Fundamentals of Computer Science

2

Eric Puster

Learning Objectives
In this chapter, you will learn about:

- The basic science of computers
- Common vocabulary used by computer programmers
- The building blocks of computer code
- The general approach to solving a problem using a program, computational thinking
- Programming best practices to be used in generating code
- The framework for preparing software for use
- How to read code made by others and identify common problems

Practice Domains: Tasks, Knowledge, and Skills

– K006. Computer programming fundamentals and computational thinking

Case Vignette
Complaints from clinicians and nurses about numerous preventative health maintenance alerts have risen to the top of the health system. A clinical informaticist is instructed to integrate them into a single workflow addressed by population health nurses. The plan, as devised, involves creating a list that shows the patients with the most alerts at the top, along with contact information and barriers to care. When the patient's needs have been addressed, their name drops off the list to allow the next patient to rise to the top. The informaticist meets with the lead programmer for an early design meeting and is told that this system cannot be created. When asked why, the programmer cites privacy concerns, database structure limitations, and processing capacity; what technical challenges might prevent such a system from being created? What practical constraints will such a system be forced to obey? Why hasn't a system like this already been deployed everywhere in the United States?

Introduction

Long, long ago, automatic weaving looms were created to make the patterned fabric to replace armies of poorly paid weavers. At first, a good deal of human help was needed to reconfigure the machine for each new pattern, but humans were far from perfect for this process. To prevent waste, the loom operators began using punched cards to instruct their looms which weaving pattern to use when creating each bolt of fabric. The common elements of the various patterns were broken down to a binary code, as each position on a card was either punched or not punched.

Hematologists took hold of this idea in 1952 and began using punched cards to represent patient cases. They then used these cards to create cross-references for certain disease characteristics, enabling more accurate differential diagnoses based on those characteristics. At first, humans compared the cards, but in 1961, a computer began to be used for the purpose. This was the beginning of computers in medicine [1]. Before long, computers were used to store data about individual patients, predict diagnosis based on clinical observations, and send orders electronically. The electronic health record system was born.

Configuring those systems is one of the most challenging tasks in the world of computing. Not only can a slight mistake mean life or death, but the designers of software must contend with government policy, hospital politics, inadequate funding, changing standards of care, and the need for the software to communicate with a host of outside systems.

Fortunately, since the invention of the discipline of computer science, quality assurance techniques have developed. The definition of **Software Quality** has reached a mature international standard. The process of **Computational**

E. Puster (✉)
RTI International, Durham, NC, USA
e-mail: epuster@rti.org

J. T. Finnell, B. E. Dixon (eds.), *Clinical Informatics Study Guide*, https://doi.org/10.1007/978-3-030-93765-2_2

Thinking for breaking down a real-world problem into programmable steps has been thoroughly studied. And **Coding Best Practices**, an informal set of guidelines to achieve the quality standard, are widely available.

Unfortunately, however, programmers are not trained in medicine. Even those with a long experience in **Health Information Technology** (HIT) are not kept abreast of the latest changes and generally know little of how medicine is truly practiced. For this reason, they must confer with clinicians who have a foot in both worlds, who speak both languages. If clinicians and researchers can communicate their world-changing ideas to HIT professionals, the world will change.

Programming and Computational Thinking

Generally speaking, for a computer to perform any task, even the most banal, we must boldly depart from the familiar shores of standard human thinking and step into a world of pure logic. This does not mean "smarter logic." One common misconception is that a computer is "smart" because of how well it can complete various tasks. In reality, computers are quite the opposite, blindly following every instruction given with no thought to the consequences unless taught exactly what to look for. The task of a **computer programmer** is to instruct a computer to perform a task correctly, covering every possible contingency in a reasonable amount of time, using limited computing resources.

There are exceptions to this micromanaging approach. Artificial Intelligence aims to task the computer to build its own logic and does not follow the rules set forth here. This topic is covered in Chap. 16.

Computer Primer

Before we can delve into the wonderful world of computers, we need to develop a shared understanding of some important terms. As in every profession, programmers have developed their lexicon to facilitate their work. Only a portion of these terms will be included here, but they should be sufficient to at least discuss computer science concepts.

The Von Neumann Model

Von Neumann was an early pioneer in electronic computing, and his simple model [2] for the parts of a computer still holds mostly true today. The names of these parts will be used liberally throughout this chapter.

Input: Information from the external world. Obtained from devices like keyboards and mice.

Memory: Stores data for use by the computer. This includes Random Access Memory (RAM) and Solid-State Drives (SSD). The reason for different types of memory is that the closer they are to the heart of the machine, the faster they are, but the smaller in capacity.

Central Processing Unit (CPU) Control Unit: Follows instructions to move data, tell the logic unit what to do with it, and deal with input and output data. Modern high-powered computers may have more than one.

CPU Logic Unit: Carries out operations on the data from inputs and memory. The speed of a CPU clock (3 GHz) refers to the maximum number of commands this unit can perform each second (3 GHz = 3 billion commands/second). Modern computers usually have several.

Output: Information sent outside the computer. Monitors and printers are common devices that use computer output.

Programming Terms

Although different programming languages vary widely, some concepts are common between them.

Variable Just like in Algebra, a **variable** represents a piece of data that is determined and re-determined as the process goes on. The variable is created with a specific line of code called a **declaration**. This line of code is a command that names the variable, may give it a starting value and type, and instructs the computer to reserve a little space in memory for it. If the value is ever changed during the program, the computer overwrites the value in memory with the new value.

There are many variable types, but a few important ones are the integer (such as 10, abbreviated **int**), the floating-point number (such as 10.263, abbreviated **float**), and the boolean (TRUE or 1, and FALSE or 0, abbreviated **bool**).

Function A **function** is a specific group of commands that a program may need to perform many times. A common function is the "=" on the keypad of a calculator. When the user presses this function button, a **function call** is initiated. The calculator takes whatever is on the screen, the operator such as + or –, and whatever the user enters afterward, combines these inputs and **returns** the result to the calculator screen.

A function in a computer program may be a single line or a million lines, but they all take inputs, called **arguments**, and return a result. A function is also **declared**, just as a variable is, and stored in memory. In the C programming language (a very popular language), functions are declared like this: **return function(arguments)**. An example of a function in C is **sqrt()**, which calculates the square root of a number. The function call for the square root of 4 would be written **x**

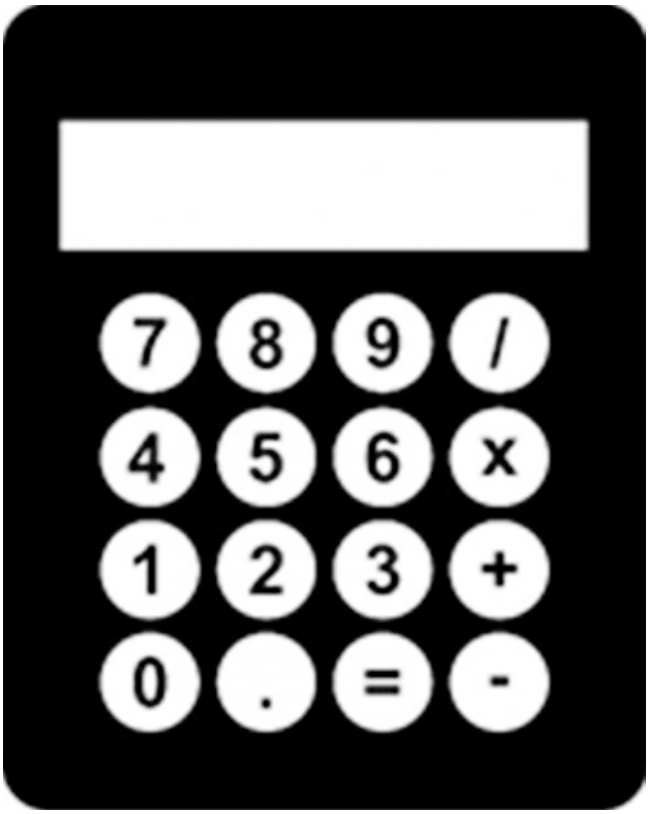

Fig. 2.1 An example calculator

= sqrt(4), where the answer, 2, would be returned into the variable **x** (Fig. 2.1).

