1 Computers, People, and Programming

"Specialization is for insects."

—R. A. Heinlein

In this chapter, we present some of the things that we think make programming important, interesting, and fun. We also present a few fundamental ideas and ideals. We hope to debunk a couple of popular myths about programming and programmers. This is a chapter to skim for now and to return to later when you are struggling with some programming problem and wondering if it's all worth it.

1.1 Introduction

Like most learning, learning how to program is a chicken and egg problem: We want to get started, but we also want to know why what we are about to learn matters. We want to learn a practical skill, but also make sure it is not just a passing fad. We want to know that we are not going to waste our time, but don't want to be bored by still more hype and moralizing. For now, just read as much of this chapter as seems interesting and come back later when you feel the need to refresh your memory of why the technical details matter outside the classroom.

This chapter is a personal statement of what we find interesting and important about programming. It explains what motivates us to keep going in this field after decades. This is a chapter to read to get an idea of possible ultimate goals and an idea of what kind of person a programmer might be. A beginner's technical book inevitably contains much pretty basic stuff. In this chapter, we lift our eyes from the technical details and consider the big picture: Why is programming a worthwhile activity? What is the role of programming in our civilization? Where can a programmer make contributions to be proud of? Where does programming fit into the greater world of software development, deployment, and maintenance? When people talk about "computer science," "software engineering," "information technology," etc., where does programming fit into the picture? What does a programmer do? What skills does a good programmer have?

To a student, the most urgent reason for understanding an idea, a technique, or a chapter may be to pass a test with a good grade – but there has to be more to learning than that! To someone working in the software industry, the most urgent

reason for understanding an idea, a technique, or a chapter may be to find something that can help with the current project and that will not annoy the boss who controls the next paycheck, promotions, and firings – but there has to be more to learning than that! We work best when we feel that our work in some small way makes the world a better place for people to live in. For tasks that we perform over a period of years (the "things" that professions and careers are made of), ideals and more abstract ideas are crucial.

Our civilization runs on software. Improving software and finding new uses for software are two of the ways an individual can help improve the lives of many. Programming plays an essential role in that.

1.2 Software

Good software is invisible. You can't see it, feel it, weigh it, or knock on it. *Software* is a collection of programs running on some computer. Sometimes, we can see the computer. Often, we can see only something that contains the computer, such as a telephone, a camera, a bread maker, a car, or a wind turbine. We can see what that software does. We can be annoyed or hurt if it doesn't do what it is supposed to do. We can be annoyed or hurt if what it is supposed to do doesn't suit our needs.

How many computers are there in the world? We don't know; billions at least. There may be more computers in the world than people. We need to count servers, desktop computers, laptops, tablets, smartphones, and computers embedded in "gadgets."

How many computers do you (more or less directly) use every day? There are more than 30 computers in my car, two in my cell phone, one in my MP3 player, and one in my camera. Then there is my laptop (on which the page you are reading is being written) and my desktop machine. The air-conditioning controller that keeps the summer heat and humidity at bay is a simple computer. There is one controlling the computer science department's elevator. If you use a modern television, there will be at least one computer in there somewhere. A bit of web surfing gets you into direct contact with dozens – possibly hundreds – of servers through a telecommunications system consisting of many thousands of computers – telephone switches, routers, and so on.

No, I do not drive around with 30 laptops on the backseat of my car! The point is that most computers do not look like the popular image of a computer (with a screen, a keyboard, a mouse, etc.); they are small "parts" embedded in the equipment we use. So, that car has nothing that looks like a computer, not even a screen to display maps and driving directions (though such gadgets are popular in other cars). However, its engine contains quite a few computers, doing things like fuel injection control and temperature monitoring. The power-assisted steering involves at least one computer, the radio and the security

system contain some, and we suspect that even the open/close controls of the windows are computer controlled. Newer models even have computers that continuously monitor tire pressure.

How many computers do you depend on for what you do during a day? You eat; if you live in a modern city, getting the food to you is a major effort requiring minor miracles of planning, transport, and storage. The management of the distribution networks is of course computerized, as are the communication systems that stitch them all together. Modern farming is highly computerized; next to the cow barn you find computers used to monitor the herd (ages, health, milk production, etc.), farm equipment is increasingly computerized, and the number of forms required by the various branches of government can make any honest farmer cry. If something goes wrong, you can read all about it in your newspaper; of course, the articles in that paper were written on computers, set on the page by computers, and (if you still read the "dead tree edition") printed by computerized equipment – often after having been electronically transmitted to the printing plant. Books are produced in the same way. If you have to commute, the traffic flows are monitored by computers in a (usually vain) attempt to avoid traffic jams. You prefer to take the train? That train will also be computerized; some even operate without a driver, and the train's subsystems, such as announcements, braking, and ticketing, involve lots of computers. Today's entertainment industry (music, movies, television, stage shows) is among the largest users of computers. Even non-cartoon movies use (computer) animation heavily; music and photography are also digital (i.e., using computers) for both recording and delivery. Should you become ill, the tests your doctor orders will involve computers, the medical records are often computerized, and most of the medical equipment you'll encounter if you are sent to a hospital to be cured contains computers. Unless you happen to be staying in a cottage in the woods without access to any electrically powered gadgets (including light bulbs), you use energy. Oil is found, extracted, processed, and distributed through a system using computers every step along the way, from the drill bit deep in the ground to your local gas (petrol) pump. If you pay for that gas with a credit card, you again exercise a whole host of computers. It is the same story for coal, gas, solar, and wind power.

