Elements of Programming

M.H. van Emden

Fourth Edition

Typeset in Tex; formatted in Latex. Figures in Omnigraffle™.
Cover design by Pink Sheep Media.

Published by Andromeda Research Associates, Ltd.,
a corporation registered in British Columbia, Canada.

First edition August 2007
Second edition August 2008
Third edition August 2009
Fourth edition December 2015

Contents

Preface

Many are the ways of getting into programming. One route is to start tinkering with the html source code behind web pages. This can lead to learning about css, then JavaScript and other convenience languages like Python and Ruby and then perhaps tackling Java. Another route is to take a programming course in first year of college. One might think that for such a course a convenience language is more suitable than a bare-metal language like C.

But think again: it depends on where one is in college. Students in science, engineering, and mathematics have selected themselves by having an above average attention span and by having above-average analytical and problem-solving abilities. Reading and writing your first programs in C is no more challenging than what you encounter in a calculus or physics course. Plus, you get the benefit of starting programming at the professional end.

Whatever route you are following, you will find this book useful.

In structuring this book I have been guided by certain similarities between flight instruction and teaching programming. Flight instruction for novices begins with a *familiarization flight*. Usually only a half-hour lesson, but sometimes repeated for unusually panicky pupils. The familiarization lesson does not concentrate on anything in particular and only aims at getting the novice used to being in the air. To make this book suitable for novice programmers, its first part is familiarization. It is dedicated to make the novice feel at home with simple programs. The book is also useful to programmers with experience in other languages who want to learn C. They will probably skip Part I, although Chapter 3 may be found useful.

Only after familiarization in flight instruction come lessons that concentrate on specific topics, such as turns (climbing or descending), take-off, and landing. Part II of this book, *Base* is the programming counterpart of this stage of instruction. Part III gives a glimpse of the core of programming: *Algorithms*. After this, the reader may become curious about what is known about problem solving, that crucial skill needed for programming anything non-trivial. I have attempted to give some hints in Part IV, *Method*.

Department of Computer Science
University of Victoria
December 2015

Acknowledgements

I had the good fortune to grow up in three distinctive programming cultures: the Mathematical Centre in Amsterdam, the Lisp group in the IBM T.J. Watson Research Center, and the Department of Machine Intelligence in the University of Edinburgh. Though all of these entities have ceased to exist, I trust I am not the only surviving beneficiary.

If this book is better than others, it is due to my choice of those who were, often without knowing it, my teachers: H. Abelson, J. Bentley, W. Burge, R. Burstall, M. Cheng, A. Colmerauer, T. Dekker, E. Dijkstra, D. Gries, C. Hoare, D. Hoffman, N. Horspool, B. Kernighan, D. Knuth, R. O'Keefe, P. Plauger, R. Popplestone, F. Roberts, G. Sussman, A. van Wijngaarden, N. Wirth.

I am grateful to the Department of Electrical and Computer Engineering at the University of Victoria for insisting on returning from objects-first to functions-first; to Sherwin Arnott of Pink Sheep Media, for his cover design and his enthusiastic support of this project; to Jason Corless, Mark Halpern, Philip Kelly, Michael Levy, Belaid Moa, Julian Subda, and Jim Uhl for expert and helpful comments.

Dan Hoffman deserves special mention. He combines deep expertise in the subject matter with the intense interest that an instructor has for the text selected for his course. In addition he turned out to be an expert copy editor. I have gratefully accepted many of his suggestions.

Chapter 1

Introduction

Programming is part of information technology, which, in turn, is part of the larger phenomenon of automation. We speak of "automation" whenever a machine is used to replace the work of a human. Scholars have undoubtedly found examples of automation in the renaissance, in the middle ages, in antiquity ... Let's skip all that and use as natural starting point the time when automation first became a *problem*: the early 19th century, when Ned Ludd and his gangs of cottagers rampaged through the land to smash the spinning and weaving machines that had deprived them of their livelihood.

Automation continued in spite of such opposition. Jacquard invented the *programmable loom* early in the 19th century. The pattern to be woven was determined by instructions coded by punching holes in paper cards. Change the cards, and you change the pattern. The programmer of those times translated an artist's design into a pattern of punchings.

1.1 Computers

In computation the counterparts of a machine loom were the mechanical calculators invented in the 17th century independently by Schickard, Pascal, and Leibniz. With such a machine you could add large numbers by merely turning a crank. Though it took the brain work out of computation, you still had to turn the crank. And of course, you still had to decide what numbers to add.

Performing a complex calculation is similar to operating a mechanical loom to produce a desired pattern. Charles Babbage combined the idea of a mechanical calculator with Jacquard's. This resulted in 1834 in a design of a computer that was both *programmable* and *digital*. Realization of the design proved too hard. The automation of computation came to a standstill.

In manufacturing, however, automation continued. As important as automation was the *organization* of manufacturing. Even before the time of Ned Ludd, Adam Smith described an exquisitely organized mass production process for pins (the kind you use by the dozen to hold pieces of cloth together). Henry Ford is famous for

applying the same principle on a larger scale in the early 20th century. By this time, clerical work such as the processing of mail orders and bank transactions was done on a large scale. This motivated the organization of clerical work along the same principles as those of mass production in manufacturing.

During the second world war scientific computations reached such a scale that the same organizational techniques were needed that had earlier been applied to the processing of mail orders and bank transactions. The ideas of Babbage were revived. This time, however, there was adequate funding for the development of such devices. Moreover, electronics removed some of the obstacles that had prevented Babbage from realizing his design.

After a bewildering variety of experimental designs there emerged in the 1950s the *electronic* programmable digital computer. Although physically it has changed beyond recognition since then, the main outline of its logical structure has remained the same. It has a *processor* containing registers for storing data and circuits for performing operations on these data. It has *memory* consisting of registers similar to those in the processor. Memory can be used to store input and output data, for intermediate results, and for the instructions that control the processor. The collection of instructions in memory that control the processor is called a *program*.

This logical structure is shared by a wide range of physical systems. The range includes a sliver of silicon the size of a fingernail that sells in quantity for less than a dollar. It includes a laptop computer, as well as a workstation. It also includes a million-dollar supercomputer containing many processors and memories that are controlled by a single program.

1.2 Programs

A typical processor has a repertoire of dozens of instructions; often over a hundred. Efficiency demands that an instruction performs an extremely simple operation. As a result, programs consist of large numbers of instructions obtained by splitting up the problem of interest in excruciating detail. During the first decade programming was done this way.

Now that alternatives exist, it is seen as extremely expensive to have humans write such programs. Instead, computers are usually programmed in a *high-level language*. Examples of such languages are C, C++, Java, and C#. Many other high-level languages exist. Thus "program" can mean a text in a high-level language as well as the instructions in memory that control the processor.

In spite of the existence of high-level languages, computers still need to be controlled by instructions. How do we cause a suitable sequence of instructions to be written in the right memory locations so that the computer does what we want it to do?

The main tool is the *compiler*, which translates a high-level language program into instructions. As this translation is a large and complex task, the compiler is itself a computer program. In the simplest set-up, the same computer is used for running the compiler and for running the programs that you write.

A high-level language makes a computer easier to program by hiding the intricate hardware of the actual computer behind a simpler abstraction. Independently of how memory is actually organized, a high-level language allows us to regard it as a large collection of non-overlapping cells. These cells differ only in size, so they can be laid out in any unoccupied part of memory. The size of the cell depends on the amount of information a data item carries. The cell is smallest when the data item only distinguishes between the 128 or 256 values of the character set in use. The cell is larger when it can contain any whole number from 0 to $4,294,967,295$ (which is $2^{32} - 1$), to take as example a common range of values.

As the content of the cell can be changed, we speak of *variables* rather than "memory cells". A programming language allows us to set aside a certain part of memory to be used as a collection of named variables, and to ignore the rest of memory. As the content of the variables can change, we can only speak of any particular content when the memory is in a certain *state*. Accordingly we refer to the content of the variables collectively as *the* state.

We first study a simple kind of program that executes in the following stages:

1. **Configure:** arrange the state so that a suitable number of various types of variables are available,
2. **Initialize:** cause the input data to become the contents of selected variables (this way the input data become available for computation),
3. **Compute:** Change the state in one or more steps so that selected variables contain the desired result, and
4. **Output:** cause the contents of the selected variables to be displayed on a screen or printed on paper.

Programs of this simple kind are a good starting point for many practically useful programs.

1.3 Algorithms

We have *automation* whenever a machine does work without needing to be controlled by a human. The jar of jam on my breakfast table was filled, labeled, and capped by such a machine. In early generations of such machines this automatic behaviour was a consequence of the way in which the machine was put together. It was not possible to point to any particular part responsible for its behaviour. Changing the machine's behaviour required extensive rebuilding.

Nowadays it is more likely that we can point to a changeable part of the machine that causes its behaviour. Change this part, and the machine changes its behaviour. Likely as not, that part is a computer chip with a program stored in its memory.

This new style of implementing automation has made it easier to design machinery because the automatic behaviour is created by writing a program for a

computer. By implementing automation in this way, *programming has become the essence of automation*. All we need to do to achieve automation is to say precisely what we want done. This is the purpose of the program.

Explaining in writing how to do something is not a novelty brought on by the computer age. Since times immemorial, alchemists, witches, mothers, grandmothers, and Cordon Blue chefs have passed on their knowledge by writing recipes. Knitting patterns are also examples of pioneering attempts at precise written instructions.

Writing a program is deceptively like writing a recipe or knitting pattern. The purpose is to make clear the complete sequence of precisely defined steps to be followed. The similarity is deceptive because recipes and knitting patterns are executed by humans. True enough, these humans may curse you, the author, for forgetting a step or for thinking something obvious that turns out not to be so, but by and large they muddle through and often end up with something that the author would acknowledge as being close to the intended result.

Not so with programs. The machine that executes the program does not share with the author of the program any notion of what the intended result is supposed to be, nor does it have a repertoire of basic skills with which to improvise, should the need arise. The machine attempts to execute the instruction stated if it is unambiguously recognized as an exact member of its precisely circumscribed repertoire. If this instruction is not appropriate to the state it is in, the machine will halt without damage if we are lucky and might be destroyed otherwise. If the instruction is not recognized, then the machine aborts program execution in the way that the designer thought might cause least damage. Therefore programs for machines need to be written more precisely than instructions for humans.

So far we have only talked about programs. What about algorithms? A good cookbook may contain a thousand recipes, but that is still only a negligible proportion of everything one could concoct in the kitchen. A good cookbook is good not only because of the accuracy of the recipes, but also because of its selection of what is worth cooking or baking. So it is with programs. Programming expertise includes a collection of particularly worthwhile methods for doing things that a computer can do. There are many ways in which a program can be written to benefit from such a method. An *algorithm* is that which all such programs have in common.

An algorithm is a method of doing a task that can be so precisely described that it can control a machine with the result that the machine performs the task without need for supervision. The description of the algorithm is not the algorithm: the same algorithm can be described in different programming languages and in different ways in the same language.

1.3.1 Binary search: an example of an algorithm

To get a better idea of what we are talking about, let us look at an algorithm. An algorithm is a method for performing a task. As example of a task, consider the following. You have a stack of cards. Each card has one word written on it. The cards are ordered so that the words are in alphabetic order with the alphabetically

earliest on top. The task is to find out whether a given word occurs on a card in the stack.

This is a simplified version of a common data processing task. For example, the words could be the names of customers, while the customer's record could be written on the back of each card. As example of a method to perform this task when there may be many cards on the stack, we take *binary search*.

A first try For binary search we proceed as follows.

> Look at the word on the card halfway down the stack. If this is the word we are looking for, then we are done. Otherwise, the word we are looking at is alphabetically before or after the word we are looking for. In the first case we continue with the bottom part of the stack; otherwise with the top part.

This description would be enough to instruct a human who has not yet figured out the trick. But it is inadequate as an algorithm. The problematic aspects can be summarized as: incomplete and not well-defined, and each of these in several ways.

The description of the method is *incomplete*. The stack that we are working on gets smaller at every step. Surely, at some point "halfway down the stack" stops making sense. Another way in which the description is incomplete shows up when we ask what happens if the word is not on any of the cards. Whether or not this is a likely event, it must be accounted for. And it may well be a likely event: if the business is growing, it often happens that the word we are looking for is the name of a new customer. In such a situation it would be helpful if the method tells us where to insert a new card.

The description that we gave of the method is not *well-defined* in several ways. For example, what do some of these phrases *mean*? Take "look at ... halfway down the stack". Or "continue with the bottom part". Part of the problem is that even the description of the task is problematic: "to find out whether a word occurs on a card in the stack". Suppose a human assistant would take us at our word and hand the stack back to us while enunciating the word "yes". Though we got what we asked for, we would be more than a little annoyed at the implied "... and if you want that card, go find it yourself". It is annoying because the assistant has done the work of finding the card and then throws away the information gained by handing us back the whole pile in its original state. And even if the word does not occur in the stack, it may be useful to know where it would have occurred if it did. For example, that would be the place to insert a new card with that word.

Another try at binary search These shortcomings motivate improvements to both the task and algorithm descriptions. First the task.

Input There are two items of input: (1) A word called "input word", and (2) a stack of cards called "input stack". Each card has one word written on it. The cards are ordered so that the words are in alphabetic order with the alphabetically earliest on top. There is at least one card in the input stack.

Output We define the *insertion point* as the lowest card whose word is not alphabetically after the input word. The desired output is the input stack with a slip of paper visibly inserted immediately above the insertion point. If there is no insertion point; that is, if the input word is alphabetically before the top card, then the desired output is the input stack with the slip of paper on top.

The slip of paper ensures that we don't throw away the work done in our search for the input word. It also takes care of both eventualities: that the input word is or is not in the stack. The legalistic-sounding definition of the insertion point is engineered to be applicable whether or not there is a card with the input word in the input stack.

With this improved task description in hand we attempt a description of a method to perform the task.

Step 1 If the input word is alphabetically before the word on the top card, then halt, returning the input stack with the slip on top of the top card.

Step 2 If the word is on the bottom card or comes after it, then insert the slip below the bottom card, and halt.

Step 3 [We have not halted, so the input word is the word on the top card or is after it, or is before the word on the bottom card, if it is in the stack.] Insert a slip below the top card and a slip on top of the bottom card.

Step 4 [We have not halted, so we are holding a stack of cards with two slips in it.] If there is no card between the slips, then remove one and halt.

Step 5 [We have not halted, so there is at least one card between the slips, the input word is on or after the word of the card on top of the top slip and it is before the word on the card under the bottom slip.] Inspect the card halfway between the slips. If the word on this card is after the input word, then place the bottom slip on top of this card; otherwise, place the top slip on top of this card. Go to Step 4.

Though it's a lot more verbose than our first attempt, it is still not totally precise. (What's "the card halfway between the slips" when there are two, or indeed any even number?) One might despair at the prospect of an even more legalistic and convoluted description. Despair not. See Program 13.2, which is short, neat, and absolutely precise. Such is the joy of programming.

1.3.2 Characteristics of algorithms

Our work on the binary search example serves to highlight the characteristics of algorithms. An algorithm is something like a procedure, a method, or a recipe. It is more systematic than these in that an algorithm must have the following characteristics: *finiteness, input, output, definiteness*, and *effectiveness*.

Finiteness Execution of an algorithm must terminate. This does not hold for all *programs*. For example, a program that controls a telephone exchange is designed to continue operation without ever stopping. To terminate would be an error. Such a program does not define an *algorithm*. But many parts of the intentionally non-terminating program do execute algorithms that need to terminate to ensure correct operation of the program of which they are part.

Input An algorithm typically acts on data obtained from *input*. A program for computing the decimal digits $3.1415926535\ldots$ of the number π does not qualify as an algorithm because it does not have input, and because it does not terminate. But modify it slightly to take as input the number of digits required, and it does qualify if it terminates after the specified number of digits.

Output An algorithm has a result, which needs to be communicated to the outside world by means of *output*.

Definiteness An algorithm specifies how to perform a task by means of a well-defined set of operations. Each of these operations is really a task in itself, which may need to be done by executing another algorithm. For example, in binary search we specify as operation "If the input word is alphabetically after the word". This assumes we know how to determine of two words whether they are the same and, if not, to determine which one is alphabetically earlier. Implicitly we assume that the words are finite. In that case, it is safe to assume that it is possible to determine this in a finite amount of time. It happens to require another algorithm. It is an important one, and student programmers don't always get it right.

At another point, the binary search algorithm specifies "Inspect the card halfway between the slips" if the number of cards between the slips is more than one. This specification lacks in definiteness. If the number of cards between the slips is odd, then indeed there is exactly one card answering this description. But if there is an even number, then there are two cards with equal claim to being halfway, or none, depending on how pedantic you want to be. In fact the algorithm only works if we take the upper one of these. This is at least one way in which our improved version of binary search is still defective.

Effectiveness Suppose one would include in an algorithm an operation of the form "If there is a solution to such and such a system of equations, then do this else do that." Whether such an operation is permissible in an algorithm depends on the system of equations.

1.3.3 Need for an algorithmic language

We saw that our second attempt at a binary search algorithm, though better than the first, was still not successful. The difficulties suggest that English or any other natural language is not suitable for anything beyond an intuitive approximation to an algorithm. We need an *algorithmic language*.

1.4 Languages

An example of an algorithmic language is the programming language in which the program is written that makes the computer do what we want. Thus, C, C++, Java, and C# are examples of algorithmic languages as well as being programming languages. But such languages are far from ideal as algorithmic languages. To make them into tools for getting the best performance out of a computer, any program expressing the algorithm is cluttered to some degree with extraneous detail. For example, an algorithmic language might only have two types of number, integer and fractional. Integers in C can be specified as being signed or not, and come in up to three different sizes; six combinations in all. Fractional numbers come in two different sizes. All this may help to enhance efficiency, but introduces extraneous detail into algorithm specification.

This, then is the Programmer's Dilemma: before starting to write a program, one needs to know what the algorithm is; to know the algorithm, one should have it written in an algorithmic language. Some authors advocate the use of a language that is purely algorithmic in the sense that it is only intended to specify algorithms. Because such a purely algorithmic language does not need to control a computer, specifications written in it need not contain any extraneous detail.

This suggests a way out of the Programmer's Dilemma: use a purely algorithmic language to specify the algorithm; then translate the specification to a programming language. A problem with this approach is that there is much disagreement about the conceptual basis of a purely algorithmic language. Some advocate that it be based on the mathematical concept of *function*. Others disagree, arguing that *formal logic* should be the fundament. Both sides on this issue agree that the concept of *state* is too close to a computer to sufficiently distinguish an algorithmic language from a programming language. For others, states and state transitions are the essence of algorithms.

This stand-off has existed for decades. It is not surprising that programming languages have evolved to become a hybrid. One can choose a programming style that emphasizes the clarity or brevity with which algorithms are expressed. Or one can write the program to optimize execution speed or memory use. These four requirements, clarity, brevity, speed, and memory use, usually conflict.

