At its most basic level, programming a computer simply means telling it what to do, and this vapid-sounding definition is not even a joke. There are no other truly fundamental aspects of computer programming; everything else we talk about will simply be the details of a particular, usually artificial, mechanism for telling a computer what to do. Sometimes these mechanisms are chosen because they have been found to be convenient for programmers (people) to use; other times they have been chosen because they’re easy for the computer to understand. The first hard thing about programming is to learn, become comfortable with, and accept these artificial mechanisms, whether they make “sense” to you or not.

In fact, you shouldn’t worry if some (or even many) of the mechanisms used for programming a computer don’t make sense. It doesn’t make sense that the cold water faucet has to be on the right side and the hot one has to be on the left; that’s just the convention we’ve settled on. Similarly, many computer programming mechanisms are quite arbitrary, and were chosen not because of any theoretical motivation but simply because we needed an unambiguous way to say something to a computer.

In this introduction to programming, we’ll talk about several things: skills needed in programming, a simplified programming model, elements of real programming languages, computer representation of numbers, characters and strings, and compiler terminology.

Skills Needed in Programming

I’m not going to claim that programming is easy, but I am going to say that it is not hard for the reasons people usually assume it is. Programming is not a deeply theoretical subject like Chemistry or Physics; you don’t need an advanced degree to do well at it. (There are important principles of Computer Science, but it’s possible to get a degree after studying them and to have only vague ideas of how to apply them to practical programming. Contrariwise, we’ll experience many important Computer Science lessons by the seat of our pants, without bewildering ourselves with abstract notation; there are plenty of successful programmers who don’t have Computer Science degrees.)

Comparing programming to some physical tasks, programming does not require some innate talent or skill, like gymnastics or painting or singing. You don’t have to be strong or coordinated or graceful or have perfect pitch. Programming does, however, require care and craftsmanship, like carpentry or metalworking. If you’ve ever taken a shop class, you may remember that some students seemed to be able to turn out beautiful projects effortlessly, while other students were all thumbs and made the exact mistakes that the teacher told them not to make. What distinguished the successful students was not that they were better or smarter, but just that they paid more attention to what was going on and were more careful and deliberate about what they were doing. (Perhaps care and attention are innate skills too, like gymnastic ability; I don’t know.)
Some things you do need are (1) attention to detail, (2) stupidity, (3) good memory, and (4) an ability to think abstractly, and on several levels. Let’s look at these qualities in a bit more detail:

1. attention to detail

In programming, the details matter. Computers are incredibly stupid (more on this in a minute). You can’t be vague; you can’t describe your program 3/4 of the way and then say “Ya know what I mean?” and have the compiler figure out the rest. You have to dot your i’s and cross your t’s. If the language says you have to declare variables before using them, you have to. If the language says you have to use parentheses here and square brackets there and squiggly braces some third place, you have to.

2. stupidity

Computers are incredibly stupid. They do exactly what you tell them to do: no more, no less. If you gave a computer a bottle of shampoo and told it to read the directions and wash its hair, you’d better be sure it was a big bottle of shampoo, because the computer is going to wet hair, lather, rinse, repeat, wet hair, lather, rinse, repeat, wet hair, lather, rinse, repeat, ... I saw an ad by a microprocessor manufacturer suggesting the “smart” kinds of appliances we’d have in the future and comparing them to “dumb” appliances like toasters. I believe they had it backwards. A toaster (an old-fashioned one, anyway) has two controls, and one of them is optional: if you don’t set the darkness control, it’ll do the best it can. You don’t have to tell it how many slices of bread you’re toasting, or what kind. (“Modern” toasters have begun to reverse this trend...) Compare this user interface to most microwave ovens: they won’t even let you enter the cooking time until you’ve entered the power level.

When you’re programming, it helps to be able to “think” as stupidly as the computer does, so that you’re in the right frame of mind for specifying everything in minute detail, and not assuming that the right thing will happen unless you tell it to.

(This is not to say that you have to specify everything; the whole point of a high-level programming language like C is to take some of the busiwork burden off the programmer. A C compiler is willing to intuit a few things—for example, if you assign an integer variable to a floating-point variable, it will supply a conversion automatically. But you have to know the rules for what the compiler will assume and what things you must specify explicitly.)