The examples so far are all "operational"; they are directly involved in what you are doing. Once removed from that is the important and interesting area of design. The clothes you wear, the telephone you talk into, and the coffee machine that dispenses your favorite brew were designed and manufactured using computers. The superior quality of modern photographic lenses and the exquisite shapes in the design of modern everyday gadgets and utensils owe almost everything to computer-based design and production methods. The craftsmen/designers/artists/engineers who design our environment have been freed from many physical con-

straints previously considered fundamental. If you get ill, the medicines given to cure you will have been designed using computers.

Finally, research – science itself – relies heavily on computers. The telescopes that probe the secrets of distant stars could not be designed, built, or operated without computers, and the masses of data they produce couldn't be analyzed and understood without computers. An individual biology field researcher may not be heavily computerized (unless, of course, a camera, a digital tape recorder, a telephone, etc. are used), but back in the lab, the data has to be stored, analyzed, checked against computer models, and communicated to fellow scientists. Modern chemistry and biology – including medical research – use computers to an extent undreamed of a few years ago and still unimagined by most people. The human genome was sequenced by computers. Or – let's be precise – the human genome was sequenced by humans using computers. In all of these examples, we see computers as something that enables us to do something we would have had a harder time doing without computers.

Every one of those computers runs software. Without software, they would just be expensive lumps of silicon, metal, and plastic: doorstops, boat anchors, and space heaters. Every line of that software was written by some individual. Every one of those lines that was actually executed was minimally reasonable, if not correct. It's amazing that it all works! We are talking about billions of lines of code (program text) in hundreds of programming languages. Getting all that to work took a staggering amount of effort and involved an unimaginable number of skills. We want further improvements to essentially every service and gadget we depend on. Just think of any one service and gadget you rely on; what would you like to see improved? If nothing else, we want our services and gadgets smaller (or bigger), faster, more reliable, with more features, easier to use, with higher capacity, better looking, and cheaper. The likelihood is that the improvement you thought of requires some programming.

1.3 People

Computers are built by people for the use of people. A computer is a very generic tool; it can be used for an unimaginable range of tasks. It takes a program to make it useful to someone. In other words, a computer is just a piece of hardware until someone – some programmer – writes code for it to do something useful. We often forget about the software. Even more often, we forget about the programmer.

Hollywood and similar "popular culture" sources of disinformation have assigned largely negative images to programmers. For example, we have all seen the solitary, fat, ugly nerd with no social skills who is obsessed with video games and breaking into other people's computers. He (almost always a male) is as likely to

want to destroy the world as he is to want to save it. Obviously, milder versions of such caricatures exist in real life, but in our experience they are no more frequent among software developers than they are among lawyers, police officers, car salesmen, journalists, artists, or politicians.

Think about the applications of computers you know from your own life. Were they done by a loner in a dark room? Of course not; the creation of a successful piece of software, computerized gadget, or system involves dozens, hundreds, or thousands of people performing a bewildering set of roles: for example, programmers, (program) designers, testers, animators, focus group managers, experimental psychologists, user interface designers, analysts, system administrators, customer relations people, sound engineers, project managers, quality engineers, statisticians, hardware interface engineers, requirements engineers, safety officers, mathematicians, sales support personnel, troubleshooters, network designers, methodologists, software tools managers, software librarians, etc. The range of roles is huge and made even more bewildering by the titles varying from organization to organization: one organization's "engineer" may be another organization's "programmer" and yet another organization's "developer," "member of technical staff," or "architect." There are even organizations that let their employees pick their own titles. Not all of these roles directly involve programming. However, we have personally seen examples of people performing each of the roles mentioned while reading or writing code as an essential part of their job. Additionally, a programmer (performing any of these roles, and more) may over a short period of time interact with a wide range of people from application areas, such as biologists, engine designers, lawyers, car salesmen, medical researchers, historians, geologists, astronauts, airplane engineers, lumberyard managers, rocket scientists, bowling alley builders, journalists, and animators (yes, this is a list drawn from personal experience). Someone may also be a programmer at times and fill non-programming roles at other stages of a professional career.

The myth of a programmer being isolated is just that: a myth. People who like to work on their own choose areas of work where that is most feasible and usually complain bitterly about the number of "interruptions" and meetings. People who prefer to interact with other people have an easier time because modern software development is a team activity. The implication is that social and communication skills are essential and valued far more than the stereotypes indicate. On a short list of highly desirable skills for a programmer (however you realistically define *programmer*), you find the ability to communicate well – with people from a wide variety of backgrounds – informally, in meetings, in writing, and in formal presentations. We are convinced that until you have completed a team project or two, you have no idea of what programming is and whether you really like it. Among the things we like about programming are all the nice and

interesting people we meet and the variety of places we get to visit as part of our professional lives.

One implication of all this is that people with a wide variety of skills, interests, and work habits are essential for producing good software. Our quality of life depends on those people – sometimes even our life itself. No one person could fill all the roles we mention here; no sensible person would want every role. The point is that you have a wider choice than you could possibly imagine; not that you have to make any particular choice. As an individual you will "drift" toward areas of work that match your skills, talents, and interests.

We talk about "programmers" and "programming," but obviously programming is only part of the overall picture. The people who design a ship or a cell phone don't think of themselves as programmers. Programming is an important part of software development, but not all there is to software development. Similarly, for most products, software development is an important part of product development, but not all there is to product development.

