Chapter 1

Introduction

1.1 Why learn programming?

You can tell I'm educated, I studied at the Sorbonne
Doctored in mathematics, I could have been a don
I can program a computer, choose the perfect time
If you've got the inclination, I have got the crime

Pet Shop Boys, Opportunities (1986)

This section explains what a computer program is, and why it is important for you not only to use *computer programs, but also to* write *them.*

When people apply for a job these days, their resume typically contains a section called *computer skills*. Items listed there might include *Word*, *Excel*, or *Powerpoint*. These are the names of *application programs*, programs that have been written by certain people (in the above cases, at Microsoft corporation) to be used by other people (for example, a sales representative).

The *computer skills* section might also list items like *HTML*, *Java*, or *C++*. These are the names of *programming languages*, languages used to instruct, or program, a computer. Using a programming language, *you* can write the programs that will subsequently be used by others, or by yourself.

A computer program is a list of instructions to be automatically processed by a computer. The computer itself is stupid—all the intelligence comes from the program. In this sense, a program for the computer is like a cookbook recipe for someone who cannot cook: even with very limited skills, impressive results can be obtained, through a step-by-step instruction.

Most people simply use programs, just like they use cookbooks. A sales representative, for example, needs application programs as tools for his work. The fact that you are reading this lets us believe that you potentially belong to the category of people who also need to write programs.

There are many reasons for writing programs. Some employer might pay for it, some bachelor course might require it, but ultimately, there is a deeper reason behind it that we plan to explain next. The upshot is that nowadays, you cannot be a serious engineer, let alone a serious scientist, without at least some basic programming skills. Even in less serious contexts, we can recommend to learn programming, because it can bring about a lot of fun and satisfaction.

In the twentieth century, computers have revolutionized the way science and engineering are done. To be more concrete, we will underpin this with an example from mathematics. You probably don't expect math to be mentioned first in connection with computers; indeed, many mathematicians still use paper and pencil on a daily basis. But *what* they write down has changed. Before computers were available, it was often

necessary to write down actual numbers, and to perform calculations with them by hand. This happened not so much in writing proofs for new theorems, but in the process of *finding* these theorems. This process often requires to go over many concrete examples, or counterexamples, in order to see certain patterns, or to discover that some statement is false. The computer has tremendously accelerated this process by taking over the routine work. When you look at a mathematician's notepad today, you still find greek letters and all kinds of strange symbols, but most likely no numbers larger than ten.

There is one topic that nicely illustrates the situation, and this is the search for the *Mersenne primes*. In 1644, the French monk and mathematician Marin Mersenne established the following claim.

> **Mersenne's Conjecture.** *The numbers of the form $2^n - 1$ are prime numbers for $n = 2, 3, 5, 7, 13, 17, 19, 31, 67, 127, 257$, but for no other number $n < 257$.*

Mersenne corresponded with many of the leading mathematicians at that time, so his conjecture became widely known. Up to $n = 7$, you can verify it while you read this, and in 1644, the conjecture was already verified up to $n = 19$.

It took more than hundred years until the next exponent on Mersenne's list could be verified. In a letter to Bernoulli published in 1772, Leonhard Euler proved that $2^{31} - 1 = 2147483647$ is a prime number. But in 1876, another hundred years later, Mersenne posthumously received a heavy blow. Edouard Lucas proved that $2^{67} - 1 = 147573952589676412927$ is *not* a prime number (Lucas showed his passion for large numbers also when he invented the *Tower of Hanoi* puzzle). Lucas's proof does not work the way you would expect: it does *not* exhibit a prime factor of $2^{67} - 1$ (the most direct way of proving that a number is not prime), but it uses a clever indirect argument invented by Lucas in the same year. The factorisation of $2^{67} - 1$ remained unknown for another 25 years.

