the algorithm might call itself with an argument of zero. Eventually, *num-steps* will get down to zero, and the recursion will stop.

We will be discussing recursive definitions of C++ grammar rules in Chapter 3, and how to write recursive subprograms in C++ in Chapters 8 and beyond.

Mathematics and computer science are in some ways two paradises, two playgrounds, for skeptical people. Every claim about functions, computable functions, and recursion made above can be proven, though we don't do so here. Though not every correct program can be proven correct mathematically, every such program can be rewritten in such a way that the result can be proven correct.

Every computer with the proper software-development environment is an inexpensive laboratory for testing problem solutions empirically. If you have a bright software idea, you can build and demonstrate a working prototype yourself.

As we shall see, a mathematical approach to software development has some practical advantages over the empirical testing approach. Software engineers make use of both.

Summary

The specification stage in software development is called analysis. The software developer must understand the problem at hand in order to solve it.

An algorithm is a precise plan to solve a problem in a finite number of steps. To solve a problem, we may break it down into smaller sub-problems. Design is a critical phase in the problem-solving process.

Flowcharts are a conventional way to pictorially represent the order of execution of instructions. Pseudocode is a more commonly used informal way to describe an algorithm in words.

Program design is considered an essential step in the software development process. This problem-solving cycle may be described as a series of repeated steps: specify the problem, design an algorithm, code a program, test the program, and debug the program. Each step, or the whole series, is repeated until the problem is solved.

The three fundamental control structures used in programs are the sequence, the branch, and the loop. Any algorithm may be diagrammed by a flowchart combining these three control structures.

We use a branch control structure to cause a sequence of instructions to be executed under some conditions but not under others, or to cause one sequence to execute if a test condition is true and another to execute otherwise. We may combine and nest branches for multiply branching algorithms.

Repetition is implemented through the loop control structure. A loop must have an exit condition, which is tested either before the body of the loop (top tested) or after it

(bottom tested). A counter may be used to cause loops to execute a predetermined number of times. A sentinel value input by the user may also be used to exit from a loop.

The software designer breaks a complex problem down into sub-problems. The solution to each sub-problem may be designed as a module.

For highly complex problems such as those requiring display of data in screen windows, a program must be designed with objects in mind. Whereas in very small programs we work with instructions that manipulate data items, in larger ones we typically work with objects—data items whose behavior is built in. Some classes of objects may be defined to inherit the features of other classes. Object-oriented design and object-oriented programming are indispensable tools of software engineering today.

Application programming increasingly consists of assembling software components from standard and proprietary libraries—tool kits of predefined constants, functions, and classes.

The accepted approach to program design is the top-down method, using stepwise refinement. A design is often first expressed in pseudocode.

Writing a program can be accomplished using a seven-step process:

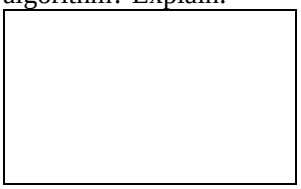
1. Identify the problem.
2. Specify output.
3. Specify input data.
4. Write an algorithm.
5. Desk check the algorithm using representative sample data.
6. Code the solution in a programming language.
7. Test the coded solution on the computer.

It is necessary to repeat steps whenever testing reveals an error.

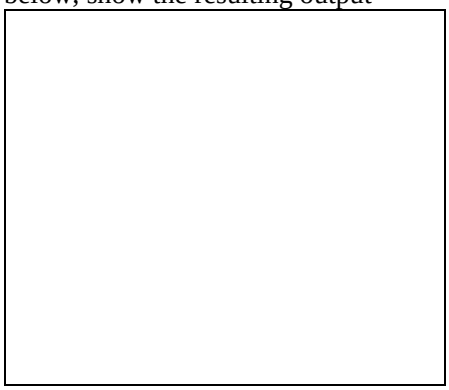
Mathematics and much of computer science share the feature that their subject matter consists of abstractions, such as algorithms, numbers, symbols, and functions. A computer is a machine that manipulates abstractions (symbols). A function is a mapping from a set to a set. A computable function is one for which an algorithm exists that starts with the function's argument value and arrives at the return value. Some functions are uncomputable. Functions that can be defined by recurrences are computable. Any computable function can be defined recursively.