We do not assume that you – our reader – want to become a professional programmer and spend the rest of your working life writing code. Even the best programmers – especially the *best* programmers – spend most of their time *not* writing code. Understanding problems takes serious time and often requires significant intellectual effort. That intellectual challenge is what many programmers refer to when they say that programming is interesting. Many of the best programmers also have degrees in subjects not usually considered part of computer science. For example, if you work on software for genomic research, you will be much more effective if you understand some molecular biology. If you work on programs for analyzing medieval literature, you could be much better off reading a bit of that literature and maybe even knowing one or more of the relevant languages. In particular, a person with an "all I care about is computers and programming" attitude will be incapable of interacting with his or her non-programmer colleagues. Such a person will not only miss out on the best parts of human interactions (i.e., life) but also be a bad software developer.

So, what do we assume? Programming is an intellectually challenging set of skills that are part of many important and interesting technical disciplines. In addition, programming is an essential part of our world, so not knowing the basics of programming is like not knowing the basics of physics, history, biology, or literature. Someone totally ignorant of programming is reduced to believing in magic and is dangerous in many technical roles. If you read Dilbert, think of the pointy-haired boss as the kind of manager you don't want to meet or (far worse) become. In addition, programming can be fun.

But what do we assume you might use programming for? Maybe you will use programming as a key tool in your further studies and work without becoming a professional programmer. Maybe you will interact with other people

professionally and personally in ways where a basic knowledge of programming will be an advantage, maybe as a designer, writer, manager, or scientist. Maybe you will do programming at a professional level as part of your studies or work. Even if you do become a professional programmer it is unlikely that you will do nothing but programming.

You might become an engineer focusing on computers or a computer scientist, but even then you will not "program all the time." Programming is a way of presenting ideas in code – a way of aiding problem solving. It is nothing – absolutely a waste of time – unless you have ideas that are worth presenting and problems worth solving.

This is a book about programming and we have promised to help you learn how to program, so why do we emphasize non-programming subjects and the limited role of programming? A good programmer understands the role of code and programming technique in a project. A good programmer is (at most times) a good team player and tries hard to understand how the code and its production best support the overall project. For example, imagine that I worked on a new MP3 player (maybe to be part of a smartphone or a tablet) and all that I cared about was the beauty of my code and the number of neat features I could provide. I would probably insist on the largest, most powerful computer to run my code. I might disdain the theory of sound encoding because it is "not programming." I would stay in my lab, rather than go out to meet potential users, who undoubtedly would have bad tastes in music anyway and would not appreciate the latest advances in GUI (graphical user interface) programming. The likely result would be disaster for the project. A bigger computer would mean a costlier MP3 player and most likely a shorter battery life. Encoding is an essential part of handling music digitally, so failing to pay attention to advances in encoding techniques could lead to increased memory requirements for each song (encodings differ by as much as 100% for the same-quality output). A disregard for users' preferences – however odd and archaic they may seem to you – typically leads to the users choosing some other product. An essential part of writing a good program is to understand the needs of the users and the constraints that those needs place on the implementation (i.e., the code). To complete this caricature of a bad programmer, we just have to add a tendency to deliver late because of an obsession with details and an excessive confidence in the correctness of lightly tested code. We encourage you to become a good programmer, with a broad view of what it takes to produce good software. That's where both the value to society and the keys to personal satisfaction lie.

1.4 Computer science

Even by the broadest definition, programming is best seen as a part of something greater. We can see it as a subdiscipline of computer science, computer engineering, software engineering, information technology, or any other software-related disci-

pline. We see programming as an enabling technology for those computer and information fields of science and engineering, as well as for physics, biology, medicine, history, literature, and any other academic or research field.

Consider computer science. A 1995 U.S. government "blue book" defines it like this: "The systematic study of computing systems and computation. The body of knowledge resulting from this discipline contains theories for understanding computing systems and methods; design methodology, algorithms, and tools; methods for the testing of concepts; methods of analysis and verification; and knowledge representation and implementation." As we would expect, the Wikipedia entry is less formal: "Computer science, or computing science, is the study of the theoretical foundations of information and computation and their implementation and application in computer systems. Computer science has many sub-fields; some emphasize the computation of specific results (such as computer graphics), while others (such as computational complexity theory) relate to properties of computational problems. Still others focus on the challenges in implementing computations. For example, programming language theory studies approaches to describing computations, while computer programming applies specific programming languages to solve specific computational problems."

Programming is a tool; it is a fundamental tool for expressing solutions to fundamental and practical problems so that they can be tested, improved through experiment, and used. Programming is where ideas and theories meet reality. This is where computer science can become an experimental discipline, rather than pure theory, and impact the world. In this context, as in many others, it is essential that programming is an expression of well-tried practices as well as the theories. It must not degenerate into mere hacking: just get some code written, any old way that meets an immediate need.

1.5 Computers are everywhere

Nobody knows everything there is to know about computers or software. This section just gives you a few examples. Maybe you'll see something you like. At least you might be convinced that the scope of computer use – and through that, programming – is far larger than any individual can fully grasp.