In 1903, Frank Nelson Cole was scheduled to give a lecture to the American Mathematical Society, whose title was 'On the Factorisation of Large Numbers'. Cole went to the blackboard, and without saying a single word, he first wrote down a calculation to obtain $2^{67} - 1$ by repeated multiplication with two. He finally had the number

```
147573952589676412927
```

on the blackboard. Then he wrote down another (much more interesting) calculation for the product of two numbers.

```
761838257287 x 193707721
------------------------
    761838257287
   6856544315583
    2285514771861
     5332867801009
```

```
        5332867801009
         5332867801009
          1523676514574
            761838257287
--------------------
147573952589676412927
```

Cole had proved that $2^{67} - 1 = 761838257287 \cdot 193707721$, making the result of Lucas believable to everybody: $2^{67} - 1$ is not a prime number! He received standing ovations for this accomplishment and later admitted that he had worked on finding these factors every Sunday for the last three years.

Today, you can start a computer algebra program on your computer (a popular one is `Maple`), type in

```
ifactor(2^67-1);
```

and within less than a second get the output

```
(761838257287)(193707721)
```

To summarize: hundred years ago, a brilliant mathematician needed three years to come up with a result that much less brilliant people (we are not talking about you) could get in less than a second today, using a computer and the right program. This seems disturbing at first sight, and thinking about the precious time of his life Cole devoted to the problem, you may even feel sorry for him. You shouldn't; rather, the story has three important lessons in store.

Tool skills. Lesson one is that Cole's calculations were extremely difficult, given the tools he had (paper, pencil, and probably very good mental arithmetic). Given the tools *you* have (the computer and a computer algebra program called `Maple`), Cole's calculations are easy routine. We are sure that Cole would feel sorry for anyone using these new tools only to reproduce some hundred-year old calculation. Useful new tools lead to new possibilities and challenges. On the one hand, this *allows* you to do more than you could do before; on the other hand it also *forces* you to do more if you want to keep up with the developments. Whatever you do, nowadays you must acquire and maintain at least some basic knowledge of computers and application programs.

Problem skills. Lesson two is that tool skills alone would not have helped Cole to factor $2^{67} - 1$. Cole also was a good mathematician who knew a lot of theory he could use to save calculations. This is the reason why he "only" needed three years.

Even nowadays, computers and application programs are not everything. There are problems that are as far from a solution as they were in pre-computer days. For example, we still cannot characterize the exponents n for which the number $2^n - 1$ is a prime number. We don't even know whether there are infinitely many such Mersenne primes.

If you plan to make a contribution here, you should not buy a faster computer with the latest version of `Maple`, but study math. Even in the case of problems for which computers can really contribute to (or actually find) the solution, you typically need to have a deep understanding of the problem in order to know *how* to use the computer. If you want to become an engineer or a scientist, you must acquire and maintain a profound knowledge about the problems you will be dealing with. This fact was true hundred years ago, and it is still true—computers have not yet learned to solve interesting problems by themselves.

Programming Skills. Lesson three is one that Cole did not live to see: nowadays, problem-specific knowledge can be turned into problem-specific computer programs. That way, the state of the art concerning Mersenne primes has advanced quite far. It turned out that Mersenne had made five mistakes: $n = 67$ and $n = 257$ in Mersenne's list do not lead to prime numbers; on the other hand, Mersenne had "forgotten" the exponents $n = 61, 89$ and 107.

As of September 2006, we know 44 Mersenne primes, the largest of which has an exponent of $n = 32,582,657$.[1] But don't believe that this one was found with off-the-shelf programs. Even after you have acquired the whole known theory of Mersenne primes, you won't be able to find the 45th Mersenne prime without writing a program yourself, crammed with the knowledge you have.

Problems occuring in the daily life of an engineer or a scientist are often not easy to solve, even with a computer at hand. In order to attack them, you need *tool skills* for the routine calculations, and *problem skills* to understand and extract the aspects of the problem that can in principal be solved by a computer. But in the end, you need *programming skills* to actually do it.

The art of computer programming. To conclude this section, let us be honest: for many people (including the authors of this book), the process of writing programs has some very non-utilitarian aspects as well. We have mentioned two of them before: fun and satisfaction. We could add mathematical beauty and ego boost. In one way or another, every passionate programmer feels at least a little bit like an artist.

The prime advocator of this view on programming is Donald E. Knuth. He is the author of a monumental and seminal series of seven books entitled *The Art of Computer Programming*. Starting with Volume I in 1968, three of the seven volumes are published by now. Drafts of Volume IV circulate since 2005, and the planned release date of Volume V is 2015 (it should be added that Knuth was born in 1938, and on his webpage `http://www-cs-faculty.stanford.edu/~knuth/taocp.html`, he at least implicitly mentions the possibiliy that Volumes VI and VII will not be written anymore).