3. good memory

There are a lot of things to remember while programming: the syntax of the language, the set of prewritten functions that are available for you to call and what parameters they take, what variables and functions you’ve defined in your program and how you’re using them, techniques you’ve used or seen in the past which you can apply to new problems, bugs you’ve had in the past which you can either try to avoid or at least recognize by their symptoms. The more of these details you can keep in your head at one time (as opposed to looking them up all the time), the more successful you’ll be at programming.
4. ability to abstract, think on several levels

This is probably the most important skill in programming. Computers are some of the most complex systems we’ve ever built, and if while programming you had to keep in mind every aspect of the functioning of the computer at all levels, it would be a Herculean task to write even a simple program.

One of the most powerful techniques for managing the complexity of a software system (or any complex system) is to compartmentalize it into little “black box” processes which perform useful tasks but which hide some details so you don’t have to think about them all the time.

We compartmentalize tasks all the time, without even thinking about it. If I tell you to go to the store and pick up some milk, I don’t tell you to walk to the door, open the door, go outside, open the car door, get in the car, drive to the store, get out of the car, walk into the store, etc. I especially don’t tell you, and you don’t even think about, lifting each leg as you walk, grasping door handles as you open them, etc. You never (unless perhaps if you’re gravely ill) have to worry about breathing and pumping your blood to enable you to perform all of these tasks and subtasks.

We can carry this little example in the other direction, as well. If I ask you to make some ice cream, you might realize that we’re out of milk and go and get some without my asking you to. If I ask you to help put on a party for our friends, you might decide to make ice cream as part of that larger task. And so on.

Compartmentalization, or abstraction, is a vital skill in programming, or in managing any complex system. Despite what I said in point 3 above, we can only keep a small number of things in our head at one time. A large program might have 100,000 or 1,000,000 or 10,000,000 lines of code. If it were necessary to understand all of the lines together and at once to understand the program, the program would be impossible to write or understand. Only if it is possible to think about small pieces in isolation will it ever be possible to work with a large program.

Compartmentalization, powerful though it is, is not automatic, and not necessarily an instant cure for all of our organizational problems. We carry a lot of assumptions around about how various things work, and things work well only as long as these assumptions hold. To return to the previous example, if I ask you to go the store and get some milk, I’m assuming that you know which kind to get, where the store is, how to get there, how to drive if you need to, etc. If some of these assumptions weren’t valid, or if there were several options for any of them, we might have to modify the way I gave you instructions. I might have to tell you to drive to the store, or to go to Safeway, or to get some two percent milk.

Therefore, we can’t simply compartmentalize all of our processes and subprocesses and forget about complexity problems forever. We have to remember at least some of the assumptions surrounding the compartmentalization scheme. We have to remember what we can and can’t expect from the processes (people, computer programs, etc.) which we call on to do tasks for us. We have to make sure that we keep our end of the bargain and
don’t fall down on any of the commitments and promises we’ve made on the tasks we’ve been asked to do and which others are assuming we’ll keep.

Thinking about the mechanics of a design hierarchy, while also using that hierarchy to avoid having to think about every detail of it at every level all of the time, is one of the things I mean by “thinking on several levels.” It’s tricky to do (obviously, it’s tricky even to describe), but it’s the only way to cut through large, complex problems.

What’s hard about programming (besides maybe having trouble with the four traits above) is mostly picky little detail and organizational problems, and people problems. A large program is a terribly complex system; a large programming project worked on by many people has to work very hard at peripheral, picayune tasks like documentation and communication if the project is to avoid drowning in a flood of little details and bugs.

**Simplified Programming Model**

Imagine an ordinary pocket calculator which can add, subtract, multiply, and divide, and which has a few memory registers which you can store numbers in. At a grossly oversimplified level (so simple that we’ll abandon it in just a minute), we can think of a computer as a calculator which is able to push its own buttons. A computer program is simply the list of instructions that tells the computer which buttons to push. (Actually, there are “keystroke programmable” calculators which you program in just about this way.)

Imagine using such a calculator to perform the following task: Given a list of numbers, compute the average of all the numbers, and also find the largest number in the list.