Most people think of a computer as a small gray box attached to a screen and a keyboard. Such computers tend to be good at games, messaging and email, and playing music. Other computers, called laptops, are used on planes by bored businessmen to look at spreadsheets, play games, and watch videos. This caricature is just the tip of the iceberg. Most computers work out of our sight and are part of the systems that keep our civilization going. Some fill rooms; others are smaller than a small coin. Many of the most interesting computers don't directly interact with a human through a keyboard, mouse, or similar gadget.

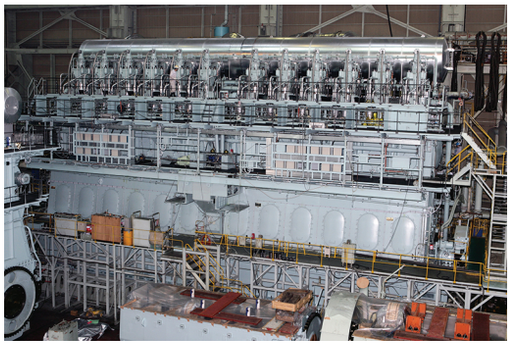
1.5.1 Screens and no screens

The idea of a computer as a fairly large rectangular box with a screen and a keyboard is common and often hard to shake off. However, consider these two computers:

Both of these "gadgets" (which happen to be watches) are primarily computers. In fact, we conjecture that they are essentially the same model computer with different I/O (input/output) systems. The left one drives a small screen (similar to the screens on conventional computers, but smaller) and the second drives little electric motors controlling traditional clock hands and a disk of numbers for day-of-month readout. Their input systems are the four buttons (more easily seen on the right-hand watch) and a radio receiver, used for synchronization with very high-precision "atomic" clocks. Most of the programs controlling these two computers are shared between them.

1.5.2 Shipping

These two photos show a large marine diesel engine and the kind of huge ship that it may power:



Consider where computers and software play key roles here:

- *Design:* Of course, the ship and the engine were both designed using computers. The list of uses is almost endless and includes architectural and engineering drawings, general calculations, visualization of spaces and parts, and simulations of the performance of parts.
- *Construction:* A modern shipyard is heavily computerized. The assembly of a ship is carefully planned using computers, and the work is guided by computers. Welding is done by robots. In particular, a modern double-hulled tanker couldn't be built without little welding robots to do the welding from within the space between the hulls. There just isn't room for a human in there. Cutting steel plates for a ship was one of the world's first CAD/CAM (computer-aided design and computer-aided manufacture) applications.
- *The engine:* The engine has electronic fuel injection and is controlled by a few dozen computers. For a 100,000-horsepower engine (like the one in the photo), that's a nontrivial task. For example, the engine management computers continuously adjust fuel mix to minimize the pollution that would result from a badly tuned engine. Many of the pumps associated with the engine (and other parts of the ship) are themselves computerized.
- *Management:* Ships sail where there is cargo to pick up and to deliver. The scheduling of fleets of ships is a continuing process (computerized, of course) so that routings change with the weather, with supply and demand, and with space and loading capacity of harbors. There are even websites where you can watch the position of major merchant vessels at any time. The ship in the photo happens to be a container vessel (one of the largest such in the world; 397m long and 56m wide), but other kinds of large modern ships are managed in similar ways.
- *Monitoring:* An oceangoing ship is largely autonomous; that is, its crew can handle most contingencies likely to arise before the next port. However, they are also part of a globe-spanning network. The crew has access to reasonably accurate weather information (from and through – computerized – satellites). They have a GPS (global positioning system) and computer-controlled and computer-enhanced radar. If the crew needs a rest, most systems (including the engine, radar, etc.) can be monitored (via satellite) from a shipping-line control room. If anything unusual is spotted, or if the connection "back home" is broken, the crew is notified.

Consider the implication of a failure of one of the hundreds of computers explicitly mentioned or implied in this brief description. Chapter 25 ("Embedded Systems Programming") examines this in slightly more detail. Writing code for a modern ship is a skilled and interesting activity. It is also useful. The cost of

sea transport is really amazingly low. You appreciate that when you buy something that wasn't manufactured locally. Sea transport has always been cheaper than land transport; these days one of the reasons is serious use of computers and information.

1.5.3 Telecommunications

These two photos show a telephone switch and a telephone (that also happens to be a camera, an MP3 player, an FM radio, a web browser, and much more):

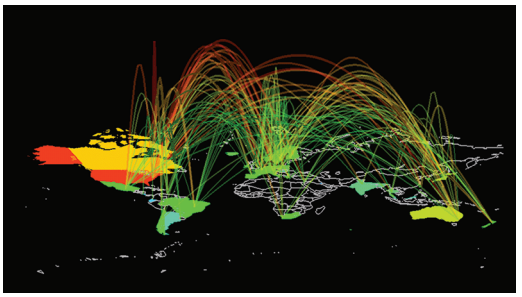
Consider where computers and software play key roles here. You pick up a telephone and "dial," the person you dialed answers, and you talk. Or maybe you get to leave a voicemail, or maybe you send a photo from your phone camera, or maybe you send a text message (hit Send and let the phone do the dialing). Obviously the phone is a computer. This is especially obvious if the phone (like most mobile phones) has a screen and allows more than traditional "plain old telephone services," such as web browsing. Actually, such phones tend to contain several computers: one to manage the screen, one to talk to the phone system, and maybe more.