Array An **array** represents a series of variables or values in a specific order. For instance, the line at the ice cream shop might be "Frank" then "Jane" then "Lucy", or **hungry_patrons = ["Frank", "Jane", "Lucy"]** written programming style. To look at a specific person in line, square brackets are used. So **hungry_patrons[0]** would be "Frank" and **hungry_patrons[1]** would be "Jane". Most languages have a prewritten function for finding the number of elements in an array. In C, it is called **size()** and is a function built into how an array works. So, for our example array, **hungry_patrons.size()** would be 3. The dot between **hungry_patrons** and **size()** shows that the **size()** function is a built-in function for arrays rather than a separate one.

String A **string** is a specific type of array made up of single characters. For instance, **"Ice Cream Shop"** is an array of 14 characters (even the space has a character code). Since **hungry_patrons** is an array of strings, and a string is a type of array, **hungry_patrons** is an array of three arrays and could be written **[["F", "r", "a", "n", "k"], ["J", "a", "n", "e"], ["L", "u", "c", "y"]]**. Strings are usually identified by being in single- or double quotes.

```
your_car = {
    type: "car",
    year: 2016,
    color: "teal"
}
```

Key

Value

Fig. 2.2 An example JavaScript object

Object Many languages use **objects**, which are variables that contain key/value pairs, functions, and even other variables. **Key/value pairs** in an object are in no particular order, so they cannot be accessed in the same way as an array. For instance, **your_car.type** or **your_car[type]** might be "sedan", while **your_car[1]** would make no sense to the computer, just as **hungry_patrons[type]** would also have no meaning. The curly brackets in the example tell the computer that everything inside the **{}** is part of **your_car**. A note: for compactness, a JavaScript object is used here, rather than C (Fig. 2.2).

Computer Language

Now that we have discussed some words for talking *about* computers, we need to discuss some words for talking *to* computers. Before a computer can execute even a single task, it must understand what it is asked to do. And because a computer makes no assumptions, everything must be spelled out to the finest detail. This is done, line by line, by laying out instructions much like a recipe. The computer reads the recipe from start to finish, performing each step. These recipes are computer **programs**. In modern programming, part of the work of programming is done by humans and part by algorithms. Before we can talk about computer instructions, however, we need to look at the alphabet used by a computer.

Binary

English has 26 individual characters to represent its spoken language, and many more if we include mathematical and scientific language. In contrast, there are just two characters in the language of computers, the base 2 number system 0 and 1. The language of the computer is built from only these two characters, and each 0 or 1 is called a "bit." The bit itself arises from an electrical impulse (but how this happens is beyond the scope of this chapter.)

A word about words: A **word** in the computer sense is a sequence of bits, not a word as in the English language sense. Bits combine to form a number, which may represent that integer exactly, or perhaps a floating-point number, or even a letter from the English alphabet. A computer will use a translation table to understand what the combination of bits represents. The most common translation table in modern applications is **Unicode** [3], so-called because it attempts to unify all possible symbols, including Asian logographic languages. In Unicode, "Hello" would be represented by the integer sequence 1032 1541 1548 1548 1551.

Fig. 2.3 A Jacquard Loom

Four bits make a nibble, eight bits make a byte, and sixteen bits make a word. In early computers, a nibble or a byte (depending on the system) comprised a single "instruction" or "number" or "idea." Complexity has grown since then, and in modern computers, the basic unit of communication is arguably the **quadword**, which comprises a sequence of 64 0's and 1's. This is the 64-bit system in commercially available machines.

Machine Code

The punched cards of the weaving loom mentioned at the beginning of the chapter could be thought of as a **machine code** to create a recipe or **program** to weave the fabric. Humans imprinted a binary code on a card ("hole" or "no hole"), which was then read by the loom to produce each successive row in a piece of fabric. It is referred to as a "code" because the card is not written in normal human language (Fig. 2.3).

Modern programmers do not generally look at machine code. This is because programs written for computers today are so complex compared with the simplicity of marking lines of fabric. The algorithms for reducing the program to the smallest size are so advanced that the code becomes difficult to translate back into a human-readable form. Translating machine code is time-consuming and expensive and contemplated only for very specific purposes, such as catching cybercriminals.

The smallest units of **machine code** refer to a single, basic **instruction**, such as fetching a number from memory, performing a logic function on two numbers, or storing a number in memory. To make things more complicated, every processor brand might have its own set of instruction codes or **Instruction Set Architecture** (ISA). For humans to work on a program, something more abstract would be helpful.

In one popular machine code [4], instructions for adding two numbers together might look like this (depending on variants and hardware).

```
00000000000100000000000010000011000000000010000000000000100000011
00000000000100010000000111000011
```

Assembly Language

The next step in simplification or abstraction uses readable human language to represent each step in a process. This level is at least understandable, even though we still need many lines of **assembly language** to perform even simple programs. To translate assembly language into machine code, programmers utilize a software tool called an **Assembler**. **Assemblers** use information about the instruction set and the machine's architecture to convert each instruction to binary.

Assembly also allows for other clever tricks, such as comments (small notes that help the programmer remember what part of the program does), some error checking, and allowing "labels" to name variables something memorable, such as "ans" for the answer to the addition problem. At one point in time, this was the level at which most programmers operated, but the need to produce more code more quickly and more reliably pushed the field to the next level.

The same instructions for adding two numbers together as above, but in Assembly language, might look something like this:

```
LD t1 0x00000001 ;Load Memory location 1 into t1
LD t2 0x00000002 ;Load Memory location 2 into t2
ADD ans, t2, t1 ;Add t1 to t2, store the answer in ans
```

For those curious, LD is the command "Load Doubleword" that loads data from memory. "0x" means hexadecimal notation, which can be thought of as a shorthand for binary. In assembly, comments start with a semi-colon; everything on the line afterward is ignored by the computer. In C, "//" denotes a comment.

Compiled Language

The next level of abstraction involved moving toward commands that are easier to read and contain more than one Assembly instruction within them using a **compiler**. What earned the compiler its name is how it converts the **compiled language** into machine code using multiple optimizers and error-checkers in various sequences, compiling the changes on top of each other. Each line of code is read into the compiler, and these statements are, in turn, broken down, analyzed, optimized, broken down further, checked for errors, etc., and finally written into machine code. The result is that complex functions can be represented in very compact statements understandable to humans.

Adding two numbers together in C looks something like this:

```
ans = t2 + t1;
```

This example is much easier to understand and removes many spots where a human might make an error.

Operators: C-like languages use several symbols for logical and arithmetic operations. Some important ones to know are shown here (Table 2.1).

As in math, there is an order of operations in computer programs. Note the difference between "=" and "==". A computer will not figure out which was meant by the programmer, which is a frequent source of bugs.

Table 2.1 Common operators in the C programming language

x + y	Return the sum of x and y	x – y	Return the difference of x and y
x * y	Return the product x and y	x / y	Return the result of x divided by y
x = y	Store a copy of y in x	x % y	Divide x by y, return the remainder
x || y	If x OR y is TRUE, return TRUE	x == y	If x and y are the same, TRUE
x && y	If x AND y are TRUE, return TRUE	x > y	If x is larger than y, TRUE
x >= y	If x is larger than or equal to y, TRUE	x != y	If x and y are not the same, TRUE
!x	If x is TRUE, return FALSE; otherwise, return TRUE (or "1")	x++	Store x+1 in x
x[y]	Return the yth value in x	x << y	Multiply x by 2 to the yth power

Table 2.2 TRUE vs. FALSE examples

Variable value	Is there something there?
0	FALSE
–1	TRUE
["Frank", "Jane", "Lucy"]	TRUE
"False"	TRUE
4 – 4 (the result of 4 minus 4)	FALSE

Compiled language also enables **code libraries** using **linkers**, which allows the user to call on common and very well-tested pre-written code to perform complicated functions. For example, to find a certain string of characters in a document of any size:

```
index = strstr(document, "you found me")
```

After this executes, the variable **index** will hold the numbered position in the string **document** where the string "you found me" first occurs. And because the code library has been reviewed and optimized many times over, the programmer does not need to worry (much) about hidden bugs. They are ready to move on to build the next part of their program. How could we possibly do any better?

What is TRUE? This brings up the side topic of what is TRUE and FALSE. These boolean logic ideas show up frequently in programming and depend somewhat on what programming language is used. Generally speaking, FALSE is equal to "0" and any value that is not FALSE is TRUE. Another way to phrase the question is, "Is there something in the variable?" This leads to some confusion for humans but makes perfect sense to the computer (Table 2.2).