You can imagine giving the calculator a list of instructions for performing this task, or you can imagine giving a list of instructions to a very stupid but very patient person who is able to follow instructions blindly but accurately, as long as the instructions consist of pushing buttons on the calculator and making simple yes/no decisions. (For our purposes just now, either imaginary model will work.)

Your instructions might look something like this:

“We’re going to use memory register 1 to store the running total of all the numbers, memory register 2 to store how many numbers we’ve seen, and register 3 to store the largest number we’ve seen. For each number in the input list, add it to register 1. Add 1 to register 2. If the number you just read is larger than the number in register 3, store it in register 3. When you’ve read all the numbers, divide register 1 by register 2 to compute the average, and also retrieve the largest number from register 3.”

There are several things to notice about the above list of instructions:

1. The first sentence, which explains what the registers are used for, is more for our benefit than the entity who will be pushing the buttons on the calculator. The entity pushing the buttons doesn’t care what the numbers mean, it just manipulates them as directed. Similarly, the words “to compute the average” and “largest” in the last sentence are
extraneous; they don’t tell the entity pushing the button anything it needs to know (or that it can even understand).

2. The instructions use the word “it” several times. Even in English, where we’re used to a certain amount of ambiguity which we can usually work out from the context, pronouns like “it” can cause problems in sentences, because sometimes it isn’t obvious what they mean. (For example, in the preceding sentence, does “they” refer to “pronouns,” “problems,” or “sentences?”) In programming, you can never get away with ambiguity; you have to be quite precise about which “it” you’re referring to.

3. The instructions are pretty vague about the details of reading the next number in the input list and detecting the end of the list.

4. The “program” contains several bugs! It uses registers 1, 2, and 3, but we never say what to store in them in the first place. Unless they all happen to start out containing zero, the average or maximum value computed by the “program” will be incorrect. (Actually, if all of the numbers in the list are negative, having register 3 start out as 0 won’t work, either.)

Here is a somewhat more detailed version of the “program,” which removes some of the extraneous information and ambiguity, makes the input list handling a bit more precise, and fixes at least some of the bugs. (To make the concept of “the number just read from the list” unambiguous, this “program” stores it in register 4, rather than referring to it by “it.” Also, for now, we’re going to assume that the numbers in the input list are non-negative.)

“Store 0 in registers 1, 2, and 3. Read the next number from the list. If you’re at the end of the list, you’re done. Otherwise, store the number in register 4. Add register 4 to register 1. Add 1 to register 2. If register 4 is greater than register 3, store register 4 in register 3. When you’re done, divide register 1 by register 2 and print the result, and print the contents of register 3.”

When we add the initialization step (storing 0 in the registers), we realize that it’s not quite obvious which steps happen once only and which steps happen once for each number in the input list (that is, each time through the processing loop). Also, we’ve assumed that the calculator can do arithmetic operations directly into memory registers. To make the loop boundaries explicit, and the calculations even simpler (assuming that all the calculator can do is store or recall memory registers from or to the display, and do calculations in the display), the instructions would get more elaborate still:

“Store 0 in register 1. Store 0 in register 2. Store 0 in register 3. Here is the start of the loop: read the next number from the list. If you’re at the end of the list, you’re done. Otherwise, store the number in register 4. Recall from register 1, recall from register 4, add them, store in register 1. Recall from register 2, add 1, store in register 2. Recall from register 3, recall from register 4, if greater store in register 3. Go back to the beginning of the loop. When you’re done: recall from register 1, recall from register 2, divide them, print; recall from register 3, print.”

We could continue to “dumb down” this list of instructions even further, but hopefully you’re getting the point: the instructions we use when programming computers have to be very precise, and at a level of pickiness and detail which we don’t usually use with each other. (Actually, things aren’t quite as bad as these examples might suggest. The “dumbing down” we’ve been doing has been somewhat in the direction of assembly language, which wise programmers don’t use much any more. In a higher-level
language such as C, you don’t have to worry so much about register assignment and individual arithmetic operators.)

Real computers can do quite a bit more than 4-function pocket calculators can; for one thing, they can manipulate strings of text and other kinds of data besides numbers. Let’s leave pocket calculators behind, and start looking at what real computers (at least under the control of programming languages like C) can do.