By emphasizing clarity, if necessary at the expense of brevity, speed, and memory use, one can use a programming language such as C, C++, Java, or C# in such a way that it is a reasonable approximation to a purely algorithmic language. In this book we do not introduce a purely algorithmic language, but use C. In this way we get programs that are both readable and executable.

An advantage of this approach is that you learn the elements of programming at the same time as learning a programming language that is widely used. Moreover, C has influenced the design of Java and, via Java, the design of C#. C has even become a part of C++. Though languages such as Javascript, Perl, and Python are referred to as "scripting languages", they are programming languages as well. The elements of programming that you learn here are also useful in learning to program in these languages.

Part I

Familiarization

Chapter 2

The first two programs

Programming is one of the things you learn best by doing. I will get you started as soon as possible running your own programs and experimenting with them so you can get them to do interesting things.

Whatever one sets out to do, something else needs to be done first. So it is in computing. However much we would now like the computer to *compute* something, we need to be able to get the computer to *communicate* first. Computation happens in the processor, which only communicates with memory. We need to be able write our data into memory (input) and to read selected parts of memory (output).

2.1 Prerequisites

Programming environment I recommend that you execute the example programs in this book on a computer and that you try variations of them. To do this, and to do the exercises, you need very little in the way of programming environment. All I assume is that you have a screen, a keyboard, and a computer with a C implementation.

Input is entered on a keyboard. Output is displayed in a window on the screen. Depending on system particulars, either the whole screen is dedicated to this window, or it is one of several windows. You need a text editor for writing new programs and for modifying existing ones.

The development cycle When we talk about a "program" in this book, we almost always mean a *source program*: a text written in the C programming language. When considering the details of using such a program, we have to distinguish three forms in which a program can exist: *source program*, *object program*, and *executable*.

Part of the home work or lab work consists of going through the following cycle:

1. *Create* or *revise* the source program with the editor.

2. *Compile* the source program to obtain the object program.

3. *Link* the object program to other object programs and load the result to obtain an executable program.

4. *Run* the executable.

Some or all of these steps can usually be combined into a single command-line entry or mouse click.

2.2 Output

Most of this book is concerned with what needs to happen inside the computer for it to be able to do something useful. However important these internal operations are, we should not lose sight of the fact that they have to be based on data (so there has to be *input*) and that we need to get a result in some form (so there has to be *output*).

In trying to find a program simple enough to serve as first example, we can leave out just about everything, but not output. Therefore the first program only performs output. In the next few pages I explain how this is done by Program 2.1*.

```
00 // A program to demonstrate output.
01 #include <stdio.h>
02 #include <math.h>
03
04 int main() {
05   printf("Good morning, everyone!\n");
06   printf("This is a number: %f\n", 3.1415926535);
07   printf("%d\n", 2+2);
08   printf("%s\n", "2+2");
09   double x;
10   printf("%f\n", x);
11   x = 1.0;
12   printf("%f\n", x);
13   printf("%f; this should be pi\n", 4*atan(x));
14   return 0;
15 }
```

Figure 2.1: A program with output only.

*When the text refers to "Program" $x.y$, please look up *Figure* $x.y$.

Line 00 The first line in the program in Program 2.1 is a *comment*. This is so because in a line containing in succession the two slash symbols

```
//
```

the slashes themselves and all text (if any) on the same line after the slashes is ignored by the compiler.

In general, comments are used to make programs readable. We should always keep in mind that the intended audience of source code is not only a computer, but also a human, you or someone else, who will need to debug or revise the code[†]. Surprisingly perhaps, writing new code is the exception in the programming profession. Most programmers' work consists in modifying existing programs. It is here that comments, carefully formulated and thoughtfully placed, are of crucial importance.

Line 01 The C programming language only provides core capabilities. Many essential facilities are not provided by the language itself, but by collections of programs called *libraries*. The C language standard includes a Standard Library.

As the Standard Library is large, one needs to specify which part one needs. In this program we need the part that deals with *input* and *output*. Accordingly, we write the *directive* `#include <stdio.h>`. Doing so gives the program access to the facilities in the part of the standard library for input and output.

Line 02 This line contains the directive to include the part of the standard library for mathematical functions such as square root, logarithms, the exponential functions, trigonometrical functions, and others. `stdio.h` and `math.h` are names of *header files*. These define the required parts of the library.

Line 03 A blank line has been inserted to improve readability.

Line 04 Programs consist mainly of *statements*: text that instructs the implementation to do something. In support of that there are directives and *definitions* that do not cause anything to happen, but serve to clarify the statements. Large programs have so many statements that they need to be grouped into units called *functions*. Functions have names and often deliver data.

When a program with many functions starts to execute, which one gets to go first? This is the function named `main`. Every program has to have one, and only one, such function. In this program the function `main` starts at line 04. The keyword `int` indicates that the type of the data returned by this function is *integer*, that is, a whole number. Some functions do not return data, and then one writes instead `void`.

[†]Sometimes the code is not read by a computer at all. John Archibald Wheeler, the physicist, is said to have kept a cardboard box on his desk with `COMPUTER` written on it. Whenever he found something difficult to understand, he would write it as a computer program. Often that was all he needed, and then he put the program in the box and forgot about it.

Braces The structure of the function `main` is highlighted in the following lines:

```
int main()
{                   // brace opening the body of "main"
...
}                   // closing brace
```

Another commonly used layout saves a line by writing instead:

```
int main() {
...
}
```

The matching braces enclose the *body* of the function, which is the part of the program that is executed when the function is called.

Line 05 Lines 05 through 14 contain *statements*. They constitute the *body* of the function `main`. Before considering line `05` as a whole, let us look at the statement

```
printf("Good morning, everyone!");
```

The form of the statement is that of a *function call*: it starts with the name of the function, `printf`, followed by one or more expressions between parentheses, followed by a semicolon. The name `printf` is short for "print formatted". The "formatted" refers to facilities to control the format of numerals and their placement on the page.

Between the parentheses is the *actual parameter* of the call to `printf`. It is a *string*, which is the text enclosed by double quotes. The result of executing the statement on this line is that the characters of the string, minus the enclosing quotes, appear in more or less the same form on whatever output medium is in force, usually a screen or paper.

Consider now the statement

```
printf("Good morning, everyone!\n");
```

The last two characters `\n` of the string are not output in the form in which they occur in the string. Together they are an end-of-line marker. They ensure that a new line is started for any output that may follow. The backslash `\` is an *escape character*. It indicates that something special is going on: it and the following character jointly stand for some output effect other than printing a character.

Line 06 The call to `printf` on this line has two actual parameters. The name of a function specifies a job to be done, in this case to print something. The actual parameters in a function call give needed information about this job, in this case what to print. Multiple actual parameters are separated by commas.

In `printf("This is a number: %f\n", 3.1415926535);` the `%` symbol in first actual parameters has in common with the character `\` that it is a special character

that does not appear as such in the printed result, but serves to modify the following character to produce a special effect. In the case of `%f` the special effect is to substitute the second actual parameter in its place and to print it in a certain format. The `f` specifies that format; it is the one that is appropriate for fractional numbers. The overall effect of the statement is to print:

```
3.141593
```

As a default, the number has been rounded to seven significant digits. How to effect rounding to fewer or more decimals will be explained in Section C.2.

Line 07 The effect of executing this line is the output

```
4
```

This example has been included to show that the printed string need not contain anything more than the item to be substituted. Also, the expression in the second actual parameter is "evaluated", so that you see the *value* 4 rather than the *expression* `2+2` that has this value.

The item to be substituted in the string in the first actual parameter is `%d`, where the `d` indicates that the format to be used is one suitable for integers (as decimal numeral; other choices are `%x` for hexadecimal and `%o` for octal, to be introduced later).

Line 08 This line has been included on purpose to have some similarity to the line above. However, now the second actual parameter is not the expression `2+2`, but it is the *string* `"2+2"`, which is the sequence of the three characters between quotes. It is this sequence of three characters that is substituted for the `%s` in the first actual parameter.

Line 09 This line introduces a *variable*. It has a *name*, which is `x`. It has a *type*, which is `double`. Every variable has a *value*. The value of `x` is restricted to be a fractional number, and this is indicated by the specified type [‡].

Line 10 As every variable has a value, let us find out what is the value of `x` by printing it. As `x` has been created without specifying a value, the implementer of the C language is not responsible for giving the variable any particular value. It may differ from system to system and may, on some systems, even differ every time the program is run.

Line 11 The variable `x` gets a value by its presence in the left-hand side of an *assignment statement*. On the right-hand side is an expression giving the value that the variable is getting as a result of executing this statement.

Line 12 We check the value of `x` again.

[‡] `double` comes from 'double-length floating-point".

Line 13 `atan` is a function provided by the mathematics part of the standard library. It is short for "arc tangent".

Line 14 The function has done all that we want it to do. It is also supposed to return a value. In the case of `main` the value should indicate whether any exceptional conditions arose during its execution. If so, then the value returned is 1; otherwise 0. The C implementation used here allows this statement to be omitted in the latter case, which we will often do from now on.

Line 15 This line contains the brace that closes the body of function `main`. It ends the function definition and the program.

To summarize we give the output of Program 2.1 below.

```
Good morning, everyone!
This is a number: 3.141593
4
2+2
0.000000
1.000000
3.141593; this should be pi
```

2.3 Evaluation of formulas

One of the earliest computer applications was to perform numerical calculations. In fact, up to the middle of the twentieth century "computer" meant a person doing numerical calculations with pencil and paper supplemented by a slide rule, books of tables, or a mechanical calculator.

The earliest computer programs evaluated mathematical formulas in an indirect way, as a sequence of instructions for transfer between memory and processor registers or to perform an operation on specified processor registers. This way a formula that can be understood at a glance on a printed page was transformed into a sizeable chunk of code that can be deciphered, but not really *read*.

The earliest programming languages were motivated by the desire to program as large a part of a numerical computation as possible directly by means of a formula written as much as possible as it is printed in a book or as it is written on a blackboard. The programming language C has this facility as well. In fact, in line 13 of Program 2.1 you may have noticed `4*atan(x)` as the counterpart of the conventional algebraic notation

$$4 \arctan(x).$$

The rules for translating conventional algebraic expressions to C are complex in detail. But most programmers don't refer to the detail: they get by with the general idea behind the translation, which I will explain here with an example. The example will be used in the next program, which also demonstrates how a program can perform input.

Newton's law Let us compute the force due to gravity, according to Newton's law, between two spherical bodies with homogeneously distributed masses m_1 and m_2 with centres at distance r. The law can be expressed by the formula

$$G\frac{m_1 m_2}{r^2} \tag{2.1}$$

The program prompts the user for m_1, m_2, and r, then inputs these values. The program outputs the result of evaluating the formula.

Progress has been made towards the ideal of having computers read traditional mathematical notation. But it is difficult to get a computer to process traditional notation in its full complexity. One difficulty in a formula like (2.1) is the fact that multiplication is implicit rather than explicit; it is implied by two variables being written next to each other. Another difficulty arises from the two-dimensional layout of the formula. For convenient processing by the compiler, it is important that a program consists of linear character sequences, so it's not a good idea to have a horizontal bar separating the dividend above from the divisor below. For the same reason, in computer programs one does not square a number by an implied operation with the exponent written a bit higher like in r^2.

But even staying within conventional notation, we can avoid the features that are awkward for compilers. We can place the symbols in a linear sequence and write all operations explicitly:

$$G \times (m_1 \times m_2)/(r \times r).$$

This translates in a straightforward fashion to an expression

```
G*(m1*m2)/(r*r)
```

that is acceptable to the C compiler. This is what you find in line 11 of Program 2.2.

```
00 #include <stdio.h>
01
02 int main() {
03   const double G = 6.673e-11;
04   printf("Input the first mass in kilograms:\n");
05   double m1; scanf("%lf", &m1);
06   printf("Input the other mass in kilograms:\n");
07   double m2; scanf("%lf", &m2);
08   printf("Input distance in metres:\n");
09   double r; scanf("%lf", &r);
10   printf("The force of gravitational attraction is ");
11   printf("%lf newtons.\n", G*(m1*m2)/(r*r));
12 }
```

Figure 2.2: A program to compute values of Newton's formula for the force due to gravitation.

For G, the constant of gravity, we use a *constant* rather than a variable. Hence line 03

```
const double G = 6.673e-11;
```

A constant is like a variable, except that it cannot occur in the left-hand side of an assignment statement. The only way to give it a value is by including in its definition an *initialization*, as was done here.

Fractional numerical values are written as a decimal fraction, possibly scaled by a power of ten. As the value of the gravitational constant is 6.673×10^{-11}, we write `6.673e-11`[§] in the program. We could have written `0.6673e-10` or `0.00000000006673` among many other equivalent possibilities.

Caveat This is not the recommended way to do small computation jobs! The handiest way to evaluate a mathematical formula is to use a pocket calculator, cell phone, or equivalent. These can handle formulas that are more complex than

$$G\frac{m_1 m_2}{r^2}$$

When calculator facilities run out, a spreadsheet program takes over. It is only for custom computations that require branches and iterations that one turns to a full-fledged programming language such as FORTRAN, MATLAB, or C.

2.4 Input

We saw how a program can communicate the result of its computation to a user. This aspect of its operation is called "output". When a program executes a statement to receive data for its computation, it is said to perform "input".

Suppose we want to use Program 2.2 to get an idea of the force of gravitational attraction on the scale of objects that are big enough to make the force measurable, yet small enough to be handled in a laboratory. For example, let the two objects be leaden spheres with a mass of 400 kilograms each. These are small enough to be placed with their centres at a distance of 0.5 metres.

Consider the following code snippet:

```
double x = 1.0;
printf("%f\n", x);
```

The variable `x` gets its value as a result of executing the initialization in the first line. If we want to obtain that value by means of input from the user, then we can write instead:

```
double x;
scanf("%f\n", &x);
```

[§]The `e` comes from "exponent".

The library function `scanf` is the input counterpart of `printf`. The call to `scanf` causes execution of the program to pause. If the user then enters an acceptable representation of a `double` value, then execution resumes with `x` having that value. Note the similarity between the lines

```
printf("%f\n", x);
scanf("%f\n", &x);
```

but also the difference. The call to `printf` does not change the value of `x`; the call to `scanf` does. To ensure that execution of `scanf` can change this value, the ampersand `&` is needed. In a later chapter the meaning of this symbol is explained.

With this explanation of `scanf` in place, you will see that a possible combination of input and output resulting from execution of Program 2.2 is:

```
Input the first mass in kilograms:
  400
Input the other mass in kilograms:
  400
Input distance in metres:
  0.5
The force of gravitational attraction is 0.000043 newtons.
```

Here the indented lines are input by the user who is running the program. You will recognize the other lines as output from the program.

2.5 Exercises

2.5.1 Outputting quotes

In Program 2.1 we used the simplest kind of string. It can be defined by the rule:

> Write a string as any sequence of letters, digits, or spaces enclosed by, but not including, the double quote (").

But suppose we want to output something like

```
He cried: "Stop!"
```

We can't do that with this simple rule, which only works for strings that don't include any quotes. This problem is addressed by using an *escape character*: if we want to include a quote in a string between quotes, then we write a backslash (\) in front of it to ensure it is interpreted as part of the string. Thus the above output can be obtained by

```
printf("He cried: \"Stop!\"\n");
```

This exercise is to write a program that produces the following output verbatim character by character and distributed over the two lines as shown:

```
She came in and closed the door quietly behind her.
He lowered his newspaper: "Hi honey, what's up?"
```

2.5.2 Other uses of the escape symbol

The escape character can be also used to convert into a string any text that extends over more than one line. This is done by including `\n`, which ensures that the rest of the string, if any, continues on the beginning of the next line. If there is no rest of the string, then future output, if any, continues on the next line. This future output may itself start with `\n`.

To output spaces until the next tab stop, insert `\t` into the string. To output `\`, insert `\\` into the string.

This exercise is to write a program that produces the following output verbatim character by character and distributed over the two lines as shown:

```
To advance output to the next tab stop, include \t.
To advance to a new line, include \n.
To output a backslash, include \\.
```

2.5.3 Checking the mathematics library

It's no surprise that mundane machines can do no better than to *approximate* transcendental objects like the number π. But even some mundane things like $1/3$ can only be printed with an error. So we can't expect the computations by the mathematics library to be exact.

Write print statements to get an impression of the accuracy of the mathematics library. Do this by printing `sqrt(0.5)*sqrt(0.5)`. Also for `tan(atan(0.5))` and `atan(tan(0.5))`. Similarly for `sin` with `asin`, `cos` with `acos`, and `exp` with `log`,

2.5.4 Your favourite formula

Modify Program 2.2 to compute *your* favourite formula. In case you don't have a favourite formula, consider Heron's formula for the area A of a triangle where the sides have lengths a, b, and c : $A = \sqrt{s(s-a)(s-b)(s-c)}$, where s, the semicircumference, is $(a+b+c)/2$.

Chapter 3

A quick tour

In this chapter you *learn by example*. It presents small, self-contained programs that have been compiled and run as shown. I explain each of these programs just far enough that you understand what it does and what part is responsible for what. I can only keep the explanations short by keeping them superficial. You will see *what* they do, but not yet know *why*. By making changes to the examples you can already do interesting things.

In the chapters following the Quick Tour, I switch to the opposite strategy: I concentrate on specialized aspects of programming and treat each of them in depth.

3.1 State-oriented programming

Recall from Section 1.2 that processors work by changing the state of memory. Thus state, and the change of it, is basic to the operation of computers. There are several approaches to programming. They differ in their treatment of state. *Functional programming* and *logic programming* simplify many tasks by making state invisible. However, state cannot always be ignored. Functional and logic programming avoid applications where it is essential to take state into account. The approach to programming that gives state a central place can be called *state-oriented* programming. Widely used languages such as Python, C, C++, Java, and C# are state-oriented in the sense of making state the focus of computation.

A computer includes a processor that performs operations on data in memory under control of a program. One can tell that the processor has done something when it leaves the memory in a different state. One characterizes the program by the change of state it causes. As memory consists of a collection of variables, the state of the memory is determined by the content of each variable. As described in section 1.2, this suggests a form of state-oriented programming, which proceeds in the following successive stages:

1. **Configure:** create the state with a suitable collection of variables,

2. **Initialize:** cause the input data to become the contents of designated variables (this way the input data become available for computation; in simple cases this stage is merged with **Configure**),

3. **Compute:** change the state in one or more steps so that designated variables contain the desired result, and

4. **Output:** cause the contents of these variables to be printed.

The first stage is described in terms of *definitions*; the other stages in terms of *statements*. The definitions configure the state by creating a suitable collection of variables. Execution of a statement causes change of state, or causes output to happen.