Interpretive Language

Compiling a program takes time, and any given compiled program is almost 100% guaranteed to have at least one bug, meaning it will need to be changed and compiled again. Eventually, this time adds up. Some programmers thought it would be nice to run code without having to compile it. This idea led to the creation of **interpreted languages**. **Interpreted languages** use "just-in-time" compiling to run each instruction right when needed, without waiting for the compiler. This allows for quick adjustments to the program, **dynamic typing** (meaning the interpreter will figure out the type of a variable without being told), as well as the ability to "emulate" the software, meaning the ability to show what the result will look like to a user after each change in the code is made.

The most popular languages in use today are only interpreted, such as JavaScript and Python, or are compiled with an option for an interpreter, such as Java and C. There are many commonalities between these different languages. In this text, we will represent concepts with C-like structures unless otherwise specified.

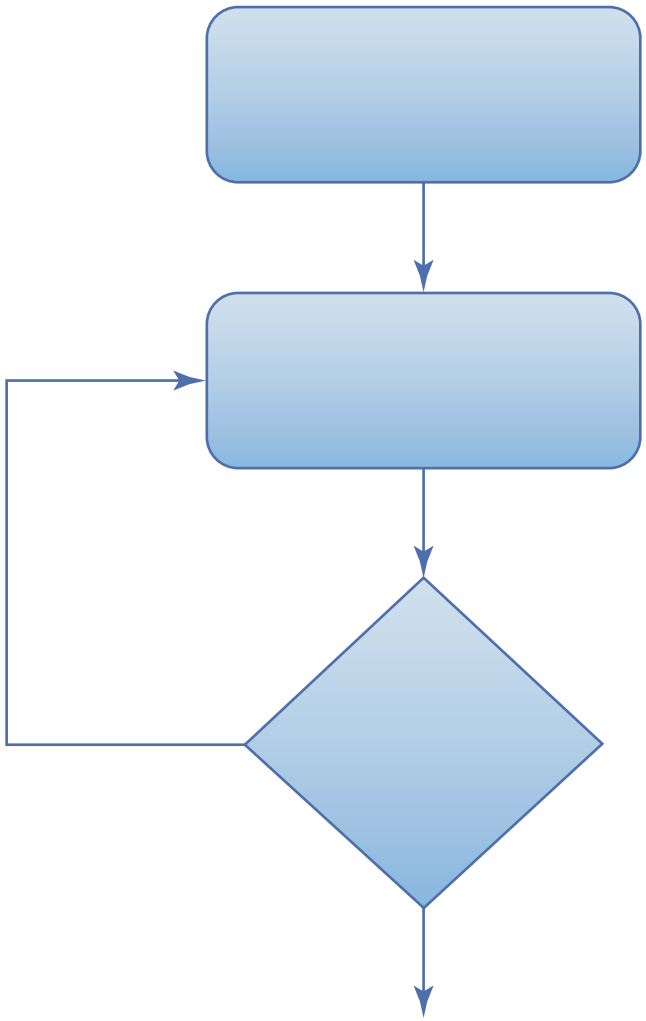

Fig. 2.4 An example flowchart with sequential, conditional and iterative elements

Control Structures

In a process flow diagram or "decision flow," each step in the process has a little box describing what it does. Working very much like those blocks, the building blocks of a program are called **Control Structures**. In programming, a control structure may also include other blocks inside of itself to break down the process further. These structures take one of four general forms: sequential blocks, conditional blocks, iterative blocks, and recursive blocks (Fig. 2.4).

Sequential

A sequential block executes a series of specific instructions in order. This is the default mode for most programming languages.

```
get_in_line();
buy_ice_cream();
eat_ice_cream();
```

After each step completes, the program marches on to the next instruction. In C-like programs, a semi-colon notes the end of an instruction.

Conditional (If/Else, Switch)

A conditional block executes several possible sets of instructions depending on the answer to a TRUE/FALSE question.

```
if ( have_flavor("chocolate") ) {
  buy_ice_cream("chocolate");
} else {
  buy_ice_cream("vanilla");
}
```

In this example, if h**ave_flavor("chocolate")** returns **TRUE,** then we will call b**uy_ice_cream("chocolate")**. If **have_flavor("chocolate")** returns **FALSE**, then we will call **buy_ice_cream("vanilla")**. A second common conditional control structure is the "switch." A "switch" command goes beyond just TRUE and FALSE, checking the value of a variable called "flavor" and deciding on a response.

One other important kind of conditional structure is the **exception**. In some languages, an exception is **thrown** if some section of code has an error. A separate section of code

```
switch (flavor) {
  case "vanilla":            // If flavor ==
"vanilla", start here
    eat_ice_cream();
    break;                 // This skips the
other cases
  case "chocolate":          // If flavor ==
"chocolate", start here
    savor_ice_cream();
    break;
  case "mint":               // If flavor ==
"mint", start here
    make_ugly_face();      // No offense to
mint-lovers
    break;
}
```

called an **exception handler** can take that exception and do something about it. For example, for the above code, consider the case that there is no ice cream. Then, "flavor" is not valid, and an exception is thrown, which might trigger an attempt to get ice cream and try again, or else call the **make_ugly_face()** function.

Iterative (While, for)

Consider the problem of looking through a list of ice cream flavors to see if the shop has chocolate. We do not know how many flavors there might be, so it is hard to write a sequence of conditional blocks to check through it. Instead, we can use an **iterative structure** to repeat the same sequence of instructions repeatedly until we conclude.

The **while loop** executes a series of instructions, then asks a question to see if it can stop:

```
while (are_we_there_yet == FALSE) {// "Are
we there yet?" "Nope"
  drive();                          // Drive
down the road a bit
  eat_chips();                        // Eat
some chips
  fiddle_with_AC();                 // Try to
get temp right
}                                  // Time to
ask again . . .
```

This loop will not stop, ever, until the condition a**re_we_there_yet == TRUE**. So, setting this loop aside and going back to our ice cream example, we could create a **while loop** that scans each group of characters to see if it is "Chocolate". If there is a match, we set a variable **are_we_there_yet** to **TRUE** to tell the loop to stop, and another variable, such as **found_it**, to **TRUE** to show we found it. If we hit the end of the sign without finding chocolate, we also set a**re_we_there_yet** to **TRUE** but in this case, set **found_it** to **FALSE**. If we are not done, we move to the next entry and check again.

A **for loop** is much like a while statement, except that some of this work is right in the first line. Consider the following:

```
found_it = FALSE;
for (int position = 0; position < ice_cream_
sign.size();
position++) {
       if    (ice_cream_sign[position]    ==
"Chocolate") {
    found_it = TRUE;
    position = ice_cream_sign.size();
  }
}
```

The first thing we do is assert we have not yet found chocolate and set **found_it** to **FALSE**. Next, we start the for loop, which has three parts separated by semicolons. The first part is the **initialize** step, which sets the variable **position** equal to 0. The for loop then makes a **test**: "Is **position** less than the size of the **ice_cream_sign** array?" If so, we continue. The last part is the **update** which tells the computer what to do after each iteration finishes. In our case, the variable **position** is increased by one to check the next spot in **ice_cream_sign**. In this way, we iterate over each position in the **ice_cream_sign** array, one at a time.

During each iteration, we check the **positionth** spot in **ice_cream_sign** to see if "Chocolate" is there. If that check returns **TRUE**, we set **found_it** to **TRUE** to mark our success, and we set **position** to a value which will ensure that the loop does not run again.

Where do we start? An important note is that if we did want to start at the beginning of a document and run through to the end, in C-like languages, the first character would be at position "0", not position "1". The reason for this is beyond the scope of this book. Starting with "0" is called "zero-indexing" and is another frequent bug-maker. The position "1" in "Hello" is "e", not "H", as might seem more natural.

Loops can present serious problems in code and are one of the most frequent sources of bugs. This will be discussed more in-depth near the end of this chapter. For now, let us consider the question of the **nested loop**. Consider this code,

meant to change every entry in **myarray** into all lowercase "z", character by character.

```
myarray = ["Car", "Red", "2016"]
// Start by iterating through each element,
like "Car"
for (int word = 0; word < myarray.size();
word++) {

  // Inner loop based on the length of the
element, so 3 for
"Car"
   for (int letter = 0; letter < myarray[
word ].size();
letter++) {

     // Set the "letter"th position of the
"word"th element to
"z"
    myarray[ word[ letter ] ] = "z";
  }
}
```

The nested structure allows the loop to cycle through each word in the array and then through each letter of each word and replace each with "z", no matter how many words or how long they are. This structure is very powerful, but the more layers, the harder it is to understand and fix if it is not working correctly.