The part of the phone that manages the screen, does web browsing, etc. is probably the most familiar to computer users: it just runs a graphical user interface to "all the usual stuff." What is unknown to and largely unsuspected by most users is the huge system that the little phone talks to while doing its job. I dial a number in Texas, but you are on vacation in New York City, yet within seconds your phone rings and I hear your "Hello!" over the roar of city traffic. Many phones can perform that trick for essentially any two locations on earth and we just take it for granted. How did my phone find yours? How is the sound transmitted? How is the sound encoded into data packets? The answer could fill

many books much thicker than this one, but it involves a combination of hardware and software on hundreds of computers scattered over the geographical area in question. If you are unlucky, a few telecommunications satellites (themselves computerized systems) are also involved – "unlucky" because we cannot perfectly compensate for the 20,000-mile detour out into space; the speed of light (and therefore the speed of your voice) is finite (light fiber cables are much better: shorter, faster, and carrying much more data). Most of this works remarkably well; the backbone telecommunications systems are 99.9999% reliable (for example, 20 minutes of downtime in 20 years – that's 20/(20*365*24*60). The trouble we have tends to be in the communications between our mobile phone and the nearest main telephone switch.

There is software for connecting the phones, for chopping our spoken words into data packets to be sent over wires and radio links, for routing those messages, for recovering from all kinds of failures, for continuously monitoring the quality and reliability of the services, and of course for billing. Even keeping track of all the physical pieces of the system requires serious amounts of clever software: What talks to what? What parts go into a new system? When do you need to do some preventive maintenance?

Arguably the backbone telecommunications system of the world, consisting of semi-independent but interconnected systems, is the largest and most complicated man-made artifact. To make things a bit more real: remember, this is not just boring old telephony with a few new bells and whistles. The various infrastructures have merged. They are also what the internet (the web) runs on, what our banking and trading systems run on, and what carry our television programs to the broadcasting stations. So, we can add another couple of photos to illustrate telecommunications:



The room is the "trading floor" of the American stock exchange on New York's Wall Street and the map is a representation of parts of the internet backbones (a complete map would be too messy to be useful).

As it happens, we also like digital photography and the use of computers to draw specialized maps to visualize knowledge.

1.5.4 Medicine

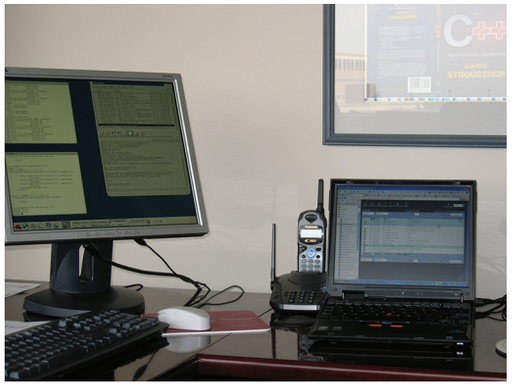
These two photos show a CAT (computed axial tomography) scanner and an operating theater for computer-aided surgery (also called "robot-assisted surgery" or "robotic surgery"):

Consider where computers and software play key roles here. The scanners basically are computers; the pulses they send out are controlled by a computer, and the readings are nothing but gibberish until quite sophisticated algorithms are applied to convert them to something we recognize as a (three-dimensional) image of the relevant part of a human body. To do computerized surgery, we must go several steps further. A wide variety of imaging techniques are used to let the surgeon see the inside of the patient, to see the point of surgery with significant enlargement or in better light than would otherwise be possible. With the aid of a computer a surgeon can use tools that are too fine for a human hand to hold or in a place where a human hand could not reach without unnecessary cutting. The use of minimally invasive surgery (laparoscopic surgery) is a simple example of this that has minimized the pain and recovery time for millions of people. The computer can also help steady the surgeon's "hand" to allow for more delicate work than would otherwise be possible. Finally, a "robotic" system can be operated remotely, thus making it possible for a doctor to help someone remotely (over the internet). The computers and programming involved are mind-boggling, complex, and interesting. The user interface, equipment control, and imaging challenges alone will keep thousands of researchers, engineers, and programmers busy for decades.

We heard of a discussion among a large group of medical doctors about which new tool had provided the most help to them in their work: The CAT scanner? The MRI scanner? The automated blood analysis machines? The high-resolution ultrasound machines? PDAs? After some discussion, a surprising "winner" of this "competition" emerged: instant access to patient records. Knowing the medical history of a patient (earlier illnesses, medicines tried earlier, allergies, hereditary problems, general health, current medication, etc.) simplifies the problem of diagnosis and minimizes the chance of mistakes.

1.5.5 Information

These two photos show an ordinary PC (well, two) and part of a server farm:



We have focused on "gadgets" for the usual reason: you cannot see, feel, or hear software. We cannot present you with a photograph of a neat program, so we show you a "gadget" that runs one. However, much software deals directly with "information." So let's consider "ordinary uses" of "ordinary computers" running "ordinary software."

A "server farm" is a collection of computers providing web services. Organizations running state-of-the-art server farms (such as Google, Amazon, and Microsoft) are somewhat close-mouthed about the details of their servers, and the specifications of server farms change constantly (so most of the information you find on the web is outdated). However, the specifications are amazing and should convince anyone that there is more to programming than simply computing a few numbers on a laptop:

- Google uses about a million servers (each more powerful than your laptop) in 25 to 50 "data centers."
- Such a data center is housed in a warehouse that might measure 60m* 100m (that's about 200ft*330ft) or more.
- In 2011, the *New York Times* reported that Google's data centers draw about 260 million watts continuously (about the same amount of energy as Las Vegas).
- Assume a server machine to be a 3GHz quad-core with 24GB of main memory. That would imply about $12*10^{15}$Hz of compute power (about 12,000,000,000,000,000 instructions per second) with $24*10^{15}$ bytes of main memory (about 24,000,000,000,000,000 8-bit bytes), and maybe 4TB of disk per server, giving $4*10^{18}$ bytes of storage.