3.2 Assignment

The first example follows the Configure, Initialize, Compute, Output pattern. The Compute stage consists of assignment statements. I discuss the four stages in order as they occur in Program 3.1.

```
0 #include<stdio.h>
1
2 int main() {
3   int x,y,z;
4   printf("Enter numbers x and y.\n");
5   scanf("%d %d", &x, &y);
6   z = x; x = y; y = z;
7   printf("x: %d; y: %d; z: %d\n", x, y, z);
8 }
```

Interaction (input indented):

```
Enter numbers x and y.
   1 2
x: 2; y: 1; z: 1
```

Figure 3.1: *Swap*, interchange the values of two variables.

Configure In the line 3 the state is configured by creating the necessary variables. The keyword `int` means two things: (1) variables are created and (2) the variables are suitable for storing whole numbers (the keyword `int` is short for "integer"). We say that the *names* of the variables are `x`, `y`, and `z`, respectively, and that their *type* is `int`. Creating a variable means that an area of memory is allocated of a size suitable for data of type `int`. The semicolon "`;`" terminates the definition.

The line could have been written as three separate definitions:

```
int x; int y; int z;
```

The shorthand shown is the usual version.

Initialize In line 4, variables `x` and `y` are initialized from input. The role of `z` will become clear as we proceed.

Compute By "compute" we understand change of state. This does not always involve "computation" in the usual sense. The state changes in lines 5–7 are designed to interchange the values of `x` and `y`, an operation that is sometimes needed as part of a computation.

At some stage in the interchange, we need the assignment statement

```
x = y;
```

Let us dissect this statement. On the left of the assignment symbol `=` is a variable. On the right is an expression; in this case the expression consists of the single variable `y`. The effect of the statement is to ensure that the variable on the left-hand side obtains as value the value of the expression on the right-hand side.

Let us now see how we could use assignment statements to exchange the values of two variables. A variable always has one value: neither more, nor fewer. This fact has important consequences. That there cannot be fewer than one value means that a variable always has a value, even when none has been assigned to it. That a variable never has more than one value implies that after `x = y;` the variable `x` has the value of `y`. The previous value of `x` has gone forever, unless this value has been saved into another variable. Exchange of the values cannot happen by somehow magically executing simultaneously `x = y;` and `y = x;`. The role of the variable `z` is to save the original value of `x` in time so that it can be assigned to `y` after the original value of `y` has been assigned to `x`.

Output In line 8, `x`, `y`, and `z` are printed. Note that `z` contains the original value of `x`.

3.3 Sequencing

In the remainder of the Quick Tour, we visit the various *control structures* that are available in C. By "control" one means the mechanisms that determine the order in which statements are executed.

The control structures can be grouped under the following headings: *sequencing*, *selection*, *iteration*, and *function call*.

Sequencing merely means that statements in the body of a function are executed in the same order as we read a text in English: within a line from left to right, while lines are processed from above to below. This is of course of crucial importance in understanding the programs we have discussed so far. I assumed that the reader would take it for granted. Now it's official.

3.4 Selection

The examples so far consisted of *straight-line* code: every statement is executed once, and executed in the order as written. We now consider code in which the next statement to be executed is not always the next in the program text. Instead, execution may continue at a statement that is selected on the basis of the value of a *condition*. Statements that allow us to do that are *selection statements*. These include *if-statements* and *if-else-statements*.

Let us first consider if-statements. These are statements of the form

`if (`C`)` S

where C is a condition and S is a statement, including its terminating semicolon. Such an if-statement executes S or it leaves the state unchanged, depending on the condition C.

A *condition* is an expression that has as value an integer. If C evaluates to any value other than 0 (typically 1), then S is executed. If C evaluates to 0, then the if-statement leaves the state unchanged.

Another type of selection statement is the if-else statement.

`if (`C`)` S_0 `else` S_1

where C is a condition and S_0 and S_1 are statements, including their terminating semicolons. If C evaluates to a non-zero value, then the if-else statement executes S_0 but not S_1. If the condition evaluates to zero, then it is the other way around.

3.4.1 Comparing two numbers

As a first example, let us consider how to write a program that reads two integers, compares them, and prints the result of the comparison.

Comparing two integers can yield three outcomes: the first is greater, the second is greater, or the two are equal. This suggests three if-statements, one for each of these contingencies. As the three conditions cover all possibilities and are mutually exclusive, one and only one of the three output statements is executed.

The conditions in the if-statements compare the integers with *comparison operators*. These include `>` with the meaning "greater than". The other comparison operators are `>=` (greater or equal), `==` (equal), `!=` (not equal), `<` (less), and `<=` (less or equal).

See Program 3.2, *Compare*.

3.4.2 Sorting three numbers

It is often required as part of a computation that a sequence of items needs to be rearranged so that it becomes sorted in increasing or decreasing order. While a typical sorting program works for sequences of any length, it is sometimes useful to have a program fragment that sorts a short sequence of fixed length. This suggests

```
00 #include <stdio.h>
01
02 int main() {
03   printf("Input two numbers\n");
04   int a, b;
05   scanf("%d %d", &a, &b);
06   if (a < b)
07     printf("first smaller\n");
08   if (a > b)
09     printf("first larger\n");
10   if (a == b)
11     printf("equal\n");
12 }
```

Sample interaction (input indented):

```
Input two numbers
   1 1
equal
```

Figure 3.2: *Compare*, use the if-statement to compare two numbers.

writing a program that prompts the user to input three integers and that prints them in non-decreasing order. A useful building block for such a program is "two-sort": code that ensures that two variables are in sorted order, interchanging them if necessary. By performing two-sort on suitably selected pairs of variables, any number of variables can be sorted.

To nail down these thoughts, it is useful to write them down in a stylized kind of English that describes the steps to be followed more explicitly than would happen in a description that conforms to the conventional constraints of grammar and style. We call this stylized English *pseudo-code*. An algorithm in pseudo-code consists of instructions for the computer possibly interspersed with text in square brackets containing comments for the human reader.

> place the three input numbers in the three variables a, b, and c
> two-sort a and b
> two-sort b and c
> [c is the greatest of the three]
> two-sort a and b
> [a, b, and c are sorted]

See Program 3.3, *Three-sort*.

3.5 Functions

An important aspect of writing a program is to decide what is to be done, and when it is to be done. In straight-line code, execution of statements is from left to right on the same line of program text and from top to bottom beyond a single line. We

```
00 #include <stdio.h>
01
02 int main() {
03   printf("Input three integers\n");
04   int a, b, c; scanf("%d %d %d", &a, &b, &c);
05
06   int x;
07   // two-sort a and b
08   if (a > b) { x = a; a = b; b = x; }
09   // two-sort b and c
10   if (b > c) { x = b; b = c; c = x; }
11   // two-sort a and b
12   if (a > b) { x = a; a = b; b = x; }
13
14   printf("The sorted input: %d  %d  %d\n", a, b, c);
15 }
```

Figure 3.3: *Three-sort*, sort three items with three two-sorts.

have seen structures for selection as departures from this order. The next step is the possibility of *packaging*: to allow the programmer to name a piece of code and to allow statements to use that name to execute that code independently of what part of the program has just been executed.

Such a piece of code is called a *function*; the statement causing its execution is a *function call*. Function calls themselves can be scheduled by selection and iteration, and the code of a function can be organized by selection, iteration, and can itself contain function calls.

Let's review what we have seen of functions, so far. In Program 2.1, line 13, we saw `atan(x)`. This is a call to a function named `atan` (short for "arc tangent") that takes an angle (i.e. a number) as actual parameter and computes the number that is the arc tangent of that angle. The computed number replaces the call in the expression `4*atan(x)`.

Program 2.1 contains several calls to the function `printf`. The purpose of the function is to produce an effect on the output medium. Neither `atan` and `printf` are defined in the program calling these functions. They are defined in a library that is made available to any program making these calls by including the line

```
#include <stdio.h>
```

in the case of `printf` and by

```
#include <math.h>
```

in the case of `atan`.

Let us look in Program 3.2 for code suitable for packaging as a function. The code can be divided into (a) a core that performs the comparisons and (b) peripheral statements for input and output. We make the separation formal by introducing a *function* to contain this core. What qualifies the core for this treatment is that it is a unit that achieves a succinctly definable goal. One can imagine that execution of this code is needed in several places in a program. Better than having to repeat the code is to package it once and for all and write a function call in place of the code.

To implement this idea, C allows a function to be *defined* in one place and to be *called* in any number of places in the program. The definition *names* the function and lists the code to be executed when the function is called. This code is the *body* of the function. As one typically wants the body to be executed with different values of selected variables, these variables are made into *formal parameters* of the function body. The function call typically contains *actual parameters*, which are the values to be substituted for the formal parameters before execution of the body. The value computed by the function call replaces the function call.

Let us look at Program 3.4, where the various features of function definition and function call can be seen. In line 14 and 16 you see two occurrences of the function call `intcmp(p,q)`. The name `intcmp` refers to the opening (line 02) of the function definition that extends from lines 02 through 09. The body of that function starts with the opening brace on line 02 and ends with the closing brace in line 08.

Lines 03, 04 and 05 are the specification of the function. It should be included in every definition of a function that is of more than ephemeral interest.

Line 08 contains a comment: any text following `/*` and before the next occurrence of `*/` is a comment in the sense of being ignored by the compiler. In this case the usual form, which is the text following `//` to the end of the line, is not convenient.

The precondition should give the information necessary to ensure that calls to the function execute without mishaps and achieve the stated purpose.

The body is intended to be executed with different values of `a` and `b`. These are the *formal parameters* of `intcmp`, a fact indicated by the *parameter list* between parentheses in the function opening in line 02. The keyword `return` causes termination of the execution of the body and it specifies the value "returned" by the function, which is the value to be substituted for the function call.

The function call `intcmp(p,q)` specifies not only the name of the function being called, but also has a *list of actual parameters* consisting of the values to be substituted for the formal parameters in the function definition.

The function `main` We have been using a function definition from the beginning; every program contains a function definition. In Program 2.1 this function definition extends over the lines 04 through 15. It has the form

```
int main() { ... return 0;}
```

The name of the function is `main`. The `int` preceding the name indicates that the function returns an integer result. The parentheses following the name enclose

```
00 #include <stdio.h>
01
02 int intcmp(int a, int b) {
03 // Purpose: return -1, 0, or 1 according to whether a is less
04 // than, equal to, or greater than b.
05 // Preconditions: None
06   if (a < b) return -1;
07   if (a > b) return 1;
08   /* a == b */ return 0;
09 }
10
11 int main() {
12   printf("Input two numbers\n");
13   int p, q; scanf("%d %d", &p, &q);
14   if (intcmp(p,q) < 0)
15     printf("first smaller\n");
16   else if (intcmp(p,q) > 0)
17     printf("first larger\n");
18   else printf("equal\n");
19 }
```

Figure 3.4: *Compare Function*, Program 3.2 reorganized by the use of a function.

the function's formal parameters. Even when there are none (as in this case) the parentheses have to be there. The braces begin and terminate the *body* of the function. Between the braces we find the code that is executed as a result of a call to the function. The statement `return 0;` specifies that 0 is to be returned as value in response to any call to `main`.

Function `main()` is exceptional in several ways:

1. In every program one function of this name has to be defined.

2. The function `main` is not called in code written by the programmer. The programmer can cause any piece of code to be executed by wrapping it up in a function definition, and then calling that function. For this to happen the program has to be in the process of execution. The programmer cannot cause this process to start; the operating system has to do it and it does it in the form of a call to the function designated as `main` by the programmer.

3. `main` can be defined with either two formal parameters or none. I mostly use the latter option.

4. If the end of the body is reached without encountering a `return` statement, then `return 0` is executed before leaving the body.

Functions serve purposes other than the one of breaking up large pieces of code. Functions make it easier to re-use code that is applicable in situations other than the one for which it was first conceived. Moreover, functions make it easier to reason about the effect of code.

3.6 Iteration

If restricted to sequencing and selection, the program resulting from a page of code would be finished executing in a matter of microseconds. To get anything useful done in a program of reasonable size, one typically uses at least one of the control structures for *iteration*, which is the repetition of the same piece of code, but with varying data. Some iteration statements have the form of a *while statement*:

`while (`C`)` S

where C, the *condition*, is an integer expression, and S, the *body*, is a statement.

If C is zero, execution of the *while* statement has no effect. Otherwise, its execution consists of repeatedly executing S as long as C is not zero. One can think of the *while* statement as having a goal, and that this goal is to make C evaluate to zero.

If neither the execution of S, nor the evaluation of C results in C ever becoming zero, then execution of the *while* statement continues forever.

3.6.1 Conversion of fuel consumption rates

Consider Program 3.5, *MPG*, that prints the following table that converts from miles per gallon to litres per 100 km for some values that cover many cars:

```
mpg   litres/100 km
15    15.68
20    11.76
25     9.41
30     7.84
35     6.72
```

See Program 3.5, *MPG*.

In the iteration in *MPG* the print statement is the one to be repeated. Between repetitions something changes in the repeated statement. Here it is the value of `mpg`, the *controlling variable*. The iteration uses `while` in a stereotyped pattern: initialize the controlling variable, test for termination, compute, and increment the controlling variable.

C recognizes the utility of this pattern by providing the *for* statement. With this control structure, the loop is written like this:

```
double mpg;
for (mpg = 15; mpg <= 40; mpg = mpg+5) {
  printf("%2d\t%5.2lf\n", mpg, conv/mpg);
}
```

```
00 #include <stdio.h>
01
02 int main() {
03   const double LPG = 3.785; // litres per U.S. gallon
04   const double KPM = 1.609; // km per mile
05   double conv // conversion factor
06               = 1 * // mile per gallon
07                 KPM * // km per gallon
08                 (1/LPG); // km per litre
09   conv = (1/conv) * // litre per km
10          100;       // litre per 100 km
11   printf("mpg\tl per 100 km\n\n");
12   double mpg = 15;
13   while (mpg <= 35) {
14     printf("%2.2lf\t%2.2lf\n", mpg, conv/mpg);
15     mpg = mpg + 5;
16   }
17 }
```

Figure 3.5: *MPG*, print a table to convert from miles per gallon to litres per 100 kilometres.

Often it is abbreviated to

```
for (double mpg = 15; mpg <= 40; mpg += 5)
  printf("%2d\t%5.2lf\n", mpg, conv/mpg);
```

when `mpg` is only needed in the body of the *for* statement. Because the statement to be repeated consists of a single statement, it makes no difference that the braces have been dropped. Finally, `mpg = mpg+5` is often shortened to `mpg += 5`.

The advantage of a *for* statement over a *while* is that the three items that characterize iterations—initialization, test for completion, and update of the controlling variable—have a fixed place. It is a common error to forget one of these.

3.6.2 Using integer divide and remainder

When an integer is divided by an integer, an integer results. Thus $3/2$ gives 1. Consequently, $(3/2) * 2$ does not get you the 3 back. But we do have that $(n/m) * m + n\%m = n$, where n and m are integers and $n\%m$ is the remainder when n is divided by m. See Program 3.6, *Div-Mod*, for integer divide and remainder in action.

Its output is

```
0 0 0 0 0 0 1 1 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3 3 4
0 1 2 3 4 5 0 1 2 3 4 5 0 1 2 3 4 5 0 1 2 3 4 5 0
```

When you read the 25 columns, you see the first 25 non-negative integers displayed as base-6 numerals.

```
0 #include <stdio.h>
1
2 int main() {
3   for (int n = 0; n < 25; n++) printf("%d ", n/6);
4   printf("\n");
5   for (int n = 0; n < 25; n++) printf("%d ", n%6);
6   printf("\n");
7 }
```

Figure 3.6: *Div-Mod*, compare results of integer divide and remainder.

3.6.3 Compound interest

What difference does it make when a bank converts an account from an interest rate of r percent per year to a daily interest of $r/365$ compounded daily? And, if that's not too expensive for the bank, why not exploit the spectacular marketing impact of compounding *by the second*?

Once upon a time such questions would be answered with a table of logarithms; nowadays with a spreadsheet. Either way, the various ways of answering them provide a useful demonstration of programming techniques. Here is a straightforward, though inefficient way.

The effect of applying an interest rate of r % to an amount a is to multiply a by $1 + r/100$. To compound the interest for n periods, one repeats n times the application of the interest rate of r/n %; that is, one mulitplies a by $(1+r/(100n))^n$. Apparently we need a function, say, *power*, that computes arbitrarily high integer powers.

See Program 3.7, *Extreme Compounding*.

A sample interaction with Program 3.7 is (input indented):

```
Enter amount:
   1000
Enter annual interest rate in percent:
   2
With interest over a year: 1020.000000
With daily compounding: 1020.200781
With compounding by the second: 1020.201339
```

3.7 Arrays

Suppose we are recording the output of a sensor at time instants that are one millisecond apart. Do we have to define thousands of variables to store these data?

```
00 #include <stdio.h>
01
02 double power(double b, int e) {
03 // Purpose: return b to the power e.
04 // Preconditions: b > 0 and e >= 0.
05   double p = 1.0;
06   for (int i = 0; i < e; i++) p *= b;
07   return p;
08 }
09 int main() {
10   printf("Enter amount:\n");
11   double amount; scanf("%lf", &amount);
12   printf("Enter annual interest rate in percent:\n");
13   double rate; scanf("%lf", &rate);
14   printf("With interest over a year: %lf\n",
15          amount*power((1.0 + rate/100), 1)
16         );
17   printf("With daily compounding: %lf\n",
18          amount*power((1.0 + rate/(100*365)), 365)
19         );
20   double numSec = 60*60*24*365; // number of seconds in a year
21   printf("With compounding by the second: %lf\n",
22          amount*power((1.0 + rate/(100*numSec)), numSec)
23         );
24 }
```

Figure 3.7: *Extreme Compounding*, compound interest at extreme frequencies.

Or suppose we want to verify that April 16 is day 106 of a non-leap year. To be able to do this for arbitrary dates, we need lengths of months. Do we have to define twelve `int` variables to store this information?

As these examples remind us, a computer application is often concerned with a sequence of data rather than with a single datum. To make it easier to work with data that are structured as a sequence, C has *arrays*. For every kind of number, there exists an array of elements all of that kind of number. For example, when we define

```
int a[10];
```

the result is an array `a` of ten integer elements. The fact that `a` is not an integer, but an array of integers is specified by the brackets. Each of the elements of the array can be treated as an integer variable. They are accessed as `a[0]`, `a[1]`, ..., `a[9]`, ten elements in all. The integers between square brackets are the *indexes* of the elements.

It is a property of C that, wherever an integer is allowed, we can have an integer expression. For example, if `x` has value 3 and `y` has value 7, then `a[y-x]` is allowed and is the same array element as `a[4]`. If `i`, `a[i]`, and `b[i]` are in the index ranges of `a` and `b`, then `a[b[i]]` and `b[a[i]]` are defined. See Program 3.8, *Array Indexing*.

```
00 #include <stdio.h>
01
02 int main() {
03   int a[] = {2,3,4,1,6,5,0,8,9,7};
04   printf("%d ", a[9]);
05   printf("%d ", a[a[9]]);
06   printf("%d ", a[a[a[9]]]);
07   printf("%d ", a[a[a[a[9]]]]);
08
09   int b[] = {6,3,0,1,2,5,4,9,7,8};
10   for (int i = 0; i < 10; i++) printf("%d ", a[b[i]]);
11   printf("\n");
12   for (int i = 0; i < 10; i++) printf("%d ", b[a[i]]);
13   printf("\n");
14 }
```

Output:

```
7 8 9 7 8
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
```

Figure 3.8: *Array Indexing*. Compare lines 10 and 12 and compare their outputs.