Let Knuth have the final say here (a quote from the beginning of Volume I):

> *The process of preparing programs for a digital computer is especially*

[1] see `www.mersenne.org`

attractive, not only because it can be economically and scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music.

1.2 How to run a program

> *In Paris they just simply opened their eyes and stared when we spoke to them in French! We never did succeed in making those idiots understand their own language.*
>
> *Mark Twain, The Innocents Abroad (1869)*

This section explains what it really means to "write a program", and how you enable the computer to run it. For this, we describe the ingredients involved in the process: the editor, *the* compiler, *the* computer *itself, and the* operating system. *Computer, compiler and operating system together form the* platform *on which you are writing programs.*

1.2.1 Editor

Writing a program is not so different from writing a letter. One composes a text, that is, a (hopefully) meaningful sequence of characters. Usually, there are certain conventions on how such a text is structured, and the purpose of the text is to transport information.

What has been said so far applies to both letters and programs. But when writing a program, there is another aspect that has to be taken into account: A program has to be "read" by a computer, meaning that it must be available to the computer in electronic form. In the future, we might be able to orally dictate the program to the computer, but nowadays, the common way is to use a keyboard and simply type it in. An *editor* is an application program that allows you to display, modify, and electronically store such typed-in text. The use of editors is not restricted to programming, of course. With some still existing romantic exceptions, even letters are composed using editors such as *Word*.

1.2.2 Compiler

Making a program available to the computer in electronic form is usually not enough. The *machine language* a computer can understand directly is very primitive and quite different from natural languages.

Writing the programs in machine language is no viable alternative, since that would require to break the program into a large number of primitive instructions that the computer can understand. This is like telling your friend to come over for dinner by telling her which muscles to move in order to get to your place.

Moreover, machine languages vary considerably between different computers. That is, in order to use a program written for one specific computer A on a different computer B, one first has to translate the program from the machine language of A to the

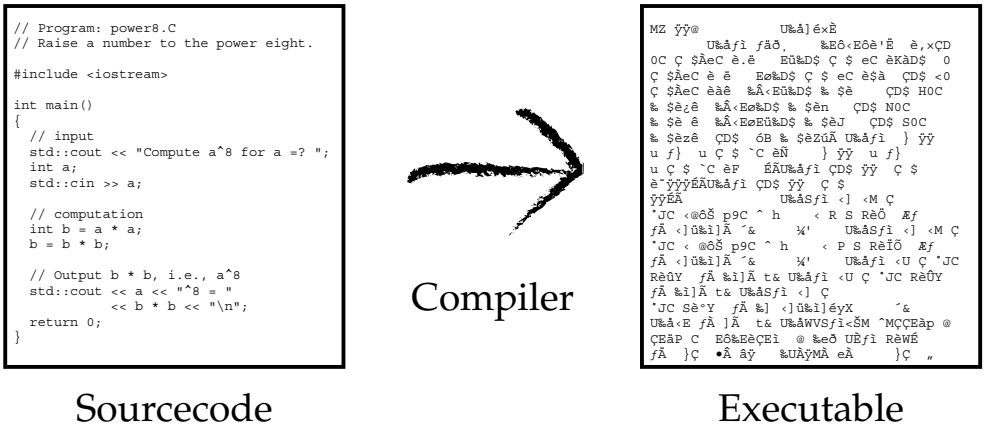

Figure 1: *A compiler translates the sourcecode into an executable program.*

machine language of B. This process, called *porting*, can be very cumbersome if the machine languages of A and B are substantially different. Also, porting can only be done with a detailed knowledge of the peculiarities of the involved computers. But this type of knowledge is not generally worthwhile to acquire, as it is tied to one very specific computer. As soon as this computer is replaced by another one, major parts of such computer-specific knowledge become worthless and have to be rebuilt from scratch.

To reduce this undesirable entanglement of computers and programs, and to allow us to write programs in less primitive language, (high-level) programming languages have been developed. These are standardized languages that form a kind of compromise between natural languages and machine language. Indeed, the use of the word "compromise" is justified because there are two conflicting goals: On the one hand, we would like to write programs in a language that is as close to natural language as possible. On the other hand, we have to make the computers understand the programming language as well; this task is obviously much easier if the programming language is close to machine language.