**Real Programming Model**

A computer program consists of two parts: code and data. The code is the set of instructions for performing a task, and the data is the set of “registers” or “memory locations” which contain the intermediate results which are used as the program performs its calculations.

Note that the code is relatively static while the data is dynamic. Once you’ve gotten a program working, its code won’t change, but every time you run it, it will typically be working with different data, so the memory locations will take on different values.

Once you’ve written a program, you’ve defined a new thing that your computer can do. The applications (text and graphic editors, spreadsheets, games, etc.) which your computer may already have are “just” programs, written by programmers using programming languages such as the one you’re about to learn.

**Elements of Real Programming Languages**

There are several elements which programming languages, and programs written in them, typically contain. These elements are found in all languages, not just C. If you understand these elements and what they’re for, not only will you understand C better, but you’ll also find learning other programming languages, and moving between different programming languages, much easier.

1. There are variables or objects, in which you can store the pieces of data that a program is working on. Variables are the way we talk about memory locations (data), and are analogous to the “registers” in our pocket calculator example. Variables may be global (that is, accessible anywhere in a program) or local (that is, private to certain parts of a program).

2. There are expressions, which compute new values from old ones.

3. There are assignments which store values (of expressions, or other variables) into variables. In many languages, assignment is indicated by an equals sign; thus, we might have

   \[ b = 3 \]

   or

   \[ c = d + e + 1 \]

   The first sets the variable \( b \) to 3; the second sets the variable \( c \) to the sum of the variables \( d \) plus \( e \) plus 1.
The use of an equals sign can be mildly confusing at first. In mathematics, an equals sign indicates equality: two things are stated to be inherently equal, for all time. In programming, there’s a time element, and a notion of cause-and-effect: after the assignment, the thing on the left-hand side of the assignment statement is equal to what the stuff on the right-hand side was before. To remind yourself of this meaning, you might want to read the equals sign in an assignment as “gets” or “receives”: \( a = 3 \) means “a gets 3” or “a receives 3.”

(A few programming languages use a left arrow for assignment

\[ a \leftarrow 3 \]

to make the “receives” relation obvious, but this notation is not too popular, if for no other reason than that few character sets have left arrows in them, and the left arrow key on the keyboard usually moves the cursor rather than typing a left arrow.)

If assignment seems natural and unconfusing so far, consider the line

\[ i = i + 1 \]

What can this mean? In algebra, we’d subtract \( i \) from both sides and end up with

\[ 0 = 1 \]

which doesn’t make much sense. In programming, however, lines like

\[ i = i + 1 \]

are extremely common, and as long as we remember how assignment works, they’re not too hard to understand: the variable \( i \) receives (its new value is), as always, what we get when we evaluate the expression on the right-hand side. The expression says to fetch \( i \)'s (old) value, and add 1 to it, and this new value is what will get stored into \( i \). So \( i = i + 1 \) adds 1 to \( i \); we say that it increments \( i \).

(We’ll eventually see that, in C, assignments are just another kind of expression.)

4. There are conditionals which can be used to determine whether some condition is true, such as whether one number is greater than another. (In some languages, including C, conditionals are actually expressions which compare two values and compute a “true” or “false” value.)

5. Variables and expressions may have types, indicating the nature of the expected values. For instance, you might declare that one variable is expected to hold a number, and that another is expected to hold a piece of text. In many languages (including C), your declarations of the names of the variables you plan to use and what types you expect them to hold must be explicit.

There are all sorts of data types handled by various computer languages. There are single characters, integers, and “real” (floating point) numbers. There are text strings
There are types which reference (point at) values of other types. Finally, there may be user-defined data types, such as structures or records, which allow the programmer to build a more complicated data structure, describing a more complicated object, by accreting together several simpler types (or even other user-defined types).

6. There are statements which contain instructions describing what a program actually does. Statements may compute expressions, perform assignments, or call functions (see below).

7. There are control flow constructs which determine what order statements are performed in. A certain statement might be performed only if a condition is true. A sequence of several statements might be repeated over and over, until some condition is met; this is called a loop.