3.7.1 A common calculation

Since ancient Egypt and Mesopotamia the most common computation has been the addition of a list of numbers. It is not too late to honour this tradition. This is the right place because the array is a natural way to store a list of numbers. See Program 3.9, *Array Sum*. Its output is:

```
Sum of month lengths: 365
```

```
00 #include <stdio.h>
01
02 int sum(int a[], int n) {
03   int s = 0;
04   for (int i = 0; i < n; i++) s = s+a[i];
05   return s;
06 }
07 int main() {
08   int monLen[] = {31,28,31,30,31,30,31,31,30,31,30,31};
09   printf("Sum of month lengths: %d\n", sum(monLen, 12));
10 }
```

Figure 3.9: *Array Sum*, compute the sum of the elements of an array.

3.8 Exercises

3.8.1 Effect of multiple assignments

Before executing the code fragment

```
x = x+y; y = x-y; x = x-y;
```

the values of `x`, `y`, and `z` are x_0, y_0, z_0, respectively. Express the values of these variables after execution in terms of the initial values.

3.8.2 Four-sort

Write a program that reads four numbers and prints them in sorted order. Consider using two-sorts.

3.8.3 Rank in sorted series

Write a program that reads four numbers n_0, n_1, n_2 and n_3 that are in increasing order. It then reads an arbitrary number, say x. The program prints how many of n_0, n_1, n_2 and n_3 are smaller than x.

3.8.4 Leibniz's formula

A formula invented by Leibniz is:

$$\frac{\pi}{8} = \sum_{n=0}^{\infty} \frac{1}{(4n+1)(4n+3)}. \tag{3.1}$$

Use the first 1000 terms of the infinite sum to obtain an approximation of π.

3.8.5 Wallis's formula

A formula invented by Wallis is:

$$\frac{\pi}{2} = \frac{2 \times 2 \times 4 \times 4 \times 6 \times 6 \times \ldots}{1 \times 3 \times 3 \times 5 \times 5 \times 7 \times \ldots} \tag{3.2}$$

Use the first 1000 factors of the infinite product to obtain an approximation of π.

3.8.6 Area of triangle

Write and use the function:

```
double area(double a, double b, double c) {
//Purpose: return the area of the triangle with sides of lengths
//  a, b, c if such a triangle exists; return -1 otherwise.
//Preconditions: a, b, and c nonnegative.
// Your code here.
}
```

See Exercise 2.5.4.

3.8.7 GCD and LCM

Write and use the functions:

```
int gcd(int x, int y) {
//Purpose: return the greatest common divisor of x and y.
//Preconditions: x and y nonnegative.
//Your code here.
}
int lcm(int x, int y) {
//Purpose: return the least common multiple of x and y.
//Preconditions: x and y nonnegative such that the
//product of x and y is representable as int.
//Your code here.
}
```

3.8.8 Fibonacci numbers

The Fibonacci numbers $f_0, f_1, f_2, f_3, \ldots$ are defined as $f_0 = 0$, $f_1 = 1$, and $f_{i+2} = f_{i+1} + f_i$ for $i = 0, 1, 2, \ldots$. Write and use the function:

```
int fibonacci(int n) {
//Purpose: return the n-th Fibonacci number.
//Preconditions: 0 <= n < N where N is such that the N-th Fibonacci
//number is the smallest that is not representable as int.
//Your code here.
}
```

3.8.9 Polynomial evaluation

Input: a fractional number x, an array a of fractional numbers, and an integer n, the length of a.
Output: a fractional number equal to

$$a_0 + x(a_1 + x(a_2 + \ldots + x(a_{n-2} + xa_{n-1}) \ldots).$$

3.8.10 Continued fraction

A continued fraction is an expression that is determined by integers

$$a_0, a_1, a_2, \ldots, a_{n-1},$$

called *convergents*. For $n = 5$, the expression is:

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{a_4}}}}$$

Write and use a function:

```
double contFrac(double a[], int n) {
//Purpose: return the value of the continued fraction
//with convergents in a[0..n-1].
//Preconditions: n positive.
//Your code here.
}
```

An interesting case has a equal to $3, 7, 15, 1, 292$ and $n = 5$.

3.8.11 Collatz sequence

The Collatz sequence is defined for any initial positive integer n by the following algorithm:

read n, a positive integer
while $n \neq 1$
    if n is odd then $n \leftarrow 3n + 1$
    else $n \leftarrow n/2$

The curious phenomenon is that the sequence seems to terminate for every initial n.

For some starting values of n we get quite short sequences. An example is 48, as one can easily check. But starting with 47 gives quite a cliff-hanger:

```
47  142  71  214  107  322  161  484  242  121  364  182
91  274  137  412  206  103  310  155  466  233  700  350
175  526  263  790  395  1186  593  1780  890  445  1336
668  334  167  502  251  754  377  1132  566  283  850
425  1276  638  319  958  479  1438  719  2158  1079
3238  1619  4858  2429  7288  3644  1822  911  2734  1367
4102  2051  6154  3077  9232  4616  2308  1154  577  1732
866  433  1300  650  325  976  488  244  122  61  184  92
46  23  70  35  106  53  160  80  40  20  10  5  16  8  4
2  1
```

Write a program that prints the length and the maximum of the Collatz sequence starting at i, for integers i from p up to and including q.

3.8.12 Odometer

An odometer consists of a row of counters, each of which counts from 0 to 9. From the right to the left, the counters count units, tens, hundreds, and so on. Every time a unit (a mile or a kilometer) is counted, the units counter is incremented. However, if this counter already shows 9, then it turns back to 0, and the counter to the left is incremented. Again, if this is already at 9, then the same story repeats with *its* left-hand neighbour, as long as there is one.

Write a program that simulates a four-digit odometer by displaying (not on paper) on successive lines the reading after each change.

3.8.13 Seven Questions

Write a program that implements the guessing game illustrated below. Inputs are indented.

```
Choose a positive integer X less than 100.
If you truthfully answer my questions,
then I will you tell what X is.

Is X less than  50  ?
If so, then enter 1; otherwise 0.
   1
Is X less than  25  ?
If so, then enter 1; otherwise 0.
   0
Is X less than  37  ?
If so, then enter 1; otherwise 0.
   0
Is X less than  43  ?
If so, then enter 1; otherwise 0.
   1
Is X less than  40  ?
If so, then enter 1; otherwise 0.
   0
Is X less than  41  ?
If so, then enter 1; otherwise 0.
   0
Is X less than  42  ?
If so, then enter 1; otherwise 0.
   1
X is 41
```

The program should not ask more than seven questions to determine the value, whatever `X`. How many questions for identifying any integer up to a thousand? Up to a million?

3.8.14 Finite differences

Before computers an important aid to computation consisted of books containing tables of various commonly used functions. The compilers of such volumes invented several tricks for decreasing the amount of computation needed. In this section we consider the table of third powers of positive integers: the sequence $1, 8, 27, 64, 125, \ldots$

If asked to compute 6^3 out of context, one would need to perform two multiplications. But if one knows enough of the preceding third powers, then multiplications can be avoided. The key is the technique of *finite differences*. Consider the tableau

```
0  1   8   27    64    125    cubes
 1   7   19   37    61     first differences
    6  12   18   24     second differences
      6   6    6      third differences are constant
```

The differences are the differences between successive entries in the line above. Differences between the function values are "first" differences. Differences between first differences are second differences, and so on. It is easily seen with a bit of calculus that the n-th differences for n-th powers of integers are constant and equal to $n!$.

The table makers' trick is the following. To get the next cube, one starts by observing that the next entry in the bottom line is 6. That gives 30 for the next entry on the line above and so on until one reaches the top line with $6+24+61+125$ equals $216 = 6^3$.

This exercise is to modify the following program to avoid multiplications.

```
#include <stdio.h>

void table(int n) {
// Purpose: print cubes of 0..n.
// Precondition: n cubed at most largest representable integer.
  for (int i = 0; i <= n; i++) printf("%d\n", i*i*i);
}
int main() {
  printf("Input a positive integer.\n");
  int n; scanf("%d", &n); table(n);
}
```

3.8.15 Point-to-point distance

Write and use the function:

```
double dist(double co_ord[]) {
//Purpose: return the distance between two points whose
//x and y coordinates are in co_ord[0..3].
//Preconditions: co_ord has size 4.
//Your code here.
}
```

3.8.16 Area of triangle

Write and use a function:

```
double area(double co_ord[]) {
//Purpose: return the area of a triangle whose vertices have
//x and y coordinates given in co_ord[0..5].
//Preconditions: co_ord has size 6.
//Your code here.
}
```

See 3.8.15 and 3.8.6.

3.8.17 Print Sudoku puzzle

Program 3.10 stores a Sudoku puzzle in an array. Add the definition of function `print` so that the puzzle in printed as shown.

```
00 #include <stdio.h>
01
02 void print(int s[], int n);
03
04 int main() {
05   const int n = 3; // blocksize
06   int s[] = {
07     0,0,0, 7,0,0, 2,1,0,
08     0,0,0, 0,5,9, 0,4,3,
09     0,0,0, 0,0,8, 9,0,0,
10
11     8,0,2, 0,0,0, 0,0,0,
12     6,5,0, 0,1,0, 0,2,4,
13     0,0,0, 0,0,0, 5,0,7,
14
15     0,0,7, 2,0,0, 0,0,0,
16     9,1,0, 5,8,0, 0,0,0,
17     0,8,4, 0,0,6, 0,0,0
18               }; // size 81
19   print(s,n);
20 }
```

Desired output:

```
| 0 0 0| 7 0 0| 2 1 0|
| 0 0 0| 0 5 9| 0 4 3|
| 0 0 0| 0 0 8| 9 0 0|

| 8 0 2| 0 0 0| 0 0 0|
| 6 5 0| 0 1 0| 0 2 4|
| 0 0 0| 0 0 0| 5 0 7|

| 0 0 7| 2 0 0| 0 0 0|
| 9 1 0| 5 8 0| 0 0 0|
| 0 8 4| 0 0 6| 0 0 0|
```

Figure 3.10: Exercise: add function definition so that the output is as shown.

3.8.18 Sort an array

Write and use a function:

```
void sort(double x[], int n) {
// Purpose: sort x[0..n-1]
// Preconditions: n >= 0 and x[0..n-1] is allocated.
```

Exclusive use of two-sorts is acceptable.

3.8.19 Find median of sample

The median of a sample $x_0, \ldots, x_{n-1}$ is defined as the middle element of the sorted version if there is a single middle element and the mean of the two middle elements otherwise. Write and use a function:

```
double median(double x[], int n) {
// Purpose: to return the median of x[0..n-1]
// Preconditions: n >= 0 and x[0..n-1] allocated.
// Your code here.
}
```

Part II

Base

Chapter 4

Fundamental data types

Computers do their work by operating on data. Much of a programmer's work consists in designing operations and data types that are just right for the application. These programmer-designed operations and data types are directly or indirectly defined in terms of the fundamental ones that are built into the programming language. In this chapter we survey the most commonly used data types that are built into C.

4.1 Data in a computer

In their first decade, computers were only used for numerical applications. These data were the fractional numbers that occur in scientific and engineering applications and the integers that dominate financial and administrative applications. Nowadays computers are often used as media machines. Much of their memories is allocated to storage for images, sounds, and text. Yet, when one looks closely, the data in memory still consists of numbers. The fundamental data types are either manifestly numeric or are numbers in disguise.

If we want to speak of *data* in a computer other than in a general way, then we need to speak of the *values* that variables of a programming language can have. Each item of information processed by a computer (including images and sounds) is processed as the value of some variable of the programming language used to write the code.

4.2 Types

The values that variables can assume are classified into *types*. There are several reasons for this. In the first place, the mathematical model of the world that the program manipulates has variables whose types are different in a mathematical sense. Some variables represent continuously changing quantities. Other variables count discrete items. A variable can also represent a character of text, a colour, the

Y-coordinate of a pixel, or a truth value of logic. Some programming errors can be traced to violations of the world model due to assigning a value of the wrong type to a variable, or due to calling a function with an actual parameter of a wrong type. When the programming language distinguishes types in a sufficiently rigorous way, the compiler can detect such type errors. Such languages are said to be *strongly typed*. C sits towards the strongly-typed end in the spectrum of programming languages.

Another reason for classifying the values of variables into types is that some mathematically defined types have infinitely many values. Therefore they cannot all be represented in computer memory. Approximations or restrictions in range are necessary. Depending on the application, we may want to spend more or fewer computer resources on making the approximation accurate or on widening the range.

4.2.1 Integers

A non-negative integer i with $0 \leq i < 2^n$ can be represented as a bit vector that is the base-2 numeral $b_{n-1} \ldots b_1 b_0$ for i:

$$i = b_{n-1}2^{n-1} + \ldots + b_1 2^1 + b_0 2^0.$$

This is the representation of the type `unsigned int`. As a consequence of this representation, the range of the unsigned integer type represented as an n-bit vector has 0 as least and $2^n - 1$ as greatest value. All integers within this range are represented exactly.

If we want to use the n bits $b_{n-1} \ldots b_1 b_0$ to represent an `int`, then about half of these bit vectors have to be used for negative values. An obvious representation is called "signed magnitude", which consists of a bit for the sign with the remaining bits dedicated to a representation of the absolute value. After an initial period of experimentation, computer designers have settled on the following representation, called *two's complement*.

Let us first look at an example of the two's complement representation of values of type `unsigned int` and `int`. For ease of presentation, we show in Table 4.1 the imaginary case where only three bits are available for representing an integer; that is, $n = 3$.

	000	001	010	011	100	101	110	111
unsigned	0	1	2	3	4	5	6	7
signed	0	+1	+2	+3	−4	−3	−2	−1
signed magnitude	+0	+1	+2	+3	−0	−1	−2	−3

Table 4.1: Three-bit representations for integers.

In the general case of two's complement representation, an integer i such that

$0 \leq i < 2^{n-1}$ is represented by a 0 followed by $b_{n-2} \ldots b_1 b_0$, n bits in all, such that

$$i = b_{n-2}2^{n-2} + \ldots + b_1 2^1 + b_0 2^0.$$

An integer i such that $-2^{n-1} \leq i < 0$ is represented by a 1 followed by $b_{n-2} \ldots b_1 b_0$, n bits in all, such that

$$i = b_{n-2}2^{n-2} + \ldots + b_1 2^1 + b_0 2^0 - 2^{n-1}.$$

These two cases can be summarized by saying that an integer i such that $-2^{n-1} \leq i < 2^{n-1}$ is represented by $b_{n-1} \ldots b_1 b_0$, n bits in all, such that

$$i = b_{n-1}(-2^{n-1}) + b_{n-2}2^{n-2} + \ldots + b_1 2^1 + b_0 2^0.$$

As a consequence of this representation, the range of the `int` type, represented as an n-bit vector, has -2^{n-1} as least and $2^{n-1} - 1$ as greatest value. All integers within this range are represented exactly.

C has the integral types `short` and `long`, which allow the possibility of integers using, respectively, fewer and more bits than `int`. Thus we could have 16 bits for `short`, 32 bits for `int`, and 64 bits for `long`. These sizes are not mandated by C; they can vary between machines. But `short` cannot be longer than `int`, nor can `int` be longer than `long`. As is the case with `int`, `short` and `long` are signed. The unsigned variants are `unsigned short` and `unsigned long`.

Integers are used for different purposes. In a program one might use an integer variable to count, for example, the number of bytes in a file containing music. The maximum value of such a counter had better be large. If four bytes are allocated for the counter, then the maximum value for an unsigned integer is $2^{32} - 1$, which would be sufficient for up to a four gigabyte file. That is a lot. But using only three bytes would give a maximum value of $2^{24} - 1$, only enough for files up to 16 megabytes. So four bytes is a reasonable size for a general-purpose integer, though one can imagine that it is not large enough for some purposes.

On the other hand if we used four-byte integers to record the amplitudes in a music file, then we would waste a lot of space. Using only two bytes would shrink the file by a factor of two, yet allow 2^{16} different values for the amplitudes. This illustrates the need for integers of different sizes. However, unless there is lack of space, it is wise to use `int` as default.

The following table shows the range of values for the various types of integer on a representative system.

4.2.2 Characters

C includes the character types `char`, `signed char`, and `unsigned char`. All three are classified under the "integral types", such as `short` and `int`. For the programs in this book, we only need the character type `char`.

The type `char` occupies at least eight bits, that is, one byte, according to the definition of C. The ASCII character set, which is adequate for representing C source

type	minimum value	maximum value
unsigned short	0	65,535
unsigned	0	4,294,967,295
unsigned long	0	18,446,744,073,709,551,615
short	-32,768	32,767
int	-2,147,483,648	2,147,483,647
long int	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

Table 4.2: Range of values of integer types on a representative system.

code and English text, contains the codes 0 through 127. So it only needs seven bits. As a result a `char` value in C has, typically, one spare bit.

In some implementations of C, `char` variables contain codes of an "extended ASCII" character set. Such character sets are not ASCII standard, but are character sets of which the codes 0 through 127 coincide with the ASCII codes where the other codes are used for characters needed for languages other than English.

Unicode has been created for languages that need more than 256 codes. It has 2^{16} codes. To accommodate such languages, C has the type `wchar_t`. It is natural to assume that variables of type `wchar_t` contain Unicode values. This is usually the case in C programs, but it is not mandated by the standard.

4.2.3 Booleans

We saw that integers are represented in the most compact way possible: n bits are used to represent 2^n different values. The Boolean type has only two values, usually called "true" and "false". The most compact representation would have a single bit.

However, computers do not handle single bits efficiently. In memory, bits are grouped in bytes, which would suggest using a whole byte to represent a Boolean. The natural unit for the processor is the word, which usually consists of multiple bytes. Hence it is not uncommon to find a Boolean represented as four bytes because words are four bytes.

The Boolean type is denoted by the keyword `bool`; the truth values by the keywords `true` and `false`. In C, truth values are represented by an integral type, where 0 means *false* and any other value means *true*.

The Boolean type is one of the more recent additions to C. Strictly speaking, the keyword is `_Bool` and neither `true` nor `false` are keywords. However, the library `stdbool.h` defines `bool` as an alternative to `_Bool` defines `true` as 1, and `false` as 0. Thus the inclusion of the `bool` type into C has been rather half-hearted. To avoid the hassle of `#include stdbool.h` we use 0 for `false` and some nonzero integer for `true`. A nice compromise between convenience and good logic is to use, instead of a literal, a Boolean expression that is manifestly true or false, like `1 == 1` and `0 == 1`.