Recursion

Recursion is not always considered a control structure, but it can do tasks that no combination of the other mentioned control structures could. In essence, a recursive process calls itself, sometimes many times, to reach an answer. An easy-to-understand (though gratuitous) example is that of the factorial from mathematics.

```
int factorial(int x) {
  if (x == 1) {return x;}
  else {return x * factorial(x - 1);}
}
```

Let us step through this, calling **factorial(3)**. The function checks to see if 3 == 1. It does not, so instead, it tries to return 3 multiplied by a function call. At this point, the program puts **factorial(3)** on the bench and calls **factorial(3 - 1)**. This function starts by checking to see if 2 == 1. It does not. So instead, the program calculates 2 multiplied by **factorial(1),** which requires putting **factorial(2)** on the bench in memory to wait for the result of **factorial(1)**. Finally, **factorial(1)** sees that **x** does equal 1 and returns 1. **Factorial(2)** comes off the bench and returns 2 times 1 a.k.a. 2, and **factorial(3)** then returns 3 times 2 a.k.a. 6.

However, keep in mind that each function call waiting to return consumes a piece of memory (this has to do with the programming stack, which is beyond the scope of this chapter). Calculating **factorial(1000)** this way may gradually choke the processor as the program stores each successive function call on a slower, more distant memory. There are other processes, like searching an organization chart for a particular person, where recursion is, by far, the most elegant approach.

To present this process, we need to introduce another data structure commonly used in programming: the "tree." In a tree, there are a series of entries linked together through parent-child relationships. A parent entry contains information about itself and an array of links to all its children. The start of a tree is the highest entry—the one that has no parent. In the diagram, you can see a tree with the highest entry named "Alice." Our recursive function will have the task of finding the entry named "Charlie" and will do so in a "depth-first" fashion, meaning that it will check if it has found the answer, and if not, will call itself on each of the children.

Call #1 examines the highest entry where it finds Alice is not Charlie and calls itself on the first link in Alice, the one pointing down to Bob. This function call is set aside in memory while function call #2 takes the stage. This call sees that Bob is not Charlie either and calls itself on the first child of Bob, Dan. This function, call #3, sees that Dan is not Charlie and that Dan has no children. It returns FALSE, meaning it finished without finding Charlie (Fig. 2.5).

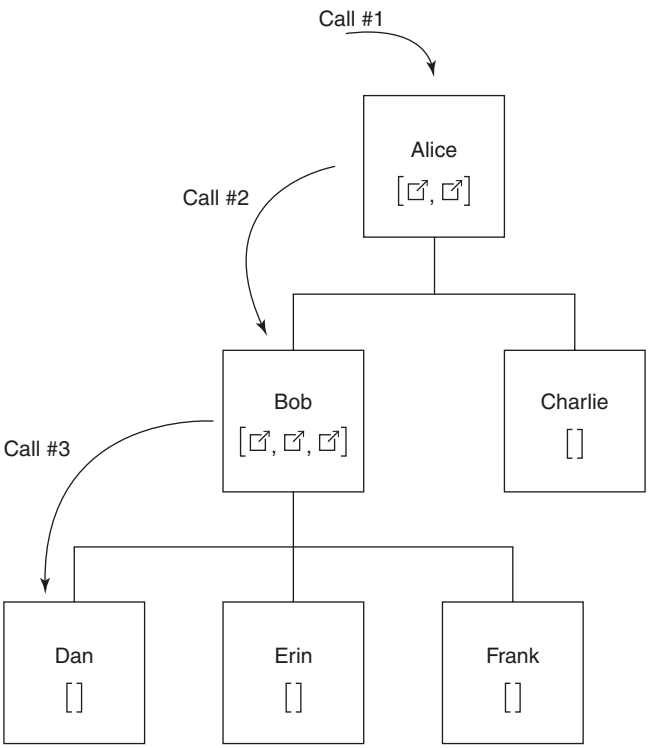

Fig. 2.5 Recursive function calls in a data tree

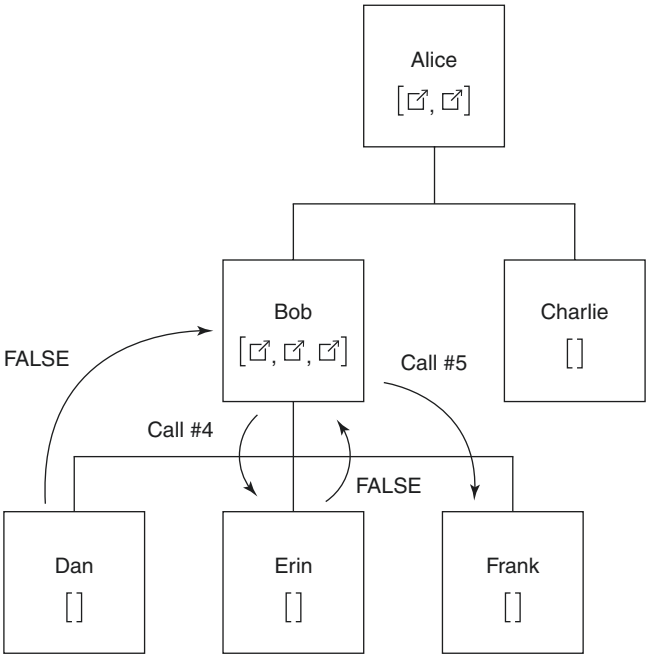

Fig. 2.6 Checking each of the children

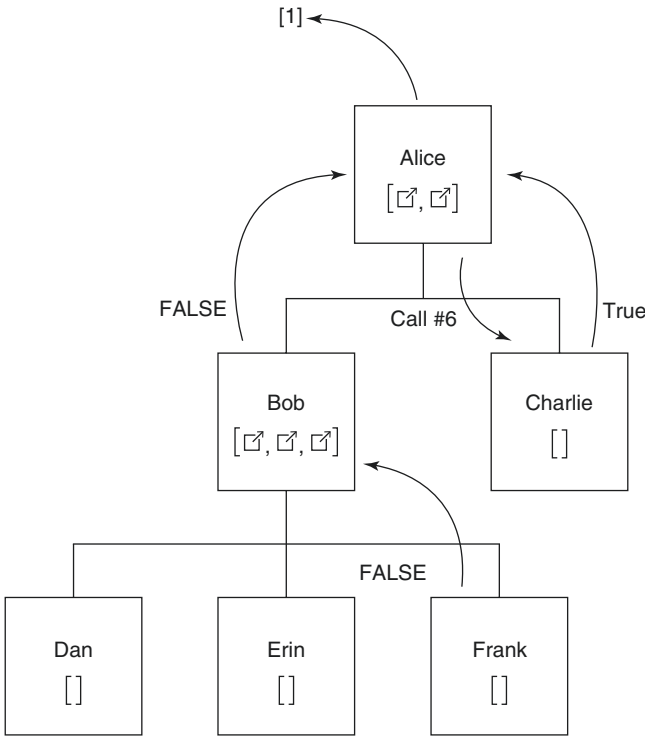

Fig. 2.7 Returning the result

At this point, a normal function would be stuck. There is no link back to Bob inside Dan. But a recursive function can overcome this because function call #2 is still sitting inside the Bob entry, waiting to continue. Function call #2 wakes back up to get the FALSE return signal from call #3, sees that there is another child of Bob, and calls #4 on Erin. This also returns FALSE, so call #5 is made on Frank. This again returns FALSE (Fig. 2.6).

At this point, call #2 gives up and returns FALSE back to call #1. Call #1 sees that Alice has another child, and so calls #6 on Charlie. Call #6 quickly finds it has found Charlie and returns TRUE. Call #1 knows that when one of its calls returns TRUE, it should send back the index in the array of children; in this case, 1 (0 was Bob). Since there was only one step, an array with a single entry [1] is the final return value for the function. If the function was searching for "Frank," it would have returned [0,2] to show the path: the first link in Alice (remember 0 is the first one!), then the third link in Bob (Fig. 2.7).

Computational Thinking

We have examined some of the common tools used to make things happen in programs, and we have also seen that solving a problem using a program can be challenging and error-prone. Experience over the decades has produced a few common approaches to reach the end goal of **Software Quality**, defined by the International Standards Organization (ISO) as [5]:

1. Functional Suitability: Gets the right result.
2. Performance Efficiency: Gets there in a reasonable time using few resources.
3. Compatibility: Friendly towards other software.
4. Usability: Minimizes user frustration.
5. Reliability: Does not crash the computer or light things on fire.
6. Security: Cannot be misused by bad actors or unwise users.
7. Maintainability: Can be understood/updated by the next programmer (especially oneself.)
8. Portability: Can be moved or replaced easily.