8. An entire set of statements, declarations, and control flow constructs can be lumped together into a function (also called routine, subroutine, or procedure) which another piece of code can then call as a unit.

When you call a function, you transfer control to it and wait for it to do its job, after which it returns to you; it may also return a value as a result of what it has done. You may also pass values to the function on which it will operate or which otherwise direct its work.

Placing code into functions not only avoids repetition if the same sequence of actions must be performed at several places within a program, but it also makes programs easy to understand, because you can see that some function is being called, and performing some (presumably) well-defined subtask, without always concerning yourself with the details of how that function does its job. (If you've ever done any knitting, you know that knitting instructions are often written with little sub-instructions or patterns which describe a sequence of stitches which is to be performed multiple times during the course of the main piece. These sub-instructions are very much like function calls in programming.)

9. A set of functions, global variables, and other elements makes up a program. An additional wrinkle is that the source code for a program may be distributed among one or more source files. (In the other direction, it is also common for a suite of related programs to work closely together to perform some larger task, but we'll not worry about that "large scale integration" for now.)

10. In the process of specifying a program in a form suitable for a compiler, there are usually a few logistical details to keep track of. These details may involve the specification of compiler parameters or interdependencies between different functions and other parts of the program. Specifying these details often involves miscellaneous syntax which doesn't fall into any of the other categories listed here, and which we might lump together as "boilerplate."
Many of these elements exist in a hierarchy. A program typically consists of functions and global variables; a function is made up of statements; statements usually contain expressions; expressions operate on objects. (It is also possible to extend the hierarchy in the other direction; for instance, sometimes several interrelated but distinct programs are assembled into a suite, and used in concert to perform complex tasks. The various “office” packages--integrated word processor, spreadsheet, etc.--are an example.)

As we mentioned, many of the concepts in programming are somewhat arbitrary. This is particularly so for the terms expression, statement, and function. All of these could be defined as “an element of a program that actually does something.” The differences are mainly in the level at which the “something” is done, and it’s not necessary at this point to define those “levels.” We’ll come to understand them as we begin to write programs.

An analogy may help: Just as a book is composed of chapters which are composed of sections which are composed of paragraphs which are composed of sentences which are composed of words (which are composed of letters), so is a program composed of functions which are composed of statements which are composed of expressions (which are in fact composed of smaller elements which we won’t bother to define). Analogies are never perfect, though, and this one is weaker than most; it still doesn’t tell us anything about what expressions, statements, and functions really are. If “expression” and “statement” and “function” seem like totally arbitrary words to you, use the analogy to understand that what they are is arbitrary words describing arbitrary levels in the hierarchical composition of a program, just as “sentence,” “paragraph,” and “chapter” are different levels of structure within a book.

The preceding discussion has been in very general terms, describing features common to most “conventional” computer languages. If you understand these elements at a relatively abstract level, then learning a new computer language becomes a relatively simple matter of finding out how that language implements each of the elements. (Of course, you can’t understand these abstract elements in isolation; it helps to have concrete examples to map them to. If you’ve never programmed before, most of this section has probably seemed like words without meaning. Don’t spend too much time trying to glean all the meaning, but do come back and reread this handout after you’ve started to learn the details of a particular programming language such as C.)

Finally, there’s no need to overdo the abstraction. For the simple programs we’ll be writing, in a language like C, the series of calculations and other operations that actually takes place as our program runs is a simpleminded translation (into terms the computer can understand) of the expressions, statements, functions, and other elements of the program. Expressions are evaluated and their results assigned to variables. Statements are executed one after the other, except when the control flow is modified by if/then conditionals and loops. Functions are called to perform subtasks, and return values to their callers, which have been waiting for them.

**Computer Representation of Numbers**

Most computers represent integers as binary numbers (see the “math refresher” handout) with a certain number of bits. A computer with 16-bit integers can represent integers from 0 to 65,535 (that is, from 0 to $2^{16}$-1), or if it chooses to make half of them negative, from -32,767 to 32,767. (We won’t get into the details of how computers handle negative numbers right now.) A 32-bit integer can represent values from 0 to 4,294,967,295, or ±2,147,483,647.
Most of today’s computers represent real (i.e. fractional) numbers using exponential notation. (Again, see “math refresher” handout. Actually, deep down inside, computers usually use powers of 2 instead of powers of 10, but the difference isn’t important to us right now.) The advantage of using exponential notation for real numbers is that it lets you trade off the range and precision of values in a useful way. Since there’s an infinitely large number of real numbers (and in three directions: very large, very small, and very negative), it will never be possible to represent all of them (without using potentially infinite amounts of space).