4.2.4 Floating-point numbers

In Section 2.3 we encountered fractional numbers. As examples we gave as alternative representations for the constant of gravity: `0.00000000006673`, `6.673e-11`, and `0.6673e-10`. We read the latter as 6.673×10^{-11} and 0.6673×10^{-10}, respectively. This is the scientific notation familiar outside of programming. We can place the decimal point wherever we want by making a suitable adjustment to the exponent. Because of this freedom in placing the decimal point, the notation is known in computing as *floating-point format*. Thus we speak of floating-point numbers as a data type.

The beauty of the floating-point format is that we can also use fractions to represent big numbers. This allows us to write any positive number, however small or large, as $w \times 10^E$ where E is an integer (positive or negative) and w is between one and ten. Such standardization is useful for representing numbers in computer memory.

But, as computer memory consists of bits, the most natural representation is in base 2 rather than 10. This suggests that we make use of the fact that any number, however small or large, can be represented as $w \times 2^E$ where E is an integer and w is between one and two.

To nail things down more specifically, floating-point format represents any nonzero number x as

$$x = s \times (1 + b_1 2^{-1} + \cdots + b_k 2^{-k}) \times 2^E$$

where s is 1 or -1 (so we can represent negative numbers), k is some fixed positive integer and E can be any integer between a negative and a positive bound.

How many bits does this representation require? The two possibilities for s can be represented in one bit, the *sign bit*. We need k bits for $b_1, \ldots, b_k$. These are called the *significand*. Finally, we need a fixed number of bits for E, the *exponent* of the floating-point format. Fixing k limits the *precision* with which a number can be represented. Fixing the number of bits for E limits the *range*: that is, how close x can come to 0 and how far away.

C does not mandate that floating-point numbers are represented in any particular way. Many processors implement the method outlined above. C provides a data type `float`. In addition, there is `double`, which is allocated at least as much space as `float`. Finally, there is `long double`, which is allocated at least as much space as `double`.

For `float`, many processors implement the single-length format of the IEEE standard for floating-point arithmetic. This implies a k equal to 23 and 8 bits allocated for E. The sign bit brings up the total number of bits to 32. The smallest and largest values that E can take are -126 and $+127$, respectively. This leaves two special values for the 8 bits reserved for E. These are used to ensure that a floating-point number can become 0. Other special values are provided for, including positive and negative infinity.

If a processor implements the IEEE standard for `float`, then it is likely to implement `double` according the IEEE standard's double-length format. This implies

k equal to 52 and 11 bits for E. With the sign bit this adds up to eight bytes. This format gives greater precision and a larger range. In many implementations of C, `long double` is the same as `double`.

Both `float` and `double` allow us to represent extremely small and extremely large numbers; the latter allowing more extreme extremes. How do we choose between these? Experience shows that even innocent-looking computations can suffer from lack of precision. Accordingly many experienced programmers abide by the rule to use `double` for computation and to reserve `float` for situations where large amounts of numerical data needs to be stored and storage capacity is a concern.

This explains that you often see `double` as a *de facto* default.

4.3 Representation of values in programs

How do variables get their values? When the value comes from input, the program only shows the variable, not the value. A variable can also get its value by copying it from another one, as can happen in an assignment statement such as `x = y;` and in an initializing definition of the form `int x = y;`. But we can also specify the value itself in the program text, as in `x = 2;`. The item on the right of the assignment symbol is a *literal*. In this section we explain how to write literals of various types.

4.3.1 Integer literals

For its integer literals, C follows to some extent the conventional notation for decimal numerals. For example, any sequence of decimal digits not starting with a zero has its conventional meaning. Some literals for integers do start with a zero. These are *octal* or *hexadecimal* numerals. The reason for these can be explained as follows.

Sometimes it is necessary to specify an integer as a bit vector rather than as a decimal literal. We obtain such a bit vector as the representation of the integer as a numeral to base two. However, writing an `int` as a numeral in base two requires one to write a sequence of 32 zeros and ones. Even if one is willing to put up with the inconvenience of writing such a long sequence, errors are likely to be introduced. It is therefore better to collect the binary digits of a bit vector in groups. Three and four are handy sizes for such groups.

We get the effect of grouping the bits in threes by writing a nonnegative integer n in base 8. It is a sequence of integers, each of which is an octal digit: one of the integers 0 through 7. As these integers only require a single decimal digit, one can use the first 8 decimal digits as octal digits. This is the *octal representation* of n, as shown in Table 4.3.

Similarly, if we write a nonnegative integer n in base 16, we get a sequence of digits that are the possible remainders on division by 16. This is the hexadecimal representation of n. The first 10 hexadecimal digits coincide with the decimal digits. For the remaining ones we use `a`, `b`, `c`, `d`, `e`, and `f` for the hexadecimal digits represented by the decimal numerals $10, \ldots, 15$, respectively. The upper-case versions `A`, `B`, `C`, `D`, `E`, and `F` are also allowed. See Table 4.3.

octal:	0	1	2	3	4	5	6	7
	000	001	010	011	100	101	110	111
	0	1	2	3	4	5	6	7
	0000	0001	0010	0011	0100	0101	0110	0111
hexadecimal:								
	8	9	a	b	c	d	e	f
	1000	1001	1010	1011	1100	1101	1110	1111

Table 4.3: The octal and hexadecimal digits and their binary representations.

The octal digits are a subset of the decimal ones, which are, in turn, a subset of the hexadecimal digits. As a consequence, one cannot tell from a sequence of octal digits what number it denotes: 123 could mean $1 \times 8^2 + 2 \times 8^1 + 3 \times 8^0$, or $1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$, or $1 \times 16^2 + 2 \times 16^1 + 3 \times 16^0$. The following rule for integer literals resolves this ambiguity.

- If the literal starts with a nonzero decimal digit, then it is a decimal numeral.
- If the literal starts with `0` followed by an octal digit, then it is an octal numeral.
- If the literal starts with `0x` or with `0X` then it is a hexadecimal numeral.

It follows that no literal can start with, for example, `08`. It follows that `0xabacadab` is an integer literal, and that `abacadab` is not.

If we want to know what type a variable is, then all we need to do is look at the definition. But what is the type of a literal? Of course we can tell the difference between integer and floating-point. But `123` is integer and is small enough to be `short`. Does that make the type of this literal `short`?

The type of an integer literal is `int`, unless it is too large to be of this type. The complete rule says that the type of an integer literal is the first of `int`, `unsigned`, and `long` into which it will fit.

When it concerns large integers one should not leave it to the C compiler to infer what the type is. For example, on an implementation with four-byte integers, the output of

```
int x = 0xffffffff; //2^32 - 1 ?
printf("%d\n", x);
```

is -1. This is because the values of integers range from -2^{31} to $2^{31} - 1$. As a result $2^{32} - 1$ lies outside the range, which makes the type of the literal `unsigned`. The assignment tries to get an `unsigned` into an `int`. Most implementations issue a warning to this effect, but produce the unintended output anyway.

Better to say explicitly what the type of `4294967295` is. It is for situations like this that C allows integer literals to be adorned with the suffix `U` or `L`. The lowercase versions are also allowed, and have the same meaning. The suffix `U` stands for "unsigned"; `L` stands for "long".

On an implementation with four-byte integers, unsigned integers range between 0 and $2^{32}-1$. A literal such as `4294967295U` says explicitly that its type is `unsigned`. As a result

```
unsigned x = 4294967295U; //2^32 - 1
printf("%u\n", x);
```

gives the correct output.

Similarly, the suffix `L` causes an integer literal to be of type `long int`. This is intended to allow some literals that would be outside of the range of `int` to fit inside the range of `long int`. However, on many implementations these types have the same size, rendering the `L` suffix nugatory.

4.3.2 Character literals

Variables of the type `char`, although only required to hold the 128 ASCII characters, correspond to memory locations consisting of 8 bits, giving 256 values. C only partially specifies the characters corresponding to these integer codes.

Literals of this type that correspond to printable characters are written as that character enclosed in single quotes. Thus, `'a'`, `'b'`, `'z'`, `'0'`, `'9'`, `'.'`, `';'`, `'\''`, `'\\'` are examples of character literals. Note the escape convention in the last two.

It is also possible the write a character literal as its ASCII code. This needs to be given in the form of three octal digits preceded by a backslash. Thus, suppose one knows that A (first letter of the alphabet, capitalized) has ASCII code 65. Then one first converts to octal, which gives 101, so that `'A'` is equivalent to `'\101'`.

For example the following program

```
#include <stdio.h>

int main() {
  printf("%c %c %c %c %c %c\n",
         '\101', '\132', '\141', '\172', '\060', '\071');
}
```

gives as output

```
A Z a z 0 9
```

4.3.3 Floating-point literals

Sometimes conventional notation for decimal fractions is acceptable unchanged as a floating-point literal, as in `0.1234`. But scientific notation, as in 1.234×10^{-1} or 1234×10^{-4}, clearly needs some adaptation. These correspond to the literals `1.234e-1`, and `1234e-4`, respectively.

In general, a floating-point literal is distinguished from an integer by the presence of a decimal point or an `e` (which may be capitalized). At least one digit before the decimal point is needed, but not after. So `1234.e-4` is acceptable as well.

The decimal point or the `e` may be absent, but not both. As we saw, `0.1234` is a floating-point literal. So is `1234e-4`. But `e-4` is not allowed. If the `e` is present, then it needs to be followed by an integer, if only by a 0. So if we want to be perverse, then we can write `0.1234e0`, or even `0.1234e+0`, or `0.1234e-0`, to be super-perverse.

4.4 Type conversions

Language designers are pulled by two opposing forces. On the one hand, variables should reflect the type distinctions that come with the application. In this way, the compiler can catch many programming errors. On the other hand, these same type distinctions are often unnecessary. Enforcing them in such situations can become unbearably pedantic. For example,

```
float x = 1;
```

is not strictly correct. The literal `1` is of type `int`, which is not type `float`. C allows this and performs an *automatic type conversion* of `int` to `float`.

Conversion from `int` to `float` is only correct up to a certain limit. See Program 4.1, where this limit is breached. A representative output is `-46`. The implementation that gave this output warned about a `float` being initialized by an `int`, whereas the damage was really done by the automatic conversion of the `int` to `float`.

```
0 #include <stdio.h>
1
2 int main() {
3   int i = 1234567890;
4   float x = i;
5   int j = x;
6   printf("%d\n", i - j);
7 }
```

Figure 4.1: Not all integers are floating-point numbers: the output may not be zero.

Thus we see that automatic type conversions are a mixed blessing. To avoid errors caused by such conversions, we need to understand the relations between types, how to force conversions between them, and what the effects of these conversions are.

4.4.1 Relations between types

A type is a set of values. The relations between types that are easiest to explain are those that can be described as relations between sets.

Subsets The most straightforward relations occur when one type is a subset (symbol "$\subseteq$") of another:

- `short` $\subseteq$ `int` $\subseteq$ `long`
- `unsigned short` $\subseteq$ `unsigned` $\subseteq$ `unsigned long`
- `float` $\subseteq$ `double` $\subseteq$ `long double`

Overlapping sets The next simplest relation is between the signed and unsigned versions of the integral types. The signed version of an integral type represented in n bits ranges from -2^{n-1} to $2^{n-1}-1$, whereas the unsigned version ranges from 0 to 2^n-1. That is, the positive signed integers coincide with the smaller half of the unsigned integers. These two types have a significant and useful overlap.

Another significant and useful overlap is between integers and floating-point types. Up to a certain bound, all integers are also floating-point numbers. As Program 4.1 shows, this does not hold for integers beyond this bound. Even though such large integers can only be approximated by floating-point numbers, the approximation is good enough for some purposes.

Characters and integers Conceptually the set of characters has nothing to do with the set of integers. However, for historical reasons, characters can be treated as integers in C.

It can be useful to perform arithmetic on characters. Suppose that the implementation uses the ASCII character set for `char` and that `char ch` contains an upper case character. Then `ch + 'a' - 'A'` is the integer code for the lower-case version of the character contained in `char ch`. This is because the upper-case letters are adjacent to each other in alphabetic order; the same holds for the lower-case letters. This is all we need to know about the ASCII codes for characters to be able to convert between upper and lower case according to the method shown.

Similarly,

```
for (int n = 1234567; n ; n /= 10) printf("%c", '0' + (n % 10));
```

prints the digits of that number, alas in reverse order.

Booleans and integers Conceptually the set of Booleans has nothing to do with the set of integers. However, for historical reasons, integers can be treated as Booleans in C. An integer is considered the same as `false` if it is zero, and as `true` otherwise.

For example,

```
sum = 0; while (n) { sum = sum+n; n--; }
```

computes the sum of the first n natural numbers.

4.4.2 Forced type conversions

One can convert an expression to a type by writing the name of that type between parentheses in front of the expression. For example, the conversions that happen automatically in Program 4.1 are forced in

```
printf("%d\n", 1234567890 - (int)(float)1234567890);
```

resulting in the same output of `-46`, on the same implementation.

In the next example we come back to the trick of using arithmetic on characters to convert from upper case to lower case. By forcing character type on the resulting integer code, we cause the characters to be output. See Program 4.2.

```
00 #include <stdio.h>
01
02 int main() {
03   char ch;
04   printf("How many characters to convert?\n");
05   int n; scanf("%d\n", &n);
06   while (n) {
07     scanf("%c", &ch);
08     printf("%c", (char)(ch + 'a' - 'A'));
09     n--;
10   }
11   printf("\n");
12 }
```

Output and (indented) input:

```
How many characters to convert?
   10
   ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghij
```

Figure 4.2: Converting from upper case input to its lower case version for output. Without the forced conversion to `char`, the integer for the numerical code would be printed.

The character conversion is interesting because one has to know very little about the character set. In case you are curious about the actual codes, you can force a conversion from character to integer type to trick `printf` into revealing the numerical code. See Program 4.3, which types all printable ASCII codes on one long line to keep the program simple.

Apart from the insertion of a few line breaks, this output is as shown in Table 4.4.

```
0 #include <stdio.h>
1
2 int main() {
3   for (char c = 32; c < 127; c++)
4     printf("%3d %c|", c, c);
5   printf("\n");
6 }
```

Figure 4.3: *Printable ASCII*: displaying the printable range `char` codes. To keep the program simple, the output is on one line.

```
 32  | 33 !| 34 "| 35 #| 36 $| 37 %| 38 &| 39 '| 40 (| 41 )|
 42 *| 43 +| 44 ,| 45 -| 46 .| 47 /|
 48 0| 49 1| 50 2| 51 3| 52 4| 53 5| 54 6| 55 7| 56 8| 57 9|
 58 :| 59 ;| 60 <| 61 =| 62 >| 63 ?| 64 @|
 65 A| 66 B| 67 C| 68 D| 69 E| 70 F| 71 G| 72 H| 73 I|
 74 J| 75 K| 76 L| 77 M| 78 N| 79 O| 80 P| 81 Q|
 82 R| 83 S| 84 T| 85 U| 86 V| 87 W| 88 X| 89 Y| 90 Z|
 91 [| 92 \| 93 ]| 94 ^| 95 _| 96 `|
 97 a| 98 b| 99 c|100 d|101 e|102 f|103 g|104 h|105 i|
106 j|107 k|108 l|109 m|110 n|111 o|112 p|113 q|
114 r|115 s|116 t|117 u|118 v|119 w|120 x|121 y|122 z|
123 {|124 ||125 }|126 ~|
```

Table 4.4: Result of running Program 4.3 on an implementation where codes $0, \ldots, 127$ coincide with ASCII. The output was broken up into several lines.

4.4.3 Automatic arithmetic conversions

In the following fragment, the operands of division have incompatible types.

```
int i = 1; float x = 1.0; printf("%f", x/i);
```

This requires a decision as to whether the division operator denotes the integer or the floating-point version of the operation.

Such decisions are made by *automatic type conversion*, which is based on numerical types being part of a hierarchy. If an arithmetic operator has operands at different levels in the hierarchy, then the lower-level operand is converted to the type of the other operand.

One criterion for being high or low in the hierarchy is whether one type is a subset of the other. This takes care of ordering among the integer types and among the floating-point types.

In addition, the floating-point types are placed higher in the hierarchy than the integer types. This in spite of the fact that some integers are not floating-point numbers. We saw 1234567890 as an example.

This rule has the virtue of avoiding loss of information in many cases. Between floating-point types, the lower type can convert to the higher type without loss of information. The same holds between the integer types.

4.5 Enumerations

It can be useful to have one or more finite sets of arbitrary objects rather than integers. These are specified by enumerating their members. In C such sets are modelled by means of an *enumeration*. Examples of such sets are

{ *Earth*, *Water*, *Fire*, *Air* } and { *cyan*, *magenta*, *yellow*, *black* }.

In C such types are created by means of the `enum` keyword. The above examples are implemented by the following declarations.

```
enum element {Earth, Water, Fire, Air};
enum colour {cyan, magenta, yellow, black};
```

After such a declaration, one can define variables that have the newly introduced types. One would like to write simply `element e0, e1;` or `colour c0, c1;`, but with the declarations in the above form, the definitions have to be `enum element e0, e1;` or `enum colour c0, c1;`.

C allows the introduction of names for types by means of the keyword `typedef`. We can combine the declaration of the enumeration type with `typedef` as follows:

```
typedef enum {Earth, Water, Fire, Air} element;
typedef enum {cyan, magenta, yellow, black} colour;
```

When the declaration is in this form we can write `element e0, e1;` and `colour c0, c1;`.

Sometimes one wants to work with ordered sets. Though these have a natural mapping to integers, it is better not to identify them with integers. Common examples are

{ *Monday*, . . . , *Friday* } and { *January*, . . . , *December* }

Whether the `enum` sets are ordered or not, C identifies them with integers. This allows arithmetic to be performed on them, as shown in Program 4.4. The default is for the integers to start at 0. C provides syntax to override this default. For example, one would prefer `January` to be identified with 1 rather than 0; see Program 4.4.

```
00 #include <stdio.h>
01
02 int main() {
03   typedef enum{Monday, Tuesday, Wednesday, Thursday,
04                Friday, Saturday, Sunday} Day;
05   for (Day day = Monday; day <= 10; day++)
06     printf("%d  ", day);
07
08   typedef enum {
09     January = 1,
10     // the default would be 0, which doesn't feel right
11     February, March, April, May, June, July,\
12     August, September, October, November, December
13   } Month;
14   Month m1 = February, m2 = August;
15   printf("\nMonths from February to August: %d\n", m2 - m1);
16
17   // m1 = Monday; // error: wrong type
18
19   enum Permissions {read = 4, write = 2, execute = 1};
20   printf("%d  %s\n", read, "  read only");
21   printf("%d  %s\n", read+write, "  read and write");
22   printf("%d  %s\n", read+write+execute
23                    , "  read, write, and execute"
24         );
25 }
```

Figure 4.4: Examples of enumerations. Line 15 commented out shows that, although the values of an enumeration are encoded as integers, they are distinct types. In this way errors, such as the one shown in this line, get caught by the compiler.

```
0  1  2  3  4  5  6  7  8  9  10
Months from February to August: 6
4    read only
6    read and write
7    read, write, and execute
```

Table 4.5: Result of running Program 4.4.