Computational Thinking focuses on the first part: getting the right result. But it also considers many of the other factors in software quality. Denning [6] defines computational thinking as "the mental skills and practices for designing computations that get computers to do jobs for us, and explaining and interpreting the world as a complex of information processes." We will discuss a few techniques along these lines, thinking about a system designed to irrigate a garden.

Specification

The first step in programming involves no computer code, and strictly speaking, it is outside the box of Computational Thinking. We must first identify precisely is the question we are going to answer. This includes mechanical questions, such as the size of the garden, and computational ones, such

as how to decide if watering is needed and how much. Answering these questions upfront will save time by creating a solution that covers what is needed and no more. We will address the subject of specification more in-depth later in the chapter. Here is the start of our specification.

Aims:
- Primary: To encourage the growth of corn in a garden plot using irrigation.
  - Secondary: To minimize water waste.
  - Tertiary: To minimize user interaction.

Unanswered Questions:
- How do we decide how much water is needed?
  - How does the system dispense the water?

Decomposition

To accomplish our aims, we need to break them down into some specific, solvable problems. This process is called **decomposition**. As the program solves each of the problems, it returns the result to the main program. When all results are in, the aims are reached. Turning our attention to irrigation, there are a few tasks that make up the overall aims:

1. Calculate total water needed per day.
2. Calculate total water supplied by other means today.
3. Calculate how far to open the tap to spread that water flow over 24 h.
4. Open the valve.

The reader may have noticed an assumption: we are meant to spread the water flow over 24 h. Assumptions are the bane of programmers—do not assume anything if possible. Whenever feasible, nail down every fact in the specification. Some assumptions can be unavoidable: we might assume that the water supply has water, that the hoses are connected and not leaking, that the valve opener has power connected, etc. In these cases, we still want to identify each assumption to build exception handlers or other systems to reach the best end we can (perhaps by sending a message to the user or using a backup system).

The reader may also have noticed that each of these steps requires more decomposition, especially how we will figure out how to calculate the total of other sources of water. Here is the first attempt in pseudocode.

1. Access a weather internet site.
2. Find information about weather prediction for today.
3. Scan for rainfall prediction.
4. Read rainfall prediction for the day into a variable.

Pseudocode

Pseudocode is not another programming language. It is like an author outlining their book before they begin writing it. To make pseudocode, a programmer describes roughly what they want to accomplish with each code section to complete the solution.

Another critical assumption was just made: the system will have access to the internet. This will need to be added to the specification, or another way to predict the rainfall will need to be used.

Abstraction

There may be other water sources, such as animal life, sprinklers not intended to hit the garden, pipe leakage, the sudden eruption of an artesian well, or simply a poor prediction from the meteorologist. However, these are not important enough or easy enough to predict to be worth the trouble. The problem can be simplified to:

"How much water do I need today."

- "How much will fall from the sky today."

"Total water to dispense today."

This is an **abstraction**. It simplifies a problem from an unanswerable question to an answerable one.

Whether or not to use an **abstraction** is often a matter of art, estimating the likely effect to decide whether it should be included in the model. This is not specific to the world of computing, being common to physics, statistics, and informatics.

A **function** is another form of abstraction. For instance, in **C**, **sqrt()** is a function that calculates the square root of a number. The programmer does not need to understand Newton's method for square roots to find one, they just call the function, and the correct answer is delivered to them.

This function is part of the **C** code **library**, a group of widely used functions that have been written and thoroughly tested.

Pattern Recognition

Recognizing patterns in a problem allows us to create an abstraction (such as a function or a loop) to handle the issue every time it appears, rather than typing the code all over again. In our irrigation example, we assigned 24 h to the irrigation. Why? Consider the pattern of updates on meteorological estimates. At least once per day, a new prediction is made. The closer we are to the period of time being predicted, the more accurate our calculation is.

Recognizing this pattern makes our solution more likely to give us the right answer. After all, we might simply spread the yearly average rainfall over 9 months. But then we would be farther off the mark every day during the 9 months.

Also, each of the steps of calculation needs to be solved multiple times. As such, they should be in functions or loops that can be run over and over, rather than a single million-line program that repeats almost character for character at the start of each new day during the 9 months.

Parallel Processing

In years past, the computer had a single processor that needed to execute each command in order. If a command had to wait for half a second for information to be fetched from the hard drive, everything would come to a standstill for the full half-second, even if the next command did not require that data yet. Since that time, hardware and software designers have enabled a way to overcome this obstacle through parallel processing.

Think about a racetrack where runners all run on a single lane. Each racer must finish before the next can begin. Adding parallel tracks makes the race much more exciting (and takes less time) as all the racers can compete simultaneously. Software optimizers or programmers identify sections of code that can run simultaneously without interfering with each other and mark them. The computer then turns these sections of code into a series of executable statements called **threads**. Then, while thread #1 waits for the slow hard drive to respond, thread #2 can run its commands. Increasing the number of **cores** or **processors** in a single computer allows for more and more threads to run simultaneously.

This kind of computing is also often called **asynchronous**. It increases performance, but this parallel competition also makes a new type of bug possible called a **race condition** if the threads are not truly independent. One example might be a news website that wants a user to pay before they see an article. Two threads are started: one to check to see if the user is a paying customer, the other thread to load the content. The race is on! If the programmer is not careful, the "content loading" thread might show the article before the "paying customer check" thread finishes, letting the user get the article for free.

Parallel vs. Asynchronous: These two terms are often interchanged but are not the same. "Parallel" refers to two threads executing simultaneously without waiting for the other to finish. "Asynchronous" also refers to threads running simultaneously without waiting but more specifically refers to the two threads not having to work according to the same rhythm. For example, if a laptop with a slow processor is playing a game on a website, and the website server is preparing the next level of the game using a faster processor, the laptop and the server are working together, but each to a different rhythm.

In our irrigation example, finding the prediction for daily rainfall and calculating the total water needed are separate, isolated calculations. The order in which they occur does not matter. They both access one piece of data (the date), but they do not change it, so running these two commands in parallel (or asynchronously) works well. However, both need to finish before calculating the amount of water to dispense; otherwise, we will cause a race condition and undesirable results. (In the worst case, an error that causes the program to stop completely or dispense an infinite amount of water, or at best, the same amount of water each day despite what the prediction is).

Algorithm Design

The word **algorithm** is often used to describe parts of programs, but they are not the same. An algorithm is a set of instructions to complete a calculation, such as determining how much water to dispense. But the program as a whole must also operate a valve and perform other functions. In this way, an algorithm is a part of a program and requires its own consideration.

In our garden irrigation system, the algorithm appears simple, but we could consider plenty of modifications that would introduce more subtleties. For instance, we might average the predictions from several websites or use a weighted historical average using the usual rainfall in past years. We might want to consider the price of water at different times of the day or measure the user's habits to predict the likelihood that the water flow will cause problems with water pressure (i.e., someone showering in the house.) We could also consider the time of day, watering more at night than when the sun is up. The program could even take minute-by-minute rainfall measures to predict how much water will be needed for the rest of the day.

Each of these could bring benefits to our aims but could also subvert them. For example, determining when the user will need water for other things might mean more user interaction, which we do not want. On the other hand, averaging the predictions of multiple websites would help a great deal if one of the websites were to go down. We also need to consider the available inputs and outputs, the computing resources at hand, and the end goal in mind. A garden irrigation algorithm that is 100% accurate but requires a small university's computing resources is not useful.

Coding Best Practice

It is important to discuss the right way to cement these principles in code along these same lines. Even if the code reaches the end goal of giving us the right answer, there are right ways to code, and there are wrong ways to code. A quote attributed to Tom Cargill of Bell Labs [7]: "The first 90% of the code accounts for the first 90% of the development time. The remaining 10% of the code accounts for the other 90% of the development time." Coding projects may require 180% of the expected time due to code being written the wrong way, leading to rewrites and workarounds. Following best practices can make the difference between a software project being on headlines or headstones. Unfortunately, **coding best practice** is not outlined step-by-step in a position paper; rather, it is an informal set of rules generally followed by successful programmers. Here are some of the more prominent ones:

Commenting Almost all languages allow for commenting. This is the text written into a program that is ignored by the interpreter/compiler/assembler and is exclusively for the programmer's use. They should be written liberally. There are two reasons for commenting gratuitously: (1) writing down a chain of thought might reveal errors, and (2) code is awfully difficult to understand without it, even for the programmer who wrote it. Verbose comments are encouraged as a best practice.