Suppose you decide to give yourself six decimal digits’ worth of storage (that is, you decide to devote an amount of memory capable of holding six digits) for each value. If you put three digits to the left and three to the right of the decimal point, you could represent numbers from 999.999 to -999.999, and as small as 0.001. (Furthermore, you’d have a resolution of 0.001 everywhere: you could represent 0.001 and 0.002, as well as 999.998 and 999.999.) This would be a workable scheme, although 0.001 isn’t a very small number and 999.999 isn’t a very big one.

If, on the other hand, you used exponential notation, with four digits for the base number and two digits for the exponent, you could represent numbers from $9.999 \times 10^9$ to $-9.999 \times 10^9$, and as small as $1 \times 10^{-99}$ (or, if you cheat, $0.001 \times 10^{-99}$). You can now represent both much larger numbers and much smaller; the tradeoff is that the absolute resolution is no longer constant, and gets smaller as the absolute value of the numbers gets larger. The number 123,456 can only be represented as 123.4, and the number 123,456 can only be represented as 123,400. You can’t represent 999.999 anymore; you have to settle for 999.9 ($9.999 \times 10^3$) or 1000 ($1.000 \times 10^3$). You can’t distinguish between 999.998 and 999.999 anymore.

Since superscripts are difficult to type, computer programming languages usually use a slightly different notation. For example, the number $1.234 \times 10^5$ might be indicated by $1.234e5$, where the letter e replaces the “times ten to the” part.

You will often hear real, exponential numbers referred to on computers as “floating point numbers” or simply “floats,” and you will also hear the term “double” which is short for “double-precision floating point number.” Some computers also use “fixed point” real numbers, (which work along the lines of our “three to the left, three to the right” example of a few paragraphs back), but those are comparatively rare and we won’t need to discuss them.

It’s important to remember that the precision of floating-point numbers is usually limited, and this can lead to surprising results. The result of a division like $1/3$ cannot be represented exactly (it’s an infinitely repeating fraction, $0.333333...$), so the computation $(1 ÷ 3) \times 3$ tends to yield a result like $0.999999...$ instead of 1.0. Furthermore, in base 2, the fraction $1/10$, or 0.1 in decimal, is also an infinitely repeating fraction, and cannot be represented exactly, either, so $(1 ÷ 10) \times 10$ may also yield $0.999999...$. For these reasons and others, floating-point calculations are rarely exact. When working with computer floating point, you have to be careful not to compare two numbers for exact equality, and you have to ensure that “round off error” doesn’t accumulate until it seriously degrades the results of your calculations.

Characters, Strings, and Numbers

The earliest computers were number crunchers only, but almost all more recent computers have the ability to manipulate alphanumeric data as well. The computer, and our programming languages, tend
to maintain a strict distinction between numbers on the one hand and alphanumeric data on the other, so we have to maintain that distinction in our own minds as well.

One fundamental component of a computer’s handling of alphanumeric data is its character set. A character set is, not surprisingly, the set of all the characters that the computer can process and display. (Each character generally has a key on the keyboard to enter it and a bitmap on the screen which displays it.) A character set consists of letters, numbers, punctuation, etc., but the point of this discussion is not so much what the characters are but that we have to be careful to distinguish between characters, strings, and numbers.

A character is, well, a single character. If we have a variable which contains a character value, it might contain the letter ‘A’, or the digit ‘2’, or the symbol ‘&’.

A string is a set of zero or more characters. For example, the string “and” consists of the characters ‘a’, ‘n’, and ‘d’. The string “K2!” consists of the characters ‘K’, ‘2’, and ‘!’. The string “.” consists of the single character ‘.’, and the empty string “” consists of no characters at all. Not to belabor the point, but the string “123” consists of the characters ‘1’, ‘2’, and ‘3’, and the string “4” consists of the single character ‘4’.