4.6 Miscellaneous topics concerning data types

4.6.1 `sizeof`

To avoid as much as possible to have to modify code to run on a different implementation, it is possible for the code itself to find out what the sizes of the types

are. See the use of the `sizeof` operator in the program in Figure 4.5.

```
 1 #include <stdio.h>
 2 #include <stdbool.h>
 3
 4 int main() {
 5   printf("sizes of:\n");
 6   printf("bool char short int long float double long double\n");
 7   printf(
 8     "%lu    %lu    %lu     %lu   %lu    %lu     %lu      %lu\n",
 9     sizeof(bool), sizeof(char), sizeof(short), sizeof(int),
10     sizeof(long), sizeof(float), sizeof(double), sizeof(long double)
11         );
12 }
```

Figure 4.5: Program for printing the sizes of some fundamental data types.

The output of this program depends on the implementation. Here is one example:

```
sizes of:
bool char short int long float double long double
1    1    2     4   8    4     8      16
```

The system on which the previous edition of this book was developed had a different computer architecture. The resulting difference in C implementation caused this program to give as output:

```
sizes of:
bool char short int long float double long double
4    1    2     4   4    4     8      16
```

Apparently this computer architecture gave no support for double-length integers. To make them available in C would require software simulation, which implies a performance penalty. The implementers of C on this architecture chose not to do this so that long integers are the same as integers. The C standard allows this. It is rare for an application to be affected, but such a difference can be crucial. It is important to be aware of the possibility.

The use of `sizeof` typically involves a variable to hold the result of this operator. Program 4.5 is not typical in this respect. What type should we give such a variable? Depending on the implementation of the C language, the result of `sizeof` could be a rather large integer. Usually we make the type `int` or `unsigned`. But, to be guarded against extreme situations, we should make the type `size_t`. This type has been especially introduced for the purpose. It ensures that in the implementation in use a variable of this type is allocated enough storage.

4.6.2 `printf` codes

type	desired notation	recommended code
(`short`, `long`) `int`	decimal	`d`
(`short`, `long`) `unsigned`	decimal	`u`
(`short`, `long`) `unsigned`	octal	`o`
(`short`, `long`) `unsigned`	hexadecimal	`x`, `X`
(`signed`, `unsigned`) `char`	alphabetic	`c`
`float`	decimal fraction	`f`
(`long`) `double`	decimal fraction	`lf`
`float`	scientific notation	`e`
(`long`) `double`	scientific notation	`e`

Table 4.6: `printf` codes for types discussed in this section.

Chapter 5

Memory

You may have been puzzled by the fact that both `x = y` and `x = 1` are correct assignment statements, even though they are quite different: in `x = y` the variable `x` gets the *value of* the variable `y`. In `x = 1` the right-hand side is not a variable with a value to be used, but is itself a value. The relationship between variables and their values needs to be made more precise.

5.1 The attributes of a variable

A variable is a contiguous area of memory. It always has an *address* and a *content*; it typically has a *name* and a *type* as well. Hence the four attributes of a variable:

Name The name of a variable is an identifier, which is in principle any sequence of letters, digits, or underscores. But to make identifiers and numerals easier to tell apart, an identifier cannot start with a digit.

Content Memory is a sequence of bytes, each of which consists of eight bits. Each bit is at any moment in one or two states. This is true whether or not the state has been determined intentionally. The variable's content is a contiguous area of memory.

Type The content of a variable is a meaningless sequence of bits unless we interpret these bits in a the right way. Only after this interpretation does the variable's content become data. How we interpret the content of a variable is determined by the type of data that the variable is intended to store.

Data can be numerical (various kinds of integer, two kinds of fractions) or text (various kinds of character).

Address As a variable is a contiguous area of memory, it consists of a sequence of one or more bytes. The address of a variable's first byte is the address of the variable.

The value of a variable is not one of its attributes, but is determined by its content and by its type. That is, only when we know the type of the variable can we interpret the meaning of the bits that constitute the content. For example a variable can have the bit sequence `0x3dcccccd` as its content. If the variable is of type `float`, then its value 0.1; if it is of type `int`, then its value is 1,036,831,949.

Variable or constant? Consider

```
const double G0 = 6.673e-11;
double G1 = 6.673e-11;
```

Here `G0` is a *constant* and `G1` is a variable. One might think that the compiler would replace every occurrence of `G0` by `6.673e-11`. This is not the case: `G0` is also a variable, as you may prove to yourself by printing `&G0`. This shows that `G0` has an address, and if it has an address, then it must be a variable. It is only a constant in the sense that the compiler prevents code from changing its content.

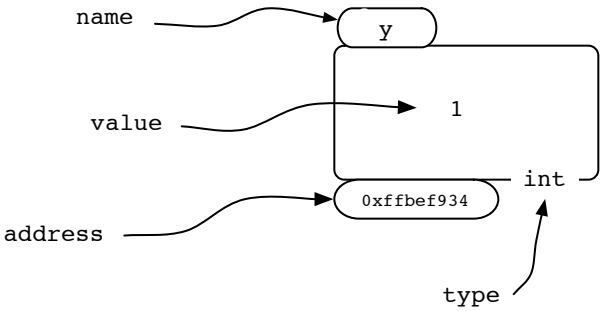


Figure 5.1: The attributes of a variable created by `int y = 1`. The address is typically different for different systems and on the same system for different compile, load, and execute cycles.

To clarify the relationship between the attributes of a variable, it is useful to outline what you need to know about data in a computer. Higher-level languages like C are designed in such a way that the programmer needs to know very little about the architecture of the computer on which the language is implemented. Here are some of the few things you do need to know.

All architectures include a processor and memory. The processor operates on data; memory stores data. Memory is a sequence of bits. Individual bits in this sequence are not directly accessible. The bits in memory are subdivided into non-overlapping cells that are eight bits wide and are called *bytes*. It is these bytes that have successive addresses. The byte with address a is followed by the byte with address $a + 1$. These addresses make the cells accessible.

Another important property of an architecture is the size of a *word*, which is the chunk of data that gets transferred between memory and processor. Many computers have four-byte words. Faster computers tend to have larger words. In the early years of the 21st century, the smallest and cheapest computers had one-byte words; these are the 8-bit microcontrollers. At the same time, the high-performance servers or workstations had 64-bit processors: eight bytes to a word.

5.2 Addresses

Given a variable's name, we can obtain its address by means of the *address operator* `&`. See Program 5.4 for examples.

The address is a nonnegative integer. It is useful to reserve one such integer that cannot be an address. This is the *null address*.

5.3 Pointers

For every type T, there is a type $T*$ called "pointer-to-T". When a variable's type is pointer-to-T for some T, it is called a *pointer* and its address is interpreted as the address of a variable of type T. A variable that has the null address as value is called a *null pointer*.

Let us create a variable `x` which is a pointer that points to an integer variable `y`. The type of `x` is `int*`; this variable is defined by `int *x;`. Here the value of `x` is the *address* of `y` rather than the value of `y`. The address is obtained by applying the address operator `&` to the variable `y`. Thus the assignment in

```
int *x, y = 1; x = &y; printf("%x\n", x);
```

causes `x` to have as value the address of `y`; `x` is said to have become a pointer to `y`. We see the address of `y`, which is the value of `x` from the print statement. This shows something like `bffff610`, which is a hexadecimal numeral. The conversion code `%x` assures conversion to this format, which is the appropriate one for addresses.

See Figure 5.2.

From the above it will be clear that the pointer `x` contains as value the *address* of `y`, not the *content* of `y`. Yet we can use `x` to obtain the value of `y` indirectly by means of the operator `*`. It is called the *dereferencing* operator.

To continue the above example:

```
int *x, y = 1; x = &y; z = *x; printf("%d\n", z);
```

causes `1` to be printed because `*x` has as value the value of the variable that `x` points to. Thus `x` has as value the address of `y` while `*x` has as value the value of `y`. It is always the case that executing

```
x = &y; z = *x;
```

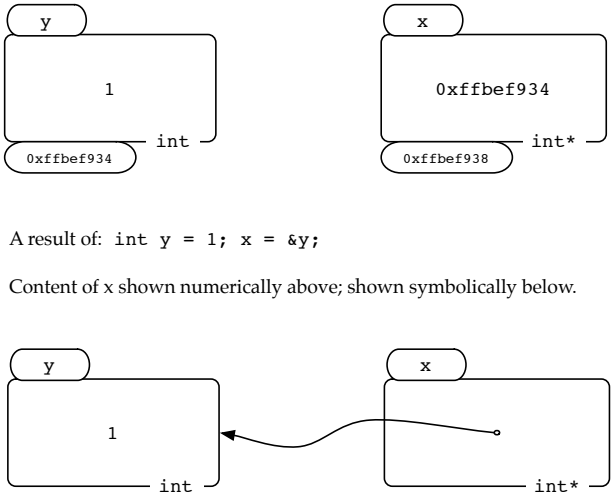


Figure 5.2: Two variables, one of which is a pointer to the other.

has the same effect on `z` as executing `z = y`. The example can be made more succinct by saying that `z = y` and `z = *(&y)` have the same effect.

A pointer variable, say, `ip`, is a variable and therefore has an address. That address can be the value of a variable, say, `ipp`. This variable could be defined as in

```
int i = 13; int *ip; int **ipp;
ip = &i; ipp = &ip;
```

One finds this usually in the shorter form

```
int i = 13, *ip = &i, **ipp = &ip;
```

With pointers one can access a variable directly or indirectly, or even doubly indirectly, as in

```
int i = 13, *ip = &i, **ipp = &ip;
printf("%d %d\n", *ip, **ipp);
```

giving

```
13 13
```

Typically direct access is used, but there are situations in which indirect access, or even doubly indirect access, is appropriate.

See Program 5.4, *Direct and Indirect*, for a direct and an indirect exchange.

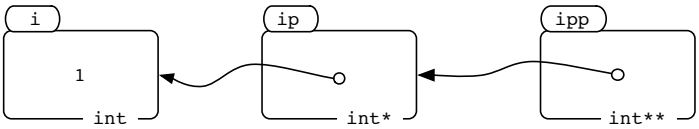


Figure 5.3: A pointer pointing to a pointer.

```
00 #include <stdio.h>
01
02 int main() {
03   int i = 0, j = 1, temp;
04   int *ip = &i, *jp = &j;
05
06   temp = i; i = j; j = temp;
07   printf("%d %d\n", i, j);
08   temp = *ip; *ip = *jp; *jp = temp;
09   printf("%d %d\n", i, j);
10 }
```

Figure 5.4: *Direct and Indirect*, exchanging the values of two variables directly and also indirectly via pointers to these variables.

5.4 Pointer errors

Of all the addresses that can be the value of a pointer variable, a relatively small subset are addresses of the first bytes of existing variables. C places no constraints on the values that can be assigned to a pointer variable. An error in such an assignment almost always causes the program to crash. See Program 5.6, *Invitation to Disaster*.

To understand the consequences of dereferencing and erroneous pointer it helps to understand the main outlines of a computer's architecture. The components of a computer that concern us here are its *processor* and its *memory*. It is the processor that operates on data. The data are stored in memory. Also stored in memory are the instructions ultimately resulting from the translation of the program's source code.

The computer's design is in one of two main categories: *von Neumann architures* and *Harvard architectures*. The former has one memory for both instructions and data. In the latter there is one memory area for instructions (*i-space*) and a separate one for data (*d-space*). In the first decades of digital computing most machines had von Neumann architectures; more recently some form of Harvard architecture is

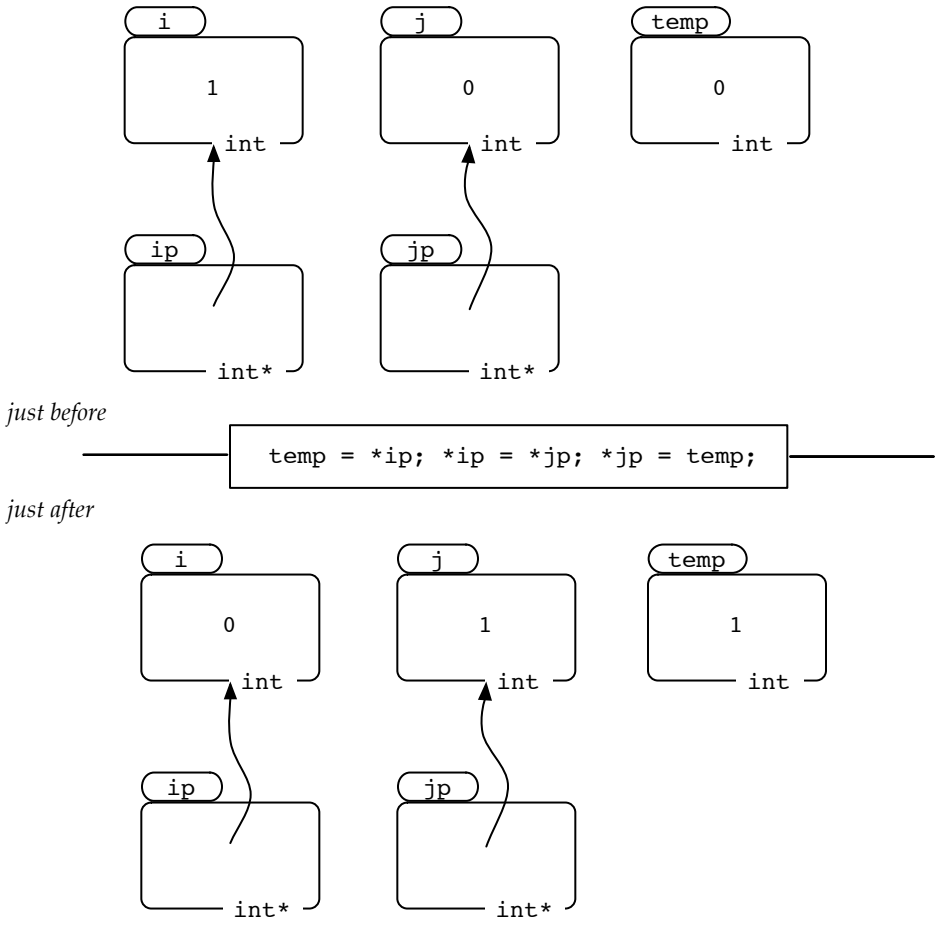


Figure 5.5: Illustrating the program in Figure 5.4.

predominant.

As long as an erroneous pointer points into d-space, dereferencing it allows execution of the program to continue, although with unintended results. Typically, sooner or later, a pointer will be dereferenced that points outside d-space. To understand the consequence of this it is useful to realize that the program may not be the only program being executed on the machine. Often there are many such programs, all under supervision of the operating system. The operating system can, and does, detect when one of the programs addresses a part of memory that is outside its d-space. In such a situation the operating system discontinues execution of such a program. If the operating system is some variant of Unix, the accompanying message is "segmentation fault".

In the case of a von Neumann architecture the situation differs only in details.

```
0 #include <stdio.h>
1
2 int main() {
3   int *ip = (int *)123456;
4   printf("%d\n", *ip); // crash
5 }
```

Figure 5.6: *Invitation to Disaster*, a program that is likely to crash.

The operating system assigns to each program a memory area that is not differentiated into i-space and d-space. It is possible for an erroneous pointer to point to an address that is an instruction's address, or even is in the middle of an instruction. This widens the repertoire of bizarre behaviours. But, just as with Harvard architectures, it is likely that an address is generated that is recognized as invalid by the operating system, followed by the program's crash.

Chapter 6

Functions

There's more to functions than what we saw of them in Chapter 3. For example, we only moved data *in* via parameters. How to get data *out* in this way? We want to execute the body of a function with different values of the actual parameters. We only did this with numerical parameters. How can a *function* be made a parameter of a function? Before all this I give a more detailed description of function definition and function that I did so far.

6.1 Blocks

A *block* is a sequence of definitions and statements enclosed in braces. If a variable defined in a block has the same name as a variable defined outside the block, then the name does not refer to the variable defined outside. The effect of defining a variable in a block is to make it impossible to reference any variable with the same name outside the block. Thus we see that, although the name is one of the attributes of a variable, this attribution is not permanent.

The properties of blocks are illustrated in Program 6.1: *Global and local*. Let us call the variables created in lines 02, 06, and 09 respectively v_0, v_1, and v_2. Note that all three have the same identifier, namely `i`. In line 10 the identifier `i` refers to v_2, and to v_2 only. There is only one variable with identifier `ip_1` in the program, and in line 11 it is the only way to access v_1. There is only one variable with identifier `ip_0` in the program, and in line 12 it is the only way to access v_0.

To understand the behaviour of Program 6.1 it may help to use the concept of *scope*. The scope of a declaration is the part of the program in which that declaration is in force. The scope of the declaration of line `02` extends from this line to line `15`. The scope of the declaration of line `06` extends from this line to line `15`. The scope of the declaration of line `09` extends from this line to line `13`. Scope needs to be contrasted with the *visibility* of a declaration. Between lines `10` through `12` three declarations are in force. Because all three use the same identifier, only the innermost declaration is visible.

The connection between identifier `i` and variable v_1 is established in line `06`, has ceased to exist between lines `09` and `13`, and is re-established in line `14`.

```
00 #include <stdio.h>
01
02 int i = 11, // i identifies variable v_0
03     *ip_0 = &i;
04
05 int main() {
06   int i = 12, // i identifies variable v_1
07       *ip_1 = &i;
08   { // block entry
09     int i = 13; // i identifies variable v_2
10     printf("value of v_2: %d\n", i);
11     printf("value of v_1: %d\n",*ip_1);
12     printf("value of v_0: %d\n",*ip_0);
13   }
14   printf("value of v_1: %d\n", i);
15 }
```

Figure 6.1: *Global and Local*: three variables, one identifier.

6.2 Function definitions

A function definition has the following structure

type name parameter-list body

The part before the *body* is called the *header* of the function. Sometimes it is necessary to make the name and type of a function known in a place where it must not be defined. In that case one writes a *function prototype*, which is the header followed by a semi-colon.

1. *type*
   is a data type such as `int` or `double`. It is the type of the value returned by the function. It can also be `void`, which specifies that the function does not return a value.

2. *name*
   is the identifier that names the function.

3. *parameter-list*
   is a list of items. The list is enclosed by parentheses; the items are separated by commas. Each item is a type followed by an identifier, which is the name of a formal parameter.

4. *body*
   is a block.

The variable names that occur in a function body can refer to three kinds of items.