Keep It Simple Eventually, the code will need to be reviewed, maybe even by the original programmer. If the code is understandable, it increases the chances of being updated or reused rather than scrapped. A more efficient way to solve a problem is not always better; others need to understand how it works.

Naming Conventions Along with the above, ensure that variables and functions are all named something meaningful. For example, **predicted_rainfall** is generally better than **pr**. Going further, it helps to have variables, and functions differentiated by differing use of underscores and capitals, such as lowercase and underscores for variables (i.e., **predicted_rainfall**) and specific capitalization and no underscores for functions (i.e., **dailyRainfallReader**()). The reader may also have noticed that there are no spaces in the names. Almost always, the compiler or interpreter has no way to know whether a space is intended to start a new piece of code or is another part of a name. There can be no ambiguity or assumptions in programming; the space represents the move to a new command or part of a command.

Modular Design When we decompose a problem into individual parts, we can then write solutions to those parts. These are modules. It is very important to segment these modules from each other because the inevitable bugs become much easier to deal with if we can isolate the module that causes them. This also allows for reusability, as a module that performs a task can be used repeatedly by different parts of a program. This saves the time of writing it again and prevents the new bugs that that would create.

Handle Garbage Gracefully A function must be written to handle any input from the user gracefully and without crashing. If invalid input is given (such as **"Golf"** or **-1** for rainfall in a day), the program must respond benignly. The best response points out the issue, such as **"Invalid input to calcVolume(): 'Golf'"** but anything other than crashing or causing other unpredictable behavior is preferred. This is also called **Programming Defensively** as if the programmer imagines that users and other programmers are going out of their way to cause trouble.

There are many, many more opinions and ideas about what constitutes coding best practice. See the sources at the end of the chapter for more information.

Operating System

Once a program is written, it needs a place to run. Many applications are now written to run on an internet browser, but all must eventually run on some kind of operating system. Windows is an example. The operating system is itself an abstraction. The full understanding of what an operating system is and what it does is beyond the scope of this text, but the simplified version is given here for context.

A computer is made of hardware: chips, capacitors, fans, etc. A tiny kernel runs inside the processor, directing the processing of instructions. On top of this runs the BIOS, which allows the processor to interact with the memory, the keyboard, and other basic devices. On top of the BIOS runs the Operating System, which provides a means for software to make hardware requests, send information to the internet, show graphics on the monitor, or load data from memory. It also ensures that processor time is shared fairly among all the applications and that all the applications behave well and

do not interfere. When a piece of software runs, processor time is supplied. All the particulars of how that happens are hidden from view.

Application

An application is a program, and it runs on the operating system, making requests of the hardware repeatedly, usually at the behest of the user. There are many popular examples of applications, from iTunes by Apple to Microsoft Teams. The most familiar applications are those that we install and show windows for us to interact with when we run them.

Background Applications Some applications are running even though there is no window showing on the screen. These are called "Background Applications" and include things such as the Notification app for Facebook. They may or may not announce their presence, and many background applications never interact with a user at all. **Push**-style background applications wait to be triggered (pushed) by other applications, while the **pull**-style pulls information automatically to know when to trigger (such as the time of day). Operating systems generally use dozens of such applications to perform their tasks. It is common for large applications to load most of themselves into faster memory in the background using a launcher application. When the user clicks to activate them, they can load more quickly.

Web Applications Some applications run on top of other applications. This is true of web applications, which do not run on the Operating System itself but rather on the internet browser (such as Safari or Chrome). Without the browser, the software cannot run. These applications require a different programming language because they make requests of another application rather than the Operating System, which brings up the subject of the API.

Application Programming Interface (API) Applications cannot tell each other what to do unless they specifically open them to communication. The specific commands and procedures for one program to communicate with another one constitute an API. In this way, applications can work together to accomplish something. Google Authentication has an API. Facebook and other applications use this when the user clicks on "Sign in Using Google." The application sends some information to Google's API to tell it who is trying to authenticate. Google asks for the password, checks it, and sends a success or failure message back to the application. If successful, the application logs the user in without ever seeing the password. Interoperability like this is crucial in medicine and is covered more in-depth in Chap. 13.

Beyond the Application

Many years ago, the environment for using a program consisted only of the user, the computer, and maybe some machinery operated directly by the computer, such as a printer. With the advent of computer networks and the internet, there came incredible opportunities for sharing data and information between computers. Modern programming employs what is often called the **client-server model**. In this model, the client computer interacts with the user, showing information and presenting buttons to push and fields to fill. The client also interacts with a server, asking for data and submitting the client's user's data. To organize this data, the server uses a database. In this way, when a client asks for data, the server can find the correct information quickly and supply it.

A simple example is the checkout of any web store. The checkout screen is part of a web application running on the user's computer or phone, presenting buttons and a display. The user enters a request to make a purchase of $100 using their credit card. The application then interacts with a server computer at the credit card company headquarters, asking if the user is authorized to make a payment and $100 available. The server performs two database queries, one to check out the credit card number, PIN, expiration date, etc., and the other to see if $100 is available. If both are valid, the server tells the database to credit $100 to the seller and replies to the web application that the request was successful. Finally, the web application shows the user that the transaction was successful.

Full-Stack Sometimes an individual describes themselves as a "full-stack" developer. This stack is distinct from the system stack mentioned earlier. What a full-stack developer is advertising is the ability to develop programs for serving users (client-side programming), serving applications (server-side programming), and serving data (database programming). They might say they do a particular kind of stack, such as a LAMP stack. This refers to a particular group of programming languages and development platforms. The LAMP stack is JavaScript (an interpretive language), **L**inux (an operating system), **A**pache (a server platform), **M**ySQL (a database platform), and **P**HP (a scripting language for websites). A full-stack application has a client, server, and database portion.

Databases Data is the bread and butter of informatics and is found most often in a database. Greater focus is placed on databases in other chapters of this text, so it will not belabor the subject here, only to say that databases use **declarative** programming languages. Structured Query Language (SQL) is an example and differs from the languages we have discussed up until now, because its lines of code are interpreted by the software running the database, not the CPU, to deter-

mine what action to take. Think about what happens when you use a search website. You declare what you are searching for, and the website returns the results. But every site makes its list of results differently because of its programming. Much the same way, declarative code makes a generic request, and the database code decides how to process it.

Preparing the Code for Use

Whether or not code is successful in use depends on how good it is, but it also depends on whether it fully addresses the problem it set out to solve. It also depends on the ability to be updated and fixed as the inevitable bugs begin to surface.

Specifications

Knowing what the code is supposed to do before the programmers get to work is crucial to success. There is no single standard method in which to prepare a software specification, but there are some common elements (Table 2.3):

Many of these elements are common to project design, which are covered in detail in Part IV of the book. Suffice it to say that without a specification, the software cannot be tested to conform to a specification, and therefore it is impossible to know if it is done or even safe for use.

Unit Testing

When each module is conceived, the method for testing it should also be considered before coding begins. Code architects create a test for each aspect of the program, referred to as a **unit test**, and the programmers write simple code to pass each test. This is called "Test-Driven Development" and has proven very effective in bringing a specification to life with as few assumptions as possible, especially in bigger projects [8]. The tools for this process are discussed in Chap. 12.

Table 2.3 Common elements in software program development

Authoring information	Who wrote the spec, and when they wrote this version
Aims	What the software is supposed to do from the perspective of the user.
Methods	What tools will be used, such as the programming language, the database platform, and the operating system?
Stakeholders	Who cares about whether the project succeeds or fails and those that influence the specifications?
Testing	How will we test that the program works and will continue to work in real-world settings?
Work Estimate	How much work will it take to get to the finish line?
Unanswered Questions	Project questions that need an answer, even those that involve no actual coding

Reading Other People's Code

Code does not read like a novel. It is designed to be understood by a computer, with no room for assumptions. In places, it will appear to be excessively lengthy, with many logic checks that seem unnecessary. In other places, a complicated step in the process may be obscured by a poorly named function that leaves the reader without any clue to the intended result. Comments may be present, but even these can be unreliable at times. As a result, reading code written by someone else is a daunting process.