We may be underestimating the amounts, and by the time you read this, we almost certainly are. In particular, efforts to minimize energy usage seem to be driving machine architectures toward more processors per server and more cores per processor. A GB is a gigabyte, that is, about 10^9 characters. A TB, a terabyte, is about 1000GB, that is, about 10^{12} characters. A PB, a petabyte (that is, 10^{15} bytes), is becoming a more common measure. This is a pretty extreme example, but every major company runs programs on the web to interact with its users/customers. Examples are Amazon (book and other sales), Amadeus (airline ticketing and automobile rental), and eBay (online auctions). Millions of companies, organizations, and individuals also have a presence on the web. Most don't run their own software, but many do and much of that is not trivial.

The other, and more traditional, massive computing effort involves accounting, order processing, payroll, record keeping, billing, inventory management, personnel records, student records, patient records, etc. – the records that essentially every organization (commercial and noncommercial, governmental and private) keeps. These records are the backbone of their respective organizations. As a computing effort, processing such records seems simple: mostly some information (records) is just stored and retrieved and very little is done to it. Examples include

- Is my 12:30 flight to Chicago still on time?
- Has Gilbert Sullivan had the measles?
- Has the coffeemaker that Juan Valdez ordered been shipped?
- What kind of kitchen chair did Jack Sprat buy in 1996 (or so)?
- How many phone calls originated from the 212 area code in August of 2012?
- What was the number of coffeepots sold in January and for what total price?

The sheer scale of the databases involved makes these systems highly complex. To that add the need to respond quickly (often in less than two seconds for individual queries) and to be correct (at least most of the time). These days, it is not uncommon for people to talk about terabytes of data (a byte is the amount of memory needed to hold an ordinary character). That's traditional "data processing" and it is merging with "the web" because most access to the databases is now through web interfaces.

This kind of computer use is often referred to as *information processing*. It focuses on data – often lots of data. This leads to challenges in the organization and transmission of data and lots of interesting work on how to present vast amounts of data in a comprehensible form: "user interface" is a very important aspect of handling data. For example, think of analyzing a work of older literature (say, Chaucer's *Canterbury Tales* or Cervantes' *Don Quixote*) to figure out what the author

actually wrote by comparing dozens of versions. We need to search through the texts with a variety of criteria supplied by the person doing the analysis and to display the results in a way that aids the discovery of salient points. Thinking of text analysis, publishing comes to mind: today, just about every article, book, brochure, newspaper, etc. is produced on a computer. Designing software to support that well is for most people still a problem that lacks a really good solution.

1.5.6 A vertical view

It is sometimes claimed that a paleontologist can reconstruct a complete dinosaur and describe its lifestyle and natural environment from studying a single small bone. That may be an exaggeration, but there is something to the idea of looking at a simple artifact and thinking about what it implies. Consider this photo showing the landscape of Mars taken by a camera on one of NASA's Mars Rovers:

If you want to do "rocket science," becoming a good programmer is one way. The various space programs employ lots of software designers, especially ones who can also understand some of the physics, math, electrical engineering, mechanical engineering, medical engineering, etc. that underlie the manned and unmanned space programs. Getting those two Rovers to drive around on Mars for years is one of the greatest technological triumphs of our civilization. One (*Spirit*) sent data back for six years and the other (*Opportunity*) is still working at the time of writing and will have its tenth anniversary on Mars in January 2014. Their estimated design life was three months.

The photo was transmitted to earth through a communication channel with a 25-minute transmission delay each way; there is a lot of clever programming and advanced math to make sure that the picture is transmitted using the minimal number of bits without losing any of them. On earth, the photo is then rendered using algorithms to restore color and minimize distortion due to the optics and electronic sensors.

The control programs for the Mars Rovers are of course programs – the Rovers drive autonomously for 24 hours at a time and follow instructions sent from earth the day before. The transmission is managed by programs.

The operating systems used for the various computers involved in the Rovers, the transmission, and the photo reconstruction are programs, as are the applications used to write this chapter. The computers on which these programs run are designed and produced using CAD/CAM (computer-aided design and computer-aided manufacture) programs. The chips that go into those computers are produced on computerized assembly lines constructed using precision tools, and those tools also use computers (and software) in their design and manufacture. The quality control for those long construction processes involves serious computation. All that code was written by humans in a high-level programming language and translated into machine code by a compiler, which is itself such a program. Many of these programs interact with users using GUIs and exchange data using input/output streams.

Finally, a lot of programming goes into image processing (including the processing of the photos from the Mars Rovers), animation, and photo editing (there are versions of the Rover photos floating around on the web featuring "Martians").

1.5.7 So what?

What do all these "fancy and complicated" applications and software systems have to do with learning programming and using C++? The connection is simply that many programmers do get to work on projects like these. These are the kinds of things that good programming can help achieve. Also, every example used in this chapter involved C++ and at least some of the techniques we describe in this book. Yes, there are C++ programs in MP3 players, in ships, in wind turbines, on Mars, and in the human genome project. For more applications using C++, see www.stroustrup.com/applications.html.