Review problems

1. Write a flowchart or pseudocode for algorithms to solve each of the following problems.
   (a) Input exactly 6 signed integers. Display only the largest of the input values, regardless of where it occurred in the input list. *Hint*: Let the data address for the first input also serve as the storage location for the largest integer found so far. Use a second data address for subsequent input values. *Sample Input:* –3,20,-4,5,7,0 *Output:* 20
   (b) Input signed integers until the current input is less than the previous input. Display the largest input value. *Sample Input:* 1,2,3,4,24,56,41 *Output:* 56
2. Does the flowchart below diagram an algorithm? Explain.

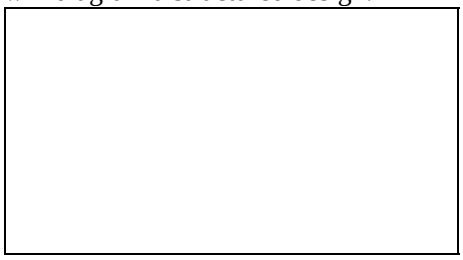


3. In the software development process, what steps are recommended before coding a program? After coding?
4. Give an example of an object that is found on the screen in the Windows or OS/2 user environment. What are some of its data attributes? Its behaviors?
5. How many times will a counter-controlled loop iterate? A sentinel-controlled loop?
6. Describe two variants of the loop control structure.
7. Consider the flowchart to the right. For each of the input pairs (A,B) shown below, show the resulting output



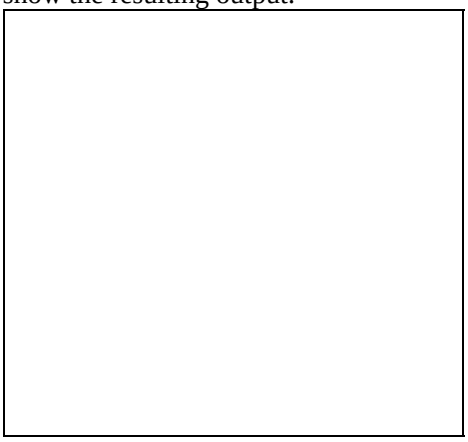
Input A	Input B	Output
1	4	_____
2	0	_____
3	1	_____

   What simple function does this algorithm calculate?
8. Is the pseudocode below an example of a structured design? Why or why not?
   1. If input file exists, open input file; else exit program
   2. Read input file, summing up contents
   3. Display sum.
9. Modify the flowchart below so that it will diagram a structured design.



10. Use pencil and paper to test a few argument values and guess what familiar mathematical functions are computed by the following recurrences:
    (a) $f(a,b) = \begin{cases} 0 & \text{if } a = 0 \\ b + f(a-1,b) & \text{otherwise} \end{cases}$
    (b) (Challenge:) $g(a,b) = \begin{cases} 0 & \text{if } a = 0 \\ b + g(\lfloor a \div 2 \rfloor, 2b) & \text{if } a \text{ is odd} \\ g(\lfloor a \div 2 \rfloor, 2b) & \text{if } a \text{ is even} \end{cases}$

11. Put these phases or sub-phases of the problem-solving process in chronological order, number the first "1", the second "2", etc.
    ____ code program
    ____ desk check
    ____ write a design
    ____ get problem specifications
    ____ test program
    ____ debug code
12. Consider the flowchart below. For each of the input pairs (A,B), shown below, show the resulting output.