The author knows no rigorous, well-studied process for reviewing code, despite being asked to do it many times. There are two main techniques to get started trying to understand a piece of code. The first involves writing in commands called **breakpoints** or using an interpreter to see exactly what the program is thinking at a given point in the process. For example, for the irrigation system, a programmer might insert a command to print the values of all the variables right before the code finishes. That command is just for debugging and forms a breakpoint. This should show how much rain is predicted to fall, the garden area, the website is used to get the information, etc.

If the code cannot be run, then the second technique is needed. First, look for something in the code with a known intent—for example, the command to read a rainfall prediction from a website, recognizable by the URL. Otherwise, there may be a particularly helpful comment or a function call to a well-known library function.

Starting from that well-understood line of code, work backward towards the beginning of the code section, and once everything from the beginning makes sense, work forward until the end of the code section. It may be helpful to add or update comments along the way as a reminder for those coming after.

Reading Other People's Code for Errors

Even more challenging is looking for errors in the code of others, especially if the program cannot be run. First, find out what the code is meant to do and how it failed. It may be valuable for the programmer to think about how they would approach the end goal of the code, as the error might be apparent by comparison.

With loops, it is useful to start to write out the value of the variables at the end of each iteration. This can reveal the pattern of advancement, and give an impression of what the data might look like 10 or 1000 iterations in (Table 2.4).

We need proceed no further to know where this will wind up.

Table 2.4 Working a loop by hand

	a	i	j

str a = "astronaut";
for (int i = 0; i < 1000; i++)
{
for (int j = 0; j < 5; j++) {
a = a + a[j];
}
a = a + " ";
}
	astronauta	0	0

	astronautas	0	1
	astronautast	0	2
	astronautastr	0	3
	astronautastro	0	4
	astronautastro a	1	0

Here are some of the most common errors, some of which have been mentioned before.

Off-by-One Zero-indexing throws humans for a loop. Consider the following code:

```
// Calculate the mean
array grades = [86, 72, 95, 100, 65, 92];
int sum = 0;
for (int i = 1; i <= grades.size(); i++) {
  sum = sum + grades[i]; // Add each grade
to the sum
}
int mean = sum / grades.size();
```

It may seem straightforward, but this code has two common errors, both relating to zero-indexing. It is in how the **for loop** is started: the variable **i** starts with a value of 1, so the first loop will add **grades[1]** to the sum. But **grades[0]** is the first number in the array, not **grades[1]**. The second error is left to the chapter exercises, which are available as an electronic download.

Unit Conversion Imperial/Metric conversion contributed to the crash of at least one space probe [9]. Consider this code:

```
// Calculate the right medicine dose
float weight = getPatientWeight();  // Prompt
nurse to enter
patient weight
// Access online database
float recDosePerWt = getRecDosePerWt("acetam
inophen");
return weight / recDosePerWt;
```

It first calls **getPatientWeight()** and stores the result in **weight**, then gets the dosing for acetaminophen by weight, and then returns the weight-adjusted dosing for a patient. This code appears innocuous, but it has a serious flaw. Will the nurse enter kg or lbs.? And what about the online source? Does it use kg or lbs.? There can be no assumptions in the code. Modern dose databases will provide the units along with the dose information, and this should be compared against the units entered by the nurse to ensure that they align, and if not, make the conversion. Finally, the units must be part of the returned value to avoid passing bad data back up the chain.

Infinite Loop Consider the following code:

```
// Count the appointments on my calendar
for the week
int day = 0;
int total_appts = 0;
int daily = 0;
while (day < 7) { // Stop when we reach the
7th day

  // Grab the number of appointments for the
day
  daily = getDayAppointments(day);
  total_appts = total_appts + daily; // Add
to the total
}
```

We start by setting some things to zero. Then we start our loop. We first fetch the number of appointments for the day, then add it to **total_appts** before going on to the next day. But we are missing a statement. The variable **day** is never actually changed, so we keep adding appointments from day 0 to **total_appts** over and over again into infinity. This loop will attempt to continue forever, consuming all computer resources in its quest for completion. This kind of bug is what usually causes the dreaded "freezing screen." There are times that an infinite loop is intentional, such as the loop to run the irrigation system every day until the end of time. But in that case, it would be instructed to wait until the next day before continuing.

Syntax Error This is a trivial error for a computer to detect but may be very challenging for a programmer. Consider the following:

Good, helpful comments describing the exact thought process are useless to identify the problem in this case. The line that declares the array "**array sieve = . . .**" does not have a semi-colon. That may seem like an innocuous issue, but the

```
// Check a number to see if prime under 100
int checkPrime(int num) {

   // if less than zero or more than 100,
return an error
  if (num < 1 || num > 100) {return -1;}

   // If not prime, one of these must be a
factor
  array sieve = [2, 3, 5, 7]

   for (int i = 0; i < 4; i++) { // Cycle
through each number in
sieve

// If the result is an integer, it can't be
prime
if (isInt(num / sieve[i])) {return FALSE;}
  }

   // If none of the numbers was a factor,
num is prime
  return TRUE;
}
```

result is that the computer sees a reserved token "for" for the **for loop** in the same line as the array declaration, and it does not know what to do. This will crash the program. Since the syntax is different in different kinds of code, the only hope when reviewing someone else's code is to look for patterns in variable declarations or line endings and try to see one that is not like the others.

Out-of-Bounds Anytime there is a logic check, make sure it is possible for the check to result TRUE or FALSE depending on what goes in. Consider the following:

```
// Eat a sandwich only if it is lunchtime
string meal = getCurrentMeal();
if (meal = "lunch") {eatSandwich();}
```

In this case, we get the name of the current meal and then run a logic check, which, if **TRUE**, results in eating a sandwich. But the logic check is not what it appears. To check if **meal** equals "**lunch**", we must use **meal == "lunch"**. The code shows the value of **meal** to **"lunch"** then checks to see if **meal** is **TRUE**. There is a string in the **meal** variable, so the result of the check is **TRUE,** and we **eatSandwich**(), no matter the time of day. Sometimes **out-of-bounds** occurs when an object or array is used instead of the intended attribute or property. For instance, **array < 5** might always return FALSE, while **array.size() < 5** will return TRUE if the array is small enough.

Another way to be **out-of-bounds** is called a **buffer overflow**. It was mentioned previously that variables have a specific place in memory, which is true of strings. Consider this code for reading the first ten characters of a string:

But what happens if **string** is less than ten characters

```
for (int i = 0; i < 10; i++) {
  // printf prints string[i] to the screen
  printf("%c", string[i])
}
```

long? This will cause the loop to start reading out what is in memory outside of **string**. As expected, this is undesirable and can be catastrophic if the program is not just reading but writing to those out-of-bounds memory locations that could contain other parts of the program.

Dirty Data When code provides for a user to enter something, it bears remembering that they may enter *anything* (mentioned above in "Handle Garbage Gracefully"). They may write "Cheese" for the year, or write programming code in the Chief Complaint (see "code injection" in another text), or even accidentally add whitespace to the beginning or end of their name. "Whitespace" here refers to characters in a string that do not appear on the printed page. The space itself is the most common example, but others the line feed, the carriage return (both for starting a new line), the tab, and the non-breaking space. These invisible characters must always be accounted for when using strings provided by a user.

Imagine a database for storing patient names. Suppose a careless assistant added a space at the end of a name. When the database is later searched for the patient's exact name (without the space at the end), the search will come up empty. The common way to prevent such problems is to **scrub** or **sanitize** the data to ensure that whatever the user enters, your program can clean it up enough to make sense of it and not crash. One slightly unusual emoji is sometimes enough to bring a massive database to its knees.

Recommended Resources/Tools

In this section, the author notes several external resources for further developing one's programming skills. The author has no financial interest in any of the resources.

Learning a Programming Language In the author's opinion, the best way to learn how to program is to find a problem the budding programmer cares about, then find a simple tool that does part of the work and builds on it. There are many free resources to learn about any given language. Some prominent ones are:

- www.learn-c.org: A member of a group of websites for learning C, JavaScript, Python, SQL, and others.
- www.cppreference.com: A reference for C/C++.
- www.w3schools.com: A reference for web development.
- https://checkio.org: Gamified code tutorials for python.
- https://stackoverflow.com: A forum where programmers discuss programming problems. Any problem you are running into has likely been seen (and solved) before.