The last two examples illustrate some important and perhaps surprising or annoying distinctions. The character ‘4’ and the string “4” are conceptually different, and neither of them is quite the same as the number 4. The string “123” consists of three characters, and it looks like the number 123 to us, but as far as the computer is concerned it is just a string. The number 123 is, when used for ordinary numeric purposes, not represented internally as a string of three characters (instead, it is typically represented as a 16- or 32-bit integer). When we have a string which contains a numeric value which we wish to manipulate as a number, we must typically ask for the string to be explicitly converted to that number somehow. Similarly, we may have reason to convert a number to a string of digits making up its decimal representation.

We may also find ourselves needing to convert back and forth between characters and the numeric codes which are assigned to each character in a character set. (For example, in the ASCII character set, the character ‘A’ is code 65, the character ‘.’ is code 46, and the character ‘4’ is, perhaps surprisingly, code 52.)

Compiler Terminology

C is a compiled language. This means that the programs you write are translated, by a program called a compiler, into executable machine-language programs which you can actually run. Executable machine-language programs are self-contained and run very quickly. Since they are self-contained, you don’t need copies of the source code (the original programming-language text you composed) or the compiler in order to run them; you can distribute copies of just the executable and that’s all someone else needs to run it. Since they run relatively quickly, they are appropriate for programs which will be written once and run many times.

A compiler is a special kind of program: it is a program that builds other programs. What happens is that you invoke the compiler (as a program), and it reads the programming language statements that you have written and turns them into a new, executable program. When the compiler has finished its work, you then invoke your program (the one the compiler just built) to see if it works.
The main alternative to a compiled computer language or program is an interpreted one, such as BASIC. An interpreted language is interpreted (by, not surprisingly, a program called an interpreter) and its actions performed immediately. If you gave a copy of an interpreted program to someone else, they would also need a copy of the interpreter to run it. No standalone executable machine-language binary program is produced.

In other words, for each statement that you write, a compiler translates into a sequence of machine language instructions which does the same thing, while an interpreter simply does it (where “it” is whatever the statement that you wrote is supposed to do).

The big advantage of an interpreted language is that your program runs right away; you don’t have to perform--and wait for--the separate tasks of compiling and then running your program. (Actually, on a modern computer, neither compiling nor interpreting takes much time, so some of these distinctions become less important.)

Actually, whether a language is compiled or interpreted is not always an inherent part of the language. There are interpreters for C, and there are compilers for BASIC. However, most languages were designed with one or the other mechanism in mind, and there are usually a few difficulties when trying to compile a language which is traditionally interpreted, or vice versa.

The distinction between compilation and interpretation, while it is very significant and can make a big difference, is not one to get worked up over. Most of the time, once you get used to the details of how you get your programs to run, you don’t need to worry about the distinction too much. But it is a useful distinction to have a basic understanding of, and to keep in the back of your mind, because it will help you understand why certain aspects of computer programming (and particular languages) work the way they do.

When you’re working with a compiled language, there are several mechanical details which you’ll want to be aware of. You create one or more source files which are simple text files containing your program, written in whatever language you’re using. You typically use a text editor to work with source files (typically you don’t want to use a full-fledged word processor, since the compiler won’t understand its formatting codes). You supply each source file (you may have one, or more than one) to the compiler, which creates an object file containing machine-language instructions corresponding to your program. Your program is not ready to run yet, however: if you called any functions which you didn’t write (such as the standard library functions provided as part of a programming language environment), you must arrange for them to be inserted into your program, too. The task of combining object files together, while also locating and inserting any library functions, is the job of the linker. The linker puts together the object files you give it, noticing if you call any functions which you haven’t supplied and which must therefore be library functions. It then searches one or more library files (a library file is simply a collection of object files) looking for definitions of the still-unresolved functions, and pulls in any that it finds. When it’s done, it either builds the final, executable file, or, if there were any errors (such as a function called but not defined anywhere) complains.

If you’re using some kind of an integrated programming environment, many of these steps may be taken care of for you so automatically and seamlessly that you’re hardly aware of them.