What does it mean "to make the computers understand the programming language"? In the end, any program has to be translated into machine language. The process of this translation is called *compilation*. Now you will probably ask: "Where is the benefit of this whole programming language concept? In order to do the translation I still have to know all these computer-specific details, don't I?" Right. If you would have to translate the program yourself. The key is: You are not supposed to translate it yourself. Instead, let a program do it for you. Such a program is referred to as a *compiler*; it translates a given program in a programming language, the *sourcecode*, into a program in machine language, the *executable*. See Figure 1 for an illustration.[2]

In summary: The big benefit of (high-level) programming languages is that they

[2]The picture of the executable is somewhat inappropriate, since it does not show what the computer gets to see after compilation, but rather what *you* might see when you (accidentally) load the executable into the editor. The main point that we are trying to make here is that the executable is not human-readable.

abstract from the capabilities of specific computers. Programs written in a high-level language can be run on all kinds of computers, as long as a compiler for the language is available on the particular computer.

1.2.3 Computer

If you are not interested in writing compilers, it is not necessary to understand in detail how a computer works. But there are some basic principles behind the design of most computers that *are* important to understand. These principles form the *von Neumann architecture*, and they are important, since almost all programming languages are tailored to the von Neumann architecture.

Any computer with von Neumann architecture has a *random access memory* (RAM, or simply main memory), and a *central processing unit* (CPU, or simply processor). The main memory stores the program to be run, but also data that the program requires as input, and data that the program produces as output. The processor is the "brain" of the computer: it executes the program, meaning that it carries out the sequence of instructions prescribed by the program in machine language.

Main memory. You can think of the computer's main memory as a long row of switches, each of them being either on or off. During program execution, switches are flipped. At any time, the memory content—the current positions of all switches—defines the *program state*. The program state completely determines what happens next. Conceptually, we also consider user input and program output as part of the program state, even though the corresponding "switches" might be in the user's brain, or on printed paper.

Since modern computers are capable of flipping several switches (32, say) at the same time, consecutive switches are accordingly grouped into *memory cells*. The positions of all switches in the cell define the *content* of the cell; in more abstract terms, the switches are called *bits*, each of them capable of storing one of the numbers $\{0, 1\}$. In this sense, you can interpret the content of a memory cell as a binary number with, for example, 32 digits. We also say that we have a 32-bit machine, or a 32-bit system.

Each memory cell is uniquely identified by its *address*. You can think of the address simply as the position of the memory cell in the list of all memory cells.

To look up bit values, or to flip bits within a specific memory cell, the cell has to be *accessed* through its address. Think of a robot arm with 32 fingers that you can tell to move to memory cell number 17.

The term *random access* refers to a physical property of the computer's memory: the time it takes to access a cell (to "move to its bits") is the same for all cells; in particular, it does *not* depend on the address of the cell. When you think in terms of the robot arm analogy, it becomes clear that random access cannot be taken for granted. It is not necessary to discuss the physical means by which random access is realized; the important point here is that random access frees us from thinking about where to store

a data item in order to access it efficiently.

Processor. You can think of the computer's processor as a box that is able to load and then execute the machine language instructions of a program in order. The processor has some memory cells of its own, called registers, and it can transfer data from the computer's main memory to its registers, and vice versa. The register contents are also part of the program state. Most importantly, the processor can perform a fixed set of simple operations (like adding or subtracting register contents), directly corresponding to the machine langauge instructions. This is where the functionality of the whole program comes from in the end. Even very complicated and useful programs can be put together from a simple set of machine language instructions.

A single instruction acts like a mathematical function: given the current program state, a valid instruction generates a new and well-defined next program state. This implies that any sequence of instructions, and in particular the whole program has a well-defined behavior, depending on the initial program state.

1.2.4 Operating system

We have seen that in order to write a program and run it, you first have to start an editor, type in the program, then call the compiler to translate the program into machine language, and finally tell the computer to execute it. In all this "starting", "calling" and "telling", you rely on the computer's *operating system* (OS), a program so basic that you may not even perceive it as a program. Popular operating systems are *Windows*, *Unix*, *Linux*, and *Mac OS*.