1. If the name occurs in the parameter list, then the name refers to a formal parameter.

2. If the body contains a definition of a variable of that name, then the name refers to a *local variable*.

   Local variables are local in the sense that they are only present in memory when the body in which they occur is being executed. Their names are local in the sense that they refer to the local variable and not to any possibly existing variables with that name defined elsewhere.

   The memory for a local variable is allocated when execution passes the opening brace of the body; it is de-allocated upon termination of the function call. The value such a variable has is not retrieved on later re-entry into the same body*.

3. If neither of the above is the case, then the name refers to a variable defined outside any function. Such a variable is said to be a *global variable*.

6.3 Function calls

If an expression is a function call, then its evaluation takes place in the following steps.

1. The actual parameters, which are expressions, are evaluated.

2. The function call is replaced by the function body. Local variables are unassigned. Global variables retain their values. For every formal parameter a local variable is created of the same name and type and is initialized with the value resulting from the evaluation of the actual parameter corresponding to the formal parameter. This process is called *parameter passing*.

3. Execution of the body starts at the first statement of the body. How the body's execution ends depends on whether the function returns a value.

   We first consider the case where the function does return a value. Normally the final step in executing the body is the execution of a *return statement* with an argument. The effect of such a statement is that the value of the argument is substituted for the expression that constitutes the function call. If execution reaches the end of the body without encountering a `return` statement, then the effect is as if a `return` statement with no expression were executed; the value that replaces the function call is not defined.

*This behaviour is the default. We will encounter an exception later.

We now consider the case where the function does not return a value. In this case the definition has the keyword `void` in place of the return type. Again execution of `return` terminates execution of the body. But in this case there is no expression after `return`.

See Program 6.2, *Evanescent Exchange*, for an example of function definition and call.

```
00 #include <stdio.h>
01
02 void foo(int u, int v) {
03   int k = u; u = v; v = k;
04 }
05 int main() {
06   int x = 1, y = 2;
07   printf("%d %d\n", x, y);
08   foo(x, y);
09   printf("%d %d\n", x, y);
10 }
```

Figure 6.2: *Evanescent Exchange*: a function possibly intended, but certainly failing, to exchange its actual parameters.

Exchanging the values of two variables is something frequently needed—ideal code for encapsulating in a function. Consider Program 6.2, *Evanescent Exchange*. When we run this program, we find that the values of `x` and `y` have not been exchanged. How can this be explained?

Remember that in parameter passing, the actual parameters `x` and `y` are regarded as expressions and their *values* are used to initialize the formal parameters `u` and `v`. These formal parameters are variables local to the body of the function called. These variables are distinct from `x` and `y`. Hence the values of the formal parameters `u` and `v` are exchanged; the values of `x` and `y` remain unchanged.

How *do* we define a function that exchanges the values of the variables that occur as actual parameters in a call to the function?

Consider again Program 5.4. Here there are two variables that get their values exchanged. It is done in two ways: directly and indirectly via pointers to the variables. The direct method in the function `foo` in Program 6.2 results in the values of the local copies of the actual parameters being exchanged, but not the actual parameters themselves. Program 6.3, *Exchange*, uses the indirect method. As always, the actual parameters are copied into the formal parameters. But here these copies are pointers and they are used to access the values of the actual parameters, so that the desired exchange is effected.

Parameter-passing mechanisms In step 2 in the above description of calling a function we saw that to the formal parameters there correspond local variables of

the same name and type. Upon entering the body of the function, these variables are assigned the values obtained by evaluating the actual parameters in the previous step.

This is the parameter-passing mechanism of C. It is called *call-by-value*. Some other programming languages have a different mechanism, possibly as alternative to call-by-value.

Without the existence of pointers, call-by-value would exclude the possibility of using a parameter for output. In Program 6.3 we see how a pointer can be passed by value, yet enable a function call to produce output via a parameter. This specific use of pointers is sometimes called "call-by-reference". This meaning of the term is restricted to C. C++, for example, has two distinct parameter-passing mechanisms. One of these is call by value. The other has no counterpart in C, and is named "call by reference".

```
00 #include <stdio.h>
01
02 void swap(int* u, int* v) {
03 // Purpose: Exchange values of the variables
04 // pointed to by the actual parameters.
05 // Preconditions: None.
06   int k = *u; *u = *v; *v = k;
07 }
08 int main() {
09   int x = 1, y = 2;
10   printf("%d %d\n", x, y);
11   swap(&x, &y);
12   printf("%d %d\n", x, y);
13 }
```

Figure 6.3: *Exchange*: exchange actual parameters.

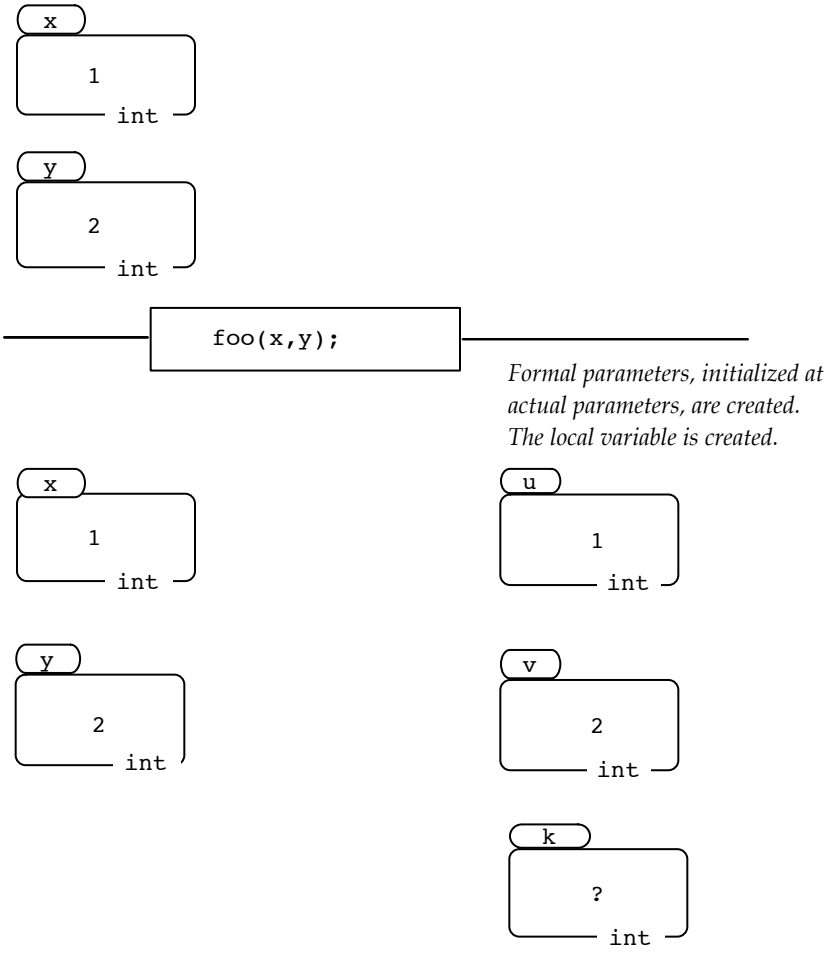


Figure 6.4: Illustrating the program in Figure 6.2: function entry.

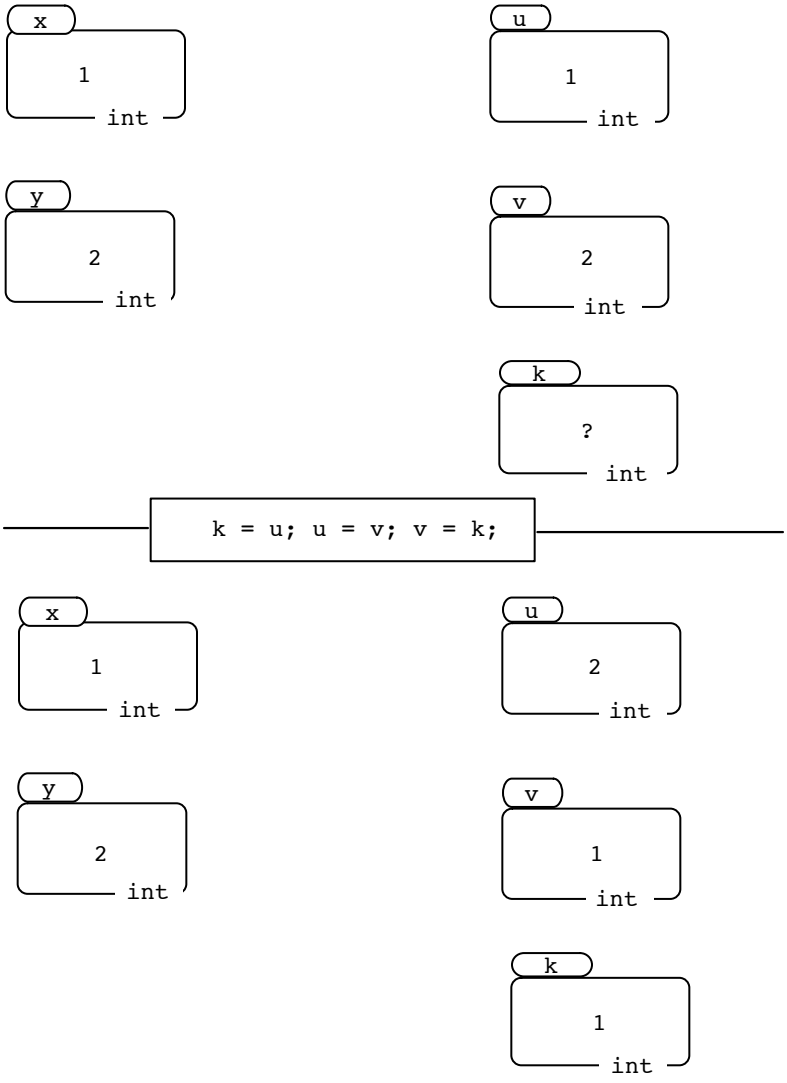


Figure 6.5: Program in Figure 6.2: the exchange operation.

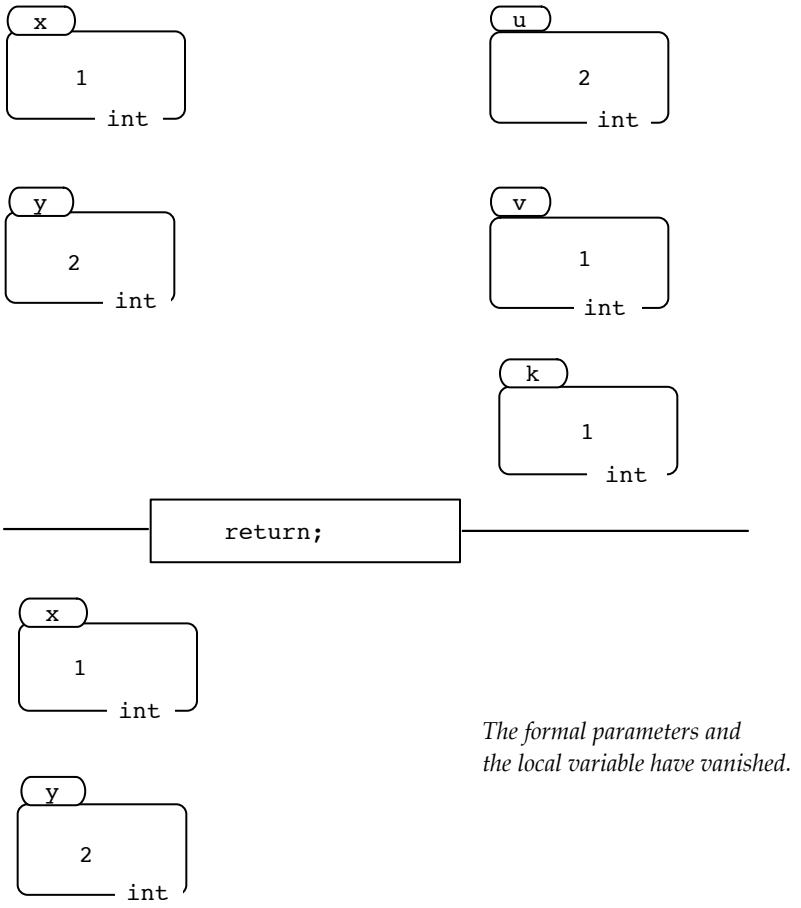


Figure 6.6: Program in Figure 6.2: function exit—nothing happened.

6.4 Functions as actual parameters

Thus far we have encountered three kinds of things that have names: variables, constants, and functions. Let's compare them. Constants have all the attributes of variables: they are areas of memory that have an address, a type, a content, and, typically, a name. Constants differ only in the unchangeability of their contents.

The body of a function consists of code, and code occupies memory. So we can regard a function as an area of memory. It also has a name. Apparently, functions have their main attributes in common with variables. But there are important differences.

To start with, the content of the memory area occupied by a function body cannot change. This makes a function more like a constant than like a variable. Like a constant, a function has a type. But where a constant can have only one of a small number of predefined types, the type of a function is determined by its definition: the type is the ordered list of the types of its formal parameters paired with the type of the returned value.

Another way in which functions are different from variables is that their content cannot be copied to other functions. For variables such copying makes sense: we do it so we can operate on the copy while preserving the original, or vice versa. As C provides no operations on functions, copying the content of a function would serve no purpose. Every function defined in a program is stored in one area of memory only, and its content cannot be changed by any action of the program.

Variables and constants have addresses, which are the addresses of the first byte of their memory areas. As a function is also a memory area, it has an address. Hence the address operator `&` applies to functions. For example, given the functions

```
int up(int x, int y)   {return x <= y;}
int down(int x, int y) {return x >= y;}
```

we get the addresses of these functions from the expressions `&up` and `&down`.

To demonstrate how we can use function addresses, let us revise Program 3.4. The task of comparing two items is very common. For example, we might want to compare the absolute values of two numbers. We need to generalize the outcomes "smaller" and "larger" to "before" and "after".

The possibility of functions to be actual parameters allows us to vary the comparison criterion while leaving the comparison code unchanged. See Program 6.7, *Parametrized Compare*.

We are going to make the comparison operation into an additional formal parameter of the `sort3` function. The actual parameter cannot be the comparison function itself, but it can be the address of this function. The formal parameter, like all formal parameters, needs to have its name and type listed. We do this with the aid of

```
typedef int (*cmp)(int, int);
```

The keyword `typedef` defines a type and introduces a name for it. Read this as: the type `cmp` ("number comparison") is a pointer (indicated by the `*` in `*cmp`) to a

```
00 #include <stdio.h>
01 #include <stdlib.h> // for abs()
02
03 int intcmp(int a, int b, int (*cmp)(int, int)) {
04   if ((*cmp)(a,b) < 0) return -1;
05   if ((*cmp)(a,b) > 0) return 1;
06   return 0;
07 }
08 int numComp(int a, int b) {
09   if (abs(a) < abs(b)) return -1;
10   if (abs(a) > abs(b)) return 1;
11   return 0;
12 }
13 int main() {
14   printf("Input two numbers\n");
15   int a, b;
16   scanf("%d %d", &a, &b);
17   if (intcmp(a, b, &numComp) < 0)
18     printf("first before\n");
19   else if (intcmp(a, b, &numComp) > 0)
20     printf("first after\n");
21   else printf("equal\n");
22 }
```

Figure 6.7: *Parametrized Compare*, similar to Program 3.4.

function that takes two integer parameters and delivers an `int` result. This type is that of a *function pointer*.

Functions as parameters of functions can be summarized by Program 6.9 where almost nothing else happens.

6.5 Dangling pointers

Recall from Program 6.1 that when execution reaches line 14, the variable v_2 no longer exists. In general, local variables only exist during execution of the block in which they are defined.

Now, recall that the body of a function is a block. Therefore, on exit from a function, the function's local variables have disappeared. There is no danger that such a non-existent variable is accessed via its name, because the association of that name and that variable only exists when the body in being executed. But it is possible that a pointer that is not local to the function has as value what was once the address of a local variable of the function. This kind of pointer error goes by

```
#include <stdio.h>

typedef int (*cmp)(int, int);
void two_sort(int* x, int* y, cmp f) {
// Purpose: sort parameters according to comparison function f
// Prerequisites: None
  int temp;
  if ((*f)(*y, *x)) { temp = *x; *x = *y; *y = temp; }
}
int incr(int x, int y) { return x <= y; }
int decr(int x, int y) { return y <= x; }

int main() {
    int x = 2, y = 1;
    two_sort(&x, &y, &incr);
    printf("%d %d\n", x, y);
    two_sort(&x, &y, &decr);
    printf("%d %d\n", x, y);
}
```

Figure 6.8:

```
#include <stdio.h>

int f(int x) {return x;}
int apply(int (*f)(int), int x) {return (*f)(x);}

int main() {
  printf("%d\n", apply(&f, 1234));
}
```

Figure 6.9: Example of function with function parameter.

the name of *dangling pointer*. See Program 6.10: *Dangling Pointer*, for an example.

6.6 Exercises

6.6.1 Plotting sine

Without graphics one can get a rudimentary form of plotting by printing lines consisting of spaces followed by some visible character. This exercise is the provide

```
0 #include <stdio.h>
1
2 int* foo(int i) { return &i; }
3 int main() {
4   int* ip = foo(13);
5   for (int i = 0; i < 12; i++)
6     printf("%d %d\n", i, *(ip++));
7 }
```

Sample output:

```
0 13
1 1795844960
2 32767
3 189431498
4 1
5 0
6 0
7 0
8 8
9 1795844944
10 32767
11 1795850329
```

Figure 6.10: *Dangling Pointer*: a program with undefined behaviour. It prints unpredictable integers and may crash before dying a natural death. If the first printed integer is 13, that can be explained, but is not predicted by the C language definition.

the functions missing in Program 6.11 so as to give the output shown.

```
00 #include<stdio.h>
01 #include<math.h>
02
03 double sine(double x) {
04   double y, xScale = 0.5, yScale = 20.0;
05   // scale factors to make plot fit
06   x *= xScale;
07   y = 1 - sin(x);
08   return y*yScale;
09 }
10 typedef double (*func)(double);
11 void plot(double a, double h, double b, func f, char ch);
12 int main() {
13   plot(0, 1, 22, &sine, 's');
14 }
```

Figure 6.11: Program to plot a part of the sine function.

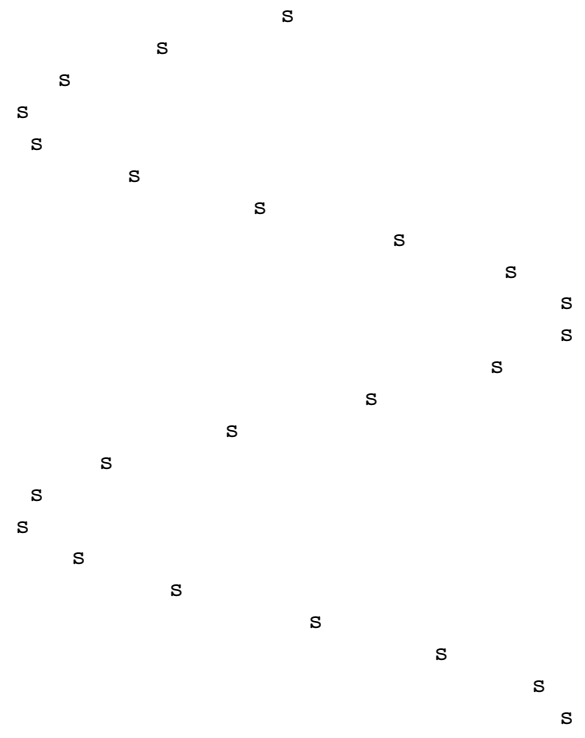


Figure 6.12: Plot of a part of the sine function.