Programming Best Practices In the opinion of the author, the best way to learn the informal rules of programming is to adapt the tools of other programmers for one's use. This will teach the way code is currently written and how to make code more useful to others. Here are some sources:

- https://opensource.com
- https://alternativeto.net: Find an open-source version of software you use, look at the code yourself.
- https://curc.readthedocs.io
- "Hints for Computer System Design" by Lampson [10]

Two sources that deserve mention for widely covering code architecture are Code Complete [11] for classical techniques and Clean Code [12] for covering the more modern Agile methods for coding.

Emerging Trends

There are many potential developments for Health Information Technology in the near future. All of them carry implications for programmers, clinicians, and ethicists alike. As advancements in this category overlap substantially with other chapters in this text, only the issues most closely tied to programming and computer systems are presented.

Cloud-Based Computing Cloud-based computing means that rather than running software locally on the user's computer (or a computer owned by the user's company) and sending requests to the server in the cloud as needed, part or all of the application is run on the server itself. This allows for more flexibility in software crashes, greater ease in updating the software, potentially better protection of corporate secrets, and makes it much easier to roll out the software to new users. All the user needs is an internet browser. New computer languages (Dart/Flutter) and development platforms (AngularJS) have been constructed to enable this structure. The principal drawback is in the speed of the application since the data must move over the internet to be used and becomes dependent on shared hardware that many users might be trying to use simultaneously. A secondary issue is security, as more organizations are involved in storing and transporting data.

Best-in-Class vs. All-in-One This conflict refers to choosing between a collection of the best-specialized application for each task (such as one app for surgery scheduling and a different one for prescription transmission) or a suite of applications from a single company that covers all needs. As expected, best-in-class applications outperform in their main task but typically struggle to communicate with systems designed by others. In the medical industry, best-in-class applications dominated initially, mostly because few organizations could afford applications for all purposes. Eventually, all-in-ones began to take the lead as communication problems proved to be too harmful. Now, as better standards proliferate (e.g., SMART on FHIR) [13], and withholding connectivity becomes illegal (e.g., information blocking) [14], best-in-class alternatives that communicate seamlessly may become much more common.

Open-Source Software Open-Source refers to the practice of sharing all information freely [15]. In the case of software, it means making all code available for anyone to view, use and change for free (usually as long as credit is given to the author.) Linux, an operating system on which both Android and iOS are based, is open-source. Most servers running web pages use Linux or one of its derivatives. The VistA project is an open-source medical record system once used by the Veterans Health Administration in the United States. It continues to be used by many clinics around the world. GNU Health is another open-source suite of interconnected applications for health management focusing on social medicine. OpenMRS and OpenHIE, developed by the Regenstrief Institute, are open-source medical records systems used in several places worldwide and excel in record-sharing capabilities. And because the code is truly open, anyone with a desire to help can adopt part of the project and become a programmer or subject matter expert shaping the next version of the software.

Distributed Computing The current model used by most electronic health record systems is to have all the patient records stored in a single database (with some backups nearby and far away) that all users contact to search for information. This arrangement is highly susceptible to accidental or malicious failures and requires occasional complete blackouts to update the system. One solution is to move to **distributed computing**, where patient records and other information are scattered throughout numerous computers and copied many times over. This would make an accident or even an attack much less likely to do significant damage to the system but poses its own challenges. One has to do with establishing how to ensure there are enough copies of a given record to be accessed easily, but not so many that all device hard drives are full. Another challenge is making sure the data is safe on all the devices it has been scattered to.

Alternate Computing Methods The way we make computers is rapidly reaching a dead-end due to the laws of physics. Increasing the speed of a computer relies on making its parts smaller. But, if they become much smaller, the vibration of a single atom could have disastrous consequences. Some alternatives have been suggested that would use completely different computational models. The quantum computer accepts a series of coefficients for physics equations, runs them until they reach a steady-state, and then produces an answer in the form of quantum bits [16, 17]. A DNA computer could theoretically store data extremely compactly and perform its functions by splicing DNA strands [18, 19]. These systems require a very different approach to the one presented here, and none has yet proven that it can truly replace our current model.

Summary

Many of the problems we face in medicine can be addressed with technology. That technology will inevitably rely on code. But the programmer is not trained in medicine, anatomy, pathology or even the sciences medicine depends on. To communicate to design solutions, clinicians must speak the language of programmers and understand some of the constraints imposed by computers. By discussing program design in the language of programmers, clinicians have a better chance of implementing the tools that patients and society require. Without that communication, the process for maintaining the health of our communities will inevitably suffer from bugs.

Here is a brief recap of the themes from this chapter:

- Computers are not smart. They only do exactly what they are told.
- Computers are powerful. They can shorten a task of a lifetime to mere seconds.
- Computers must be instructed carefully, or bad things will happen.
- Control structures are the programmer's tools and include sequential blocks, iterative blocks, conditional blocks, and recursion.

Most problems can be broken down into solvable steps using decomposition, abstraction, and pattern recognition. Keep these principles in mind when developing new or troubleshooting existing programs:

- Quality software works efficiently but can also be understood and replaced easily.
- Specifications are required to understand if a program is finished and safe.
- Verbose comments are helpful to those who come after you.
- Checking the code of others is difficult. Working it out by hand is a helpful tool but using a computer to do it is better.

Questions for Discussion

1. Returning to the question at the end of the vignette, consider the following questions:
   (a) What are the parts of such a system?
   (b) What tasks must such a system perform, and which consumes the most resources?
   (c) How might the system tasks conflict with the tasks of other users of the database? (e.g., nurses, doctors, administrators)
   (d) Are any of the tasks illegal?
2. What value does a clinical informaticist provide that even an experienced health information programmer cannot?
3. Must a clinical informaticist also be a programmer? How much programming should an informaticist know?
4. Complete the specification for the water irrigation system or another programming problem of your choice.
5. Write pseudocode to solve the problem mentioned in the vignette. Consider at least two regular preventative maintenance items related to your specialty and one barrier to care.
   (a) What parts of the solution can run in parallel?

(b) What pieces of data are needed to be stored in a database?
(c) Consider the problem as a client-server model. What parts would run on the client? Which would run on the server?

References

1. Lipkin M. Historical background on the origin of computer medicine. Proc Annu Symp Comput Appl Med Care. 1984;987–90.
2. Von Neumann J. First draft report on the EDVAC. Moore School of Electrical Engineering University of Pennsylvania. 30 June 1945.
3. The Unicode Consortium. The Unicode Standard, Version 12.1.0. Mountain View, CA; 2019.
4. Waterman A, Asanovic K. The RISC-V instruction set manual, Volume I: User-level ISA, document version 2.2. RISC-V Foundation, May 2017.
5. International Organization for Standardization. Systems and software engineering—systems and software quality requirements and evaluation (SQuaRE)—system and software quality models (ISO Standard No. 25010:2011). 2011.
6. Denning PJ, Tedre M. Computational thinking. Cambridge, MA: MIT Press; 2019. Accessed 17 Mar 2021.
7. Bentley J. Programming pearls: Bumper-Sticker computer science. Commun ACM. 1985;28(9):896–901.
8. Rafique Y, Mišić VB. IEEE Trans Software Eng. 2013;39(6):835–56. https://doi.org/10.1109/TSE.2012.28.
9. Mars Climate Orbiter Mishap Investigation Board. NASA (1999). Phase I Report.
10. Lampson BW. Hints for computer system design. ACM Oper Syst Rev. 1983;15(5):33–48.
11. McConnell S. Code complete. 2nd ed. Redmond, WA: Microsoft Press; 2004.
12. Martin RC. Clean code: a handbook of agile software craftsmanship. 1st ed. Upper Saddle River, NJ: Prentice Hall PTR; 2008.
13. Mandl KD, Kohane IS. No small change for the health information economy. N Engl J Med. 2009;360(13):1278–81.
14. 21st Century Cares Act Final Rule. 85 FR 25642.
15. What is open source? Internet source: https://opensource.com/resources/what-is-open-source. Accessed 22 Mar 2021.
16. Benioff P. The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines. J Stat Phys. 1980;22:563–91.
17. Kanamori Y, Yoo S. Quantum computing: principles and applications. J Int Technol Inf Manag. 2020;29:2. Article 3
18. Organick L, Ang S, Chen YJ, et al. Random access in large-scale DNA data storage. Nat Biotechnol. 2018;36:242–8.
19. Erlich Y, Zielinski D. DNA Fountain enables a robust and efficient storage architecture. Science. 2017;355(6328):950–4.