For example, whether you start the editor by clicking on some icon, or whether you type a command for this somewhere, the operating system makes sure that the editor program is loaded into the main memory, and that the processor starts executing it. Similarly, when you store your written program, the operating system allocates space for it on the hard disk and associates it with the file name you have provided.

A computer without operating system is like a car without tires, and most computers you can buy come with a pre-installed operating system. It is important to understand, though, that the operating system is not inextricably tied to the computer: you can take your "Windows PC" and reinstall it under Linux.

1.2.5 Platform

The computer, its operating system and the compiler are together referred to as the *platform* on which you are writing your programs. The editor is not part of the platform, since it does not influence the behavior of the program.

In an ideal world, there is no need for you to know the platform when you are writing programs in a high-level programming language. Recall that the plan is to delegate the platform-specific aspects to the compiler. A typical such platform-specific aspect is the

size of a memory cell, i.e. the number of bits that can be manipulated together. This is mostly 32 these days, but for some computers it is 64, and for very primitive computers (like they are used in smart cards, say), it can be much less than 32.

When you are using or relying on machine-oriented features of the programming language, platform-specific behavior might be the result. Many high-level programming languages have such low-level features to facilitate the translation into *efficient* machine language.

Your goal should always be to write *platform-independent* code, since otherwise, it may be very difficult to get your program to run on another computer, even if you have a compiler for that computer. This implies that certain features should be avoided, even though it might seem advantageous to use them on a specific platform.

1.2.6 Details

Von Neumann's idea of a common memory for the program *and* the data seems obvious from today's point of view, but the earliest computers like Konrad Zuse's Z3 didn't work that way. In the Z3, for example, the memory for the program was a punch tape, decoupled from the input and output device, and from the main memory.

An interesting feature of the von Neumann architecture is that it allows self-modifying programs. These are popular among the designers of computer viruses, for example.

The von Neumann architecture with its two levels of memory (main memory and processor registers) is an idealized model, and we are implicitly working under this model throughout the course.

The reality looks more complicated. Modern computers also have a *cache*, logically belonging to the main memory, but allowing much faster access to memory cells (at the price of a more elaborate and expensive design). The idea is that frequently needed data are stored in the cache to speed up the program.

While caching is certainly a good thing, it makes the life of a programmer more difficult: you can no longer rely on the fact that access time to data is independent from where they are stored. In fact, to get the full performance benefit that caching can offer, the programmer has to make sure that data are accessed in a *cache-coherent* way. Doing this, however, requires some computer-specific knowledge about the cache, knowledge we were originally trying to avoid by using high-level programming languages. Luckily, we can often ignore this issue and (successfully) rely on the automatic cache management being offered. There is also a theoretical model for so-called *cache-oblivious* algorithms, in which an algorithm explicitly does not know the parameters of the cache. Algorithms which are efficient under this model, are (in a certain sense) efficient for any concrete cache size.

In real-life applications, we also observe the phenomenon that the data to be processed are too large to fit into the computer's main memory. Operating systems can automatically deal with this by logically extending the main memory to the hard disk. However, the *swapping* that takes place when hard disk data to be accessed are transfered to the main memory incurs a severe performance penalty, much worse than poor

cache usage. In this situation, it is often useless to rely on the automatic mechanisms provided by the operating systems, and the programmer is challenged to come up with *input/output efficient* programs.

Even when we extend the von Neumann architecture to include several layers of memory, there are computers that don't fit in. Most notably, there are *parallel computers* with more than one processor. Writing efficient programs for such a computer is a task entirely different from programming for the von Neumann architecture. To take full advantage of the parallelism, programs have to be decomposed manually into independent parts, each of which is then run by one of the processors. In many cases, this is not at all a straightforward task, and specialized programming languages have to be used.

A recent successful alternative to parallel computers are networks of single-processor computers. You can even call this a computer architecture. Finally, there are *quantum computers* that are based on completely different physical principles than the von Neumann architecture. "Real" quantum computers cannot be built yet, but as a theoretical model, quantum computers exist, and algorithms are already being developed in this promising model of computation.