1.6 Ideals for programmers

What do we want from our programs? What do we want in general, as opposed to a particular feature of a particular program? We want *correctness* and as part of that, *reliability*. If the program doesn't do what it is supposed to do, and do so in a way so that we can rely on it, it is at best a serious nuisance, at worst a danger. We want it to be *well designed* so that it addresses a real need well; it doesn't really matter that a program is correct if what it does is irrelevant to us or if it correctly does something in a way that annoys us. We also want it to be *affordable*; I might prefer a Rolls-Royce or an executive jet to my usual forms of transport, but unless I'm a zillionaire, cost will enter into my choices.

These are aspects of software (gadgets, systems) that can be appreciated from the outside, by non-programmers. They must be ideals for programmers and we must keep them in mind at all times, especially in the early phases of development,

if we want to produce successful software. In addition, we must concern ourselves with ideals related to the code itself: our code must be *maintainable*; that is, its structure must be such that someone who didn't write it can understand it and make changes. A successful program "lives" for a long time (often for decades) and will be changed again and again. For example, it will be moved to new hardware, it will have new features added, it will be modified to use new I/O facilities (screens, video, sound), to interact using new natural languages, etc. Only a failed program will never be modified. To be maintainable, a program must be simple relative to its requirements, and the code must directly represent the ideas expressed. Complexity – the enemy of simplicity and maintainability – can be intrinsic to a problem (in that case we just have to deal with it), but it can also arise from poor expression of ideas in code. We must try to avoid that through good coding style – style matters!

This doesn't sound too difficult, but it is. Why? Programming is fundamentally simple: just tell the machine what it is supposed to do. So why can programming be most challenging? Computers are fundamentally simple; they can just do a few operations, such as adding two numbers and choosing the next instruction to execute based on a comparison of two numbers. The problem is that we don't want computers to do simple things. We want "the machine" to do things that are difficult enough for us to want help with them, but computers are nitpicking, unforgiving, dumb beasts. Furthermore, the world is more complex than we'd like to believe, so we don't really know the implications of what we request. We just want a program to "do something like this" and don't want to be bothered with technical details. We also tend to assume "common sense." Unfortunately, common sense isn't all that common among humans and is totally absent in computers (though some really well-designed programs can imitate it in specific, well-understood cases).

This line of thinking leads to the idea that "programming is understanding": when you can program a task, you understand it. Conversely, when you understand a task thoroughly, you can write a program to do it. In other words, we can see programming as part of an effort to thoroughly understand a topic. A program is a precise representation of our understanding of a topic.

When you program, you spend significant time trying to understand the task you are trying to automate.

We can describe the process of developing a program as having four stages:

- *Analysis:* What's the problem? What does the user want? What does the user need? What can the user afford? What kind of reliability do we need?
- *Design:* How do we solve the problem? What should be the overall structure of the system? Which parts does it consist of? How do those parts communicate with each other? How does the system communicate with its users?

- *Programming:* Express the solution to the problem (the design) in code. Write the code in a way that meets all constraints (time, space, money, reliability, and so on). Make sure that the code is correct and maintainable.
- *Testing:* Make sure the system works correctly under all circumstances required by systematically trying it out.

Programming plus testing is often called *implementation*. Obviously, this simple split of software development into four parts is a simplification. Thick books have been written on each of these four topics and more books still about how they relate to each other. One important thing to note is that these stages of development are not independent and do not occur strictly in sequence. We typically start with analysis, but feedback from testing can help improve the programming; problems with getting the program working may indicate a problem with the design; and working with the design may suggest aspects of the problem that hitherto had been overlooked in the analysis. Actually using the system typically exposes weaknesses of the analysis.

The crucial concept here is *feedback*. We learn from experience and modify our behavior based on what we learn. That's essential for effective software development. For any large project, we don't know everything there is to know about the problem and its solution before we start. We can try out ideas and get feedback by programming, but in the earlier stages of development it is easier (and faster) to get feedback by writing down design ideas, trying out those design ideas, and using scenarios on friends. The best design tool we know of is a blackboard (use a whiteboard instead if you prefer chemical smells over chalk dust). Never design alone if you can avoid it! Don't start coding before you have tried out your ideas by explaining them to someone. Discuss designs and programming techniques with friends, colleagues, potential users, and so on before you head for the keyboard. It is amazing how much you can learn from simply trying to articulate an idea. After all, a program is nothing more than an expression (in code) of some ideas.

Similarly, when you get stuck implementing a program, look up from the keyboard. Think about the problem itself, rather than your incomplete solution. Talk with someone: explain what you want to do and why it doesn't work. It's amazing how often you find the solution just by carefully explaining the problem to someone. Don't debug (find program errors) alone if you don't have to!

The focus of this book is implementation, and especially programming. We do not teach "problem solving" beyond giving you plenty of examples of problems and their solutions. Much of problem solving is recognizing a known problem and applying a known solution technique. Only when most subproblems are handled this way will you find the time to indulge in exciting and creative "out-of-the-box thinking." So, we focus on showing how to express ideas clearly in code.