Input A	***Input B***	***Output***
2	1	_____
1	3	_____
4	2	_____

    What is a way to describe the relationship between input and output?
13. Label each term below with the letter of its appropriate definition
    ____ pseudocode
    ____ object
    ____ algorithm
    ____ branch
    ____ loop
    ____ module
    ____ desk checking
    ____ stepwise refinement
    ____ top-down design
    a) A precise plan to solve a problem or complete a task in a finite number of steps.
    b) Informal natural-language way to express an algorithm.
    c) The decision control structure, in which one action is taken or else another.
    d) The iteration control structure, in which an action is repeated.
    e) A way to design and code software characterized by use of only three control structures: sequence, branch, and loop.
    f) A data item that is defined partly in terms of its behavior.
    g) Verification of program correctness without running it on a computer.
    h) A program component which may consist of one or more subprograms.
    i) A method of developing a plan for a program, beginning with an overview of the problem and breaking it down.
    j) A method that uses repeated improvements in a program design.

Problem-solving exercises

In the following exercises, prepare pseudocode or a flowchart of your solution. If you have read Appendix A, you may as a lab exercise wish to code the solutions in the language of the model processor explained there.

1. Design a program to accept keyboard input of three integers to represent the dollar amounts price, discount, and sales tax, in cents. It should display the sum of the price and the tax, minus the discount.
   Sample I/O:
   [Input:] 200
   [Input:] 20
   [Input:] 10
   [Output:] 190
2. Computers are often sold with service plans whose cost depends on the computer's value. Design a program to input the signed integers, *price* and *monitor*. If the sum of these is less than 1000, the program should display the message, "Plan costs $99.95"; otherwise it should display the message, "Plan costs $149.95".
   Sample I/O:
   Example 1: [Input:] 749 [Input:] 199 [Output:] Plan costs $99.95
   Example 2: [Input:] 1299 [Input:] 399 [Output:] Plan costs $149.95
3. Design a program to input three integers, A, B, and C. Make the necessary comparisons to display the greatest of the three.
   Sample I/O: [Input:] –38 [Input:] 300 [Input:] 77 [Output:] 300
4. Design a program to input integers, A and B. Compute and display |A – B|. Note: Read |A – B| as "the absolute value of the difference (A – B)".
5. Design a program without input that uses a loop structure to display each of the integers from 1 through 10. The only data values that may be stored initially via data statements are 0, 1, and 10.
6. Design a program that will loop to accept input of exactly three pairs of integers (A,B) and compute and display the value of A – B for each input pair.
7. Design a program to compute and display the product of two input non-negative integers. Display nothing if input includes a negative number. (Hint: Perform the multiplication as repeated addition, using one of the integers as an addend and the other integer as a counter to determine how many times to add the addend to a sum representing the product.)
8. (Challenge) Design a program to input two non-negative integers (A,B) and a positive integer (C). Compute A * B / C and display the quotient and remainder. For each of the inputs A, B, and C, loop for new input if negative values are entered. The input value of C must also be tested to be sure it is not 0. Why?
9. (Challenge) A *geometric progression* is a sequence of terms in which each term after the first term is obtained by multiplying the previous term by a constant multiplier. For example, if the first term is 7 and the constant multiplier is 3, then the resulting geometric progression is:
   7 21 63 189 567 1701 etc.
   We can compute the sum of the first *n* terms of a geometric progression. In the preceding example, the sum of the first 5 terms is:
   7 + 21 + 63 + 189 + 567 = 847.
   Design a program that will accept positive integers *n*, *first*, and *k*, input by the user, and display the first *n* terms and then the sum of those terms where the first term is *first*, the constant multiplier is *k*, and the desired number of terms is *n*.
10. (Challenge) Design a program to divide any signed integer by any other non-zero integer, using only addition, subtraction, and the three control structures. First the dividend and then the divisor are to be input from the keyboard. The divisor must be tested to avoid attempted division by 0, since division by 0 is not defined. The result is to be output as an integer quotient followed by an integer remainder. Remember to test your program with all possible sign combinations of the dividend and divisor.

11. Write pseudocode to compute the base-2 integer logarithm of an input integer. (See flowchart below.)

12. Write an algorithm to find the tallest person in a room by comparing two persons at a time. *Hint:* some persons will only have to be compared only once to any other person. Once you compare the shortest person with even one other person, for example, you will know enough never to compare that short person again with anyone.

1.