Direct expression of ideas in code is a fundamental ideal of programming. That's really pretty obvious, but so far we are a bit short of good examples. We'll come back to this, repeatedly. When we want an integer in our code, we store it in an **int**, which provides the basic integer operations. When we want a string of characters, we store it in a **string**, which provides the most basic text manipulation operations. At the most fundamental level, the ideal is that when we have an idea, a concept, an entity, something we think of as a "thing," something we can draw on our whiteboard, something we can refer to in our discussions, something our (non–computer science) textbook talks about, then we want that something to exist in our program as a named entity (a type) providing the operations we think appropriate for it. If we want to do math, we want a **complex** type for complex numbers and a **Matrix** type for linear algebra. If we want to do graphics, we want a **Shape** type, a **Circle** type, a **Color** type, and a **Dialog_box**. When we want to deal with streams of data, say from a temperature sensor, we want an **istream** type (**i** for input). Obviously, every such type should provide the appropriate operations and only the appropriate operations. These are just a few examples from this book. Beyond that, we offer tools and techniques for you to build your own types to directly represent whatever concepts you want in your program.

Programming is part practical, part theoretical. If you are just practical, you will produce non-scalable, unmaintainable hacks. If you are just theoretical, you will produce unusable (or unaffordable) toys.

For a different kind of view of the ideals of programming and a few people who have contributed in major ways to software through work with programming languages, see Chapter 22, "Ideals and History."

Review

Review questions are intended to point you to the key ideas explained in a chapter. One way to look at them is as a complement to the exercises: the exercises focus on the practical aspects of programming, whereas the review questions try to help you articulate the ideas and concepts. In that, they resemble good interview questions.

1. What is software?
2. Why is software important?
3. Where is software important?
4. What could go wrong if some software fails? List some examples.
5. Where does software play an important role? List some examples.
6. What are some jobs related to software development? List some.
7. What's the difference between computer science and programming?
8. Where in the design, construction, and use of a ship is software used?
9. What is a server farm?

10. What kinds of queries do you ask online? List some.
11. What are some uses of software in science? List some.
12. What are some uses of software in medicine? List some.
13. What are some uses of software in entertainment? List some.
14. What general properties do we expect from good software?
15. What does a software developer look like?
16. What are the stages of software development?
17. Why can software development be difficult? List some reasons.
18. What are some uses of software that make your life easier?
19. What are some uses of software that make your life more difficult?

Terms

These terms present the basic vocabulary of programming and of C++. If you want to understand what people say about programming topics and to articulate your own ideas, you should know what each means.

affordability	customer	programmer
analysis	design	programming
blackboard	feedback	software
CAD/CAM	GUI	stereotype
communication	ideals	testing
correctness	implementation	user

Exercises

1. Pick an activity you do most days (such as going to class, eating dinner, or watching television). Make a list of ways computers are directly or indirectly involved.
2. Pick a profession, preferably one that you have some interest in or some knowledge of. Make a list of activities done by people in that profession that involve computers.
3. Swap your list from exercise 2 with a friend who picked a different profession and improve his or her list. When you have both done that, compare your results. Remember: There is no perfect solution to an open-ended exercise; improvements are always possible.
4. From your own experience, describe an activity that would not have been possible without computers.
5. Make a list of programs (software applications) that you have directly used. List only examples where you obviously interact with a program (such as when selecting a new song on an MP3 player) and not cases

where there just might happen to be a computer involved (such as turning the steering wheel of your car).

6. Make a list of ten activities that people do that do not involve computers in any way, even indirectly. This may be harder than you think!
7. Identify five tasks for which computers are not used today, but for which you think they will be used at some time in the future. Write a few sentences to elaborate on each one that you choose.
8. Write an explanation (at least 100 words, but fewer than 500) of why you would like to be a computer programmer. If, on the other hand, you are convinced that you would not like to be a programmer, explain that. In either case, present well-thought-out, logical arguments.
9. Write an explanation (at least 100 words, but fewer than 500) of what role other than programmer you'd like to play in the computer industry (independently of whether "programmer" is your first choice).
10. Do you think computers will ever develop to be conscious, thinking beings, capable of competing with humans? Write a short paragraph (at least 100 words) supporting your position.
11. List some characteristics that most successful programmers share. Then list some characteristics that programmers are popularly assumed to have.
12. Identify at least five kinds of applications for computer programs mentioned in this chapter and pick the one that you find the most interesting and that you would most likely want to participate in someday. Write a short paragraph (at least 100 words) explaining why you chose the one you did.
13. How much memory would it take to store (a) this page of text, (b) this chapter, (c) all of Shakespeare's work? Assume one byte of memory holds one character and just try to be precise to about 20%.
14. How much memory does your computer have? Main memory? Disk?

Postscript

Our civilization runs on software. Software is an area of unsurpassed diversity and opportunities for interesting, socially useful, and profitable work. When you approach software, do it in a principled and serious manner: you want to be part of the solution, not add to the problems.

We are obviously in awe of the range of software that permeates our technological civilization. Not all applications of software do good, of course, but that is another story. Here we wanted to emphasize how pervasive software is and how much of what we rely on in our daily lives depends on software. It was all written by people like us. All the scientists, mathematicians, engineers, programmers, etc. who built the software briefly mentioned here started like you are starting.

Now, let's get back to the down-to-earth business of learning the technical skills needed to program. If you start wondering if it is worth all your hard work (most thoughtful people wonder about that sometime), come back and reread this chapter, the Preface, and bits of Chapter 0 ("Notes to the Reader"). If you start wondering if you can handle it all, remember that millions have succeeded in becoming competent programmers, designers, software engineers, etc. You can, too.