6.6.2 Cannon ball trajectory

This exercise is to plot the trajectory of a cannon ball fired at an elevation of 75 degrees and a muzzle velocity of 150 m/sec. Assume 9.81 m/sec^2 for the acceleration of gravity. Ignore the effect of the atmosphere.

If the positions of the cannon ball are plotted at intervals of one second, you should get the output shown in Figure 6.13. Insert a one-second delay between successive lines. In this way you get a real-time simulation of the flight of the projectile.

A delay can be obtained by means of a loop that is traversed a large number of times. Such a loop does not have to do anything in particular. See the function `wait` in Program 6.14: *Countdown*. The delays obtainable are restricted to integer multiples of a whole second. This is because the library function `time` does not give fractional seconds.

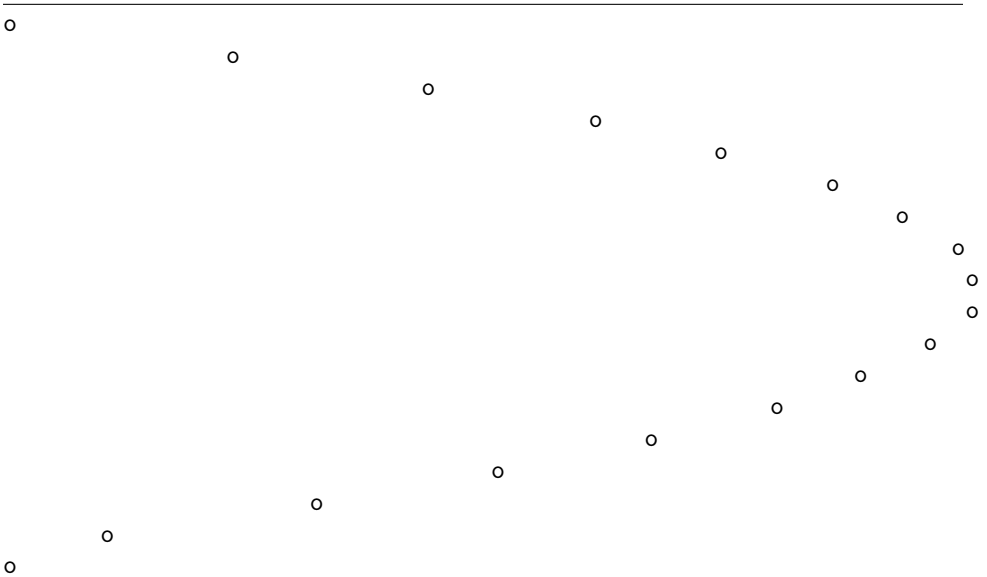


Figure 6.13: Trajectory of cannon ball.

```
00 #include <stdio.h>
01 #include <time.h>
02
03 void wait(int n) {
04 // Purpose: to delay the calling function by n seconds.
05 // Preconditions: n >= 0.
06   int t = time(NULL);
07   while (time(NULL) < t+n);
08 }
09
10 int main() {
11   for (int i=10; i>0; i--) {
12     printf("%d\n", i); wait(1);
13   }
14   printf("GO!\n");
15 }
```

Figure 6.14: *Countdown:* using a `wait` function.

Chapter 7

Expressions

In Program 2.2 we used an expression to compute a formula for gravitational attraction:

```
printf("%f newtons.\n", G*(m1*m2)/(r*r));
```

Expressions are an advance over the early days of programming, when every statement corresponded to a single machine instruction. As a result, statements could not perform more than one operation. To get the value `f` of `G*(m1*m2)/(r*r)`, one had to do instead something like:

```
f = G; f = f*m1; f = f*m2; f = f/r; f = f/r;
printf("%f newtons.\n", f);
```

Clearly, expressions, which can be as complex as one likes, are very convenient.

We will see that expressions are not only used for computing values of numerical formulas. Expressions also compute conditions to help in complex decisions. They manipulate bit vectors in support of systems programming tasks. We will see that assignment statements are also expressions, even if they do not involve arithmetic, logic, or bit operations.

7.1 The structure and value of an expression

Expressions follow to a large extent the evaluation conventions for arithmetic that we learned in school. For example, `a+b*c` means `a+(b*c)` rather than `(a+b)*c`. A difference is that expressions in programs are not restricted to arithmetic, but are used for a wider variety of applications. Also, some of the details may be different. Your high school math teacher probably had a definite opinion about whether $1/r/r$ is $1/r^2$ or whether it is just a complicated way of writing "1". C also has a definite opinion, and it may not be the same as that of high-school teachers (who may not have agreed among themselves on this particular example).

Operators and operands An expression may contain *operators*, *operands*, and parentheses. In `g*m1*m2/(r*r)` the operators are `*` and `/`, denoting the operations of multiplication and division, respectively. The operands are the values that the operators act on. In this case the operands are `g`, `m1`, `m2`, and the two occurrences of `r`.

The multiplication and division operators take two operands; they are therefore called *binary* operators. Other binary arithmetic operators are `-`, `+`, and `%`. The latter is the *remainder* operation that gives the remainder upon division of the left operand by the right operand. Other binary operators exist, to which we will come presently.

The *unary* operators take one operand. For example, the minus in `-273` is unary and it is a different minus from the binary one in `a-b`. The first acts on a single value, reversing its sign. Similarly, C has a unary minus operator, using the same symbol `-`. The unary plus was added for reasons of symmetry, not because there was any urgent need for it.

School mathematics leaves it at unary and binary operators. In C we have *conditional expressions* consisting of an operator with three operands, a *ternary* operator. For example, instead of printing the minimum of two numbers with

```
int a,b; scanf("%d %d", a, b);
if (a<b) printf("%d\n", a);
else     printf("%d\n", b);
```

we can use a conditional expression and write

```
int a,b; scanf("%d %d", a, b);
printf("%d\n", a<b ? a : b);
```

The ternary operator consists of the two symbols `?` and `:`.

Structure and value The structure of an expression is implied by a definition such as:

> *An expression E is either atomic or it is composite. If it is atomic, then it consists of a variable or literal only. If it is composite, then it consists of one operator (unary, binary, or ternary) with its one, two or three operands, which are expressions. The latter are the subexpressions of E.*

The value of an expression E is defined in a way that reflects its structure:

- if E is a literal or a variable, then the value of E is the value of that literal or variable
- if E is composite, then the value of E is the result of applying the operator of E to the values of the subexpressions of E

A definition like this, which defines an entity in terms of itself, is called a *recursive* definition. Its recursive nature allows expressions to be arbitrarily complex. It allows expressions to be *nested* inside an expression.

For example, `a+((b/c)-d)` is a composite expression where the operator is `+`. It is a binary operator, so it has a left operand, which is `a` and a right operand, which is `(b/c)-d`. The left operand `a` is an atomic expression, as it does not consist of an operator and operands. The right operand `(b/c)-d` is a composite expression, where the operator is `-`. It is a binary operator, so it has a left operand, which is the expression `b/c` and a right operand, which is the atomic expression `d`. The left operand is a composite expression with `/` as operator and two atomic expressions as operands.

Associativity and precedence In the above example, it was possible to determine the structure of `a+((b/c)-d)` because the parentheses determine at every stage what is the operator and what are the operands. Let us call such an expression *fully parenthesized*.

It is not always convenient to write fully parenthesized expressions. For example, this would mean writing `a+((b+c)+d)`, `a+(b+(c+d))`, `(a+b)+(c+d)`, `((a+b)+c)+d`, or `(a+(b+c))+d`. All of these should have the same value, so it is reasonable to be allowed to write `a+b+c+d`.

In fact we are. But `a+b+c+d`, still is interpreted as an expression according to the definition, so it is interpreted as exactly one of the above fully parenthesized expressions, namely `((a+b)+c)+d`. This interpretation is implied by a property of the addition operator: we say that it *associates to the left*. In general it means that if we write α `+` β `+` γ, where α, β, γ are any expressions, it means (α `+` β) `+` γ. All arithmetic operators associate to the left, so that `1/r/r` means`(1/r)/r`, which typically has a value that is different from `1/(r/r)`.

The property of associating to the left or to the right is sufficient to elucidate the structure of an expression without parentheses and with multiple operators, if these operators are the same. Often they are not and then we need a second property of operators. It is called *precedence*. For example, `a+b*c` means `a+(b*c)` rather than `(a+b)*c`. This is because the multiplication operator has a higher precedence than addition. See Appendix A for a listing of operators and their precedences.

7.2 Arithmetic operations

The arithmetic operators are `*` (multiply), `/` (divide), and `%` (remainder) with higher precedence, and `+` (add) and `-` (subtract) with lower precedence.

The remainder operation is only intended for integers. The other operations come in pairs, one for integers and one for floating-point numbers. If both operands of `/` are recognized as integers, then the meaning is *integer division*, so that the value of `1/2` is zero. If at least one of the operands has a floating-point type, then it means *floating-point division*, so that the value of `1.0/2.0` is one half.

Integer division gives an integer result, which is obtained by removing the

fractional part from the quotient. In fact, for all integers n and k we have that $n = (n/k) * k + n\%k$.

Multiplication, addition, and subtraction are also overloaded and come in integer and floating-point versions. On a computer with four-byte integers we get a striking difference between the values of the two expressions

`1234567890 + 1234567890` and `1234567890.0 + 1234567890.0`

as well as between

`1234567890 * 1234567890` and `1234567890.0 * 1234567890.0`

The integer versions give nonsense because of overflow, whereas the floating-point versions may give an incorrect result that, however, approximates the true value closely enough for most purposes.

7.3 Boolean operations

An expression can have one of the Boolean values `true` or `false`. Such an expression occurs as a condition in a selection statement or in an iteration statement. In a selection statement, the Boolean expression steers the execution into one branch or the other. In an iteration it determines whether execution continues.

In C one can use an integer instead of a Boolean value. The role of `false` is played integer 0; any other integer value plays the role of `false`.

An atomic numerical expression is a variable or a literal. For Boolean expressions, there is an additional possibility: it can be a *relational* or an *equality* expression. A relational expression is consists of two numerical operands and one of the binary operators

`<` `>` `<=` `>=`

An equality expression consists of two numerical operands and one of the binary operators

`==` `!=`

Note the difference between the equality operator `==` and the assignment symbol `=`.

Like their numerical counterparts, Boolean expressions can be nested. This is useful because conditions may be needed to express complex rules. Take for example the rule that tells from the number `y` of a year whether its length is 365 or 366 days. The rule can be expressed as

```
if ((y % 400) == 0) len = 366;
else if ((y % 100) == 0) len = 365;
else if ((y % 4) == 0) len = 366;
else len = 365;
```

We can say more directly that a number is that of a leap year if it is divisible by four unless it is a century year that is not divisible by four hundred. We can express such a complex condition in a single Boolean expression by means of the Boolean operations *and*, *or*, and *not*. These are denoted by the operators `&&`, `||`, and `!`, respectively. Thus we can rewrite the above nested if-statement as one that is not nested, but instead has a nested Boolean expression in its condition:

```
if ((y % 400) == 0 ||
    ((y % 4) == 0 && (y % 100) != 0)
   ) len = 366;
else len = 365;
```

The Boolean operations are defined in Table 7.1.

x	y	x && y	x || y	! x
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

Table 7.1: Definition of the Boolean operations `&&` , `||`, and `!`.

7.4 Expressions and statements

In its pure form, an expression is written to be *evaluated*: its value is obtained, but the process of evaluation does not cause a change of state. A statement, in its pure form, is the opposite: it is written to be *executed* rather than evaluated. The process of execution does not yield a value, but typically causes the state to change. The following line, which causes the decimal digits of n to be printed, contains examples of both:

```
while (n) { printf("%d", n%10); n /= 10; }
```

The entire line is a pure statement. The text `n%10` is a pure expression.

As the qualification "'pure" in the above paragraph suggests, it is also possible for an expression to change the state. Such an expression is said to have a *side effect*. The text `n /= 10` is an example of such an expression.

It is a rule in C that, whenever E is an expression,

E;

is a statement; it is called an *expression statement*. This only makes sense if E is an expression with a side effect. If you would ever see the statement

```
x;
```

it is a mistake, though it may be perfectly legal C (which it is, whenever `x` is a variable). It is a mistake because it has no effect. Many function calls are expression statements: they look like

```
f(...);
```

This need not be a mistake because the execution of `f` may have a desired effect on an output parameter or on a global variable.

7.5 Increment and decrement operators

We first encountered `n++` as a convenient shorthand for `n = n+1`. There is, however, more to it. The symbol `++`, introduced in Chapter 3, is a postfix operator: "postfix" because it is written after the operand, which is the variable it acts on. Because `n++` is an operand-operator combination, it is an expression, and it has a value. It is an expression whose evaluation has a side-effect, that of incrementing the operand.

The value of the expression `n++` is the value of `n` before the `++` operator has acted. One way to remember this is is to read `n++` as "take value, then increment". This operator has a prefix counterpart, as in `++n`, that is similar to the postfix version, except that the value of `++n` is the value of `n` after the increment.

Increment and decrement operators act on variables only. Moreover, these variables have to be of integral type. Thus `(n++)++` is not a valid expression because the outer increment attempts to act on a value rather than on a variable.

When an expression such as $E_1 + E_2$ is evaluated in C, the order of evaluation is not defined; that is, it is not defined which of the two subexpressions is evaluated first or whether they are evaluated in parallel. This implies that we have to be careful with the use of increment and decrement operators. For example if `n` and `m` have the values 1 and 2 respectively, the value of

```
++n + ++n*m
```

is not defined in C: it could be 7 or 8.

7.6 The assignment statement is an expression statement

```
x = y;
```

is an *expression statement*. This is so because `x = y` is an expression. As such it has a value. Every assignment expression has as value the value of the right-hand side. This fact is often used. As an example, let us consider

```
i = j = 0
```

Because the assignment operator associates to the right, the meaning is that of `i = (j = 0)`. Because the value of an assignment statement is that of its right-hand side, `j = 0` evaluates to 0. As a result the effect of `i = j = 0` is to set `i` and `j` both to 0. Both `i = j = 0` and `j = 0` have a value, which is 0.

As another example, let us consider

```
while ((c = getchar()) != EOF) ...
```

where `getchar` is a function that returns the first character of some sequence of characters or returns a non-character value `EOF` in case the sequence is empty. The expression `c = getchar()` is evaluated for its effect: to make `c` equal to the next character. The value of the expression `c = getchar()` is used to check whether there is a next character.

In this example we see that the assignment statement is also an expression can be useful. A disadvantage of this flexibility is that

```
if (x = y) { ...
```

cannot be flagged as an error, even though it usually is. It cannot be flagged as an error because `x = y` evaluates to the value of `y` and the condition does not have to be restricted to zero or one: any `int` is fine; all that matters is whether it is zero or not. Therefore this line needs to be checked to see if it shouldn't be

```
if (x == y) { ...
```

7.7 Fused assignment operators

As we noted, `(x++)++` is not a correct statement or expression. But we do not have to write `x = x+2`: we can shorten this to `x += 2`. We call `+=` a *fused assignment operator* because the `+` and the `=` are fused in a single operator.

For a simple left-hand side such as `x` this is not much of an improvement, though it is commonly used in such a situation. The fused assignment operator is more readily appreciated as an opportunity to simplify something like

```
incrTable[offset[prev + next]] = incrTable[offset[prev + next]] + 2;
```

to

```
incrTable[offset[prev + next]] += 2;
```

An expression of the form E_1 *op*= E_2 is equivalent to $E_1 = (E_1)$ *op* (E_2), except that E_1 is only evaluated once. Note the parentheses — otherwise we would know neither the value nor the effect of something like `x *= y+1`.

E_1 *op*= E_2 makes sense for *op* equal to any of

```
-  +  *  /  %  &  |  <<  >>  ^
```

Indeed, all these operators can be fused with assignment.

7.8 Conditional expressions

The value of any expression can be made to depend on the value of a Boolean expression. Such an expression is called a *conditional expression*. For example,

```
(y%4 == 0) ? 366 : 365
```

has the value 366 or 365, depending on whether the value of `y%4 == 0` is `true` or `false`.

We can use conditional expressions to improve code such as the following.

```
int yearLen(int yr) {
  int len;
  if (yr%400 == 0) len = 366;
  else if (yr%100 == 0) len = 365;
  else if (yr%4 == 0) len = 366;
  else len = 365;
  return len;
}
```

The general form of a conditional expression is

$$expr_0 \texttt{ ? } expr_1 \texttt{ : } expr_2$$

The rules for conditional expressions are simplest when $expr_1$ and $expr_2$ have values of the same type. In that case, the entire conditional expression has that type. Certain variations in the types of $expr_1$ and $expr_2$ are allowed. For most programmers this possibility is not important enough to warrant studying the rules.

With conditional expressions the above function simplifies to the following.

```
int yearLen(int yr) {
  return
    (yr%400 == 0) ? 366
                  : (yr%100 == 0) ? 365
                                  : (yr%4 == 0) ? 366
                                                : 365;
}
```

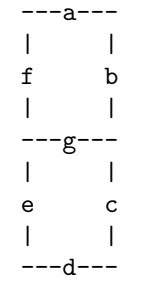
In this example abstinence from conditional expressions is bearable. This is not the case with the following.

7.8.1 Example: OCR

OCR stands for "Optical Character Recognition". It is software that takes as input an array of pixels resulting from the scanning of a text document. It produces as output the sequence of characters contained in the document.

This needs to be done in several stages. First the characters have to be isolated in the image. For each character, features are extracted from the pixels. Recognition occurs on the basis of the features found.

One can simplify the software and/or speed up recognition by limiting the style of characters to be recognized. An extreme case would be to limit oneself to digits and to require that the digits would be represented according to the *seven-segment display*:



Each of the seven segments `a, b, ..., g` is a light-emitting device that can be either switched on or not. For example, if all are switched on, the result is the digit 8. If we then switch off segments `e` and `d`, then we have the digit 9, sort of. In this way each of the digits 0 through 9 can be recognized by which of the segments is switched on. This requires a program to make the right selection among digits on the basis of the seven conditions `a`, `b`, `c`, `d`, `e`, `f`, and `g`.

This is an example where it helps to write a decision table before coding with ifs and elses. The decision table is the table on the left.

a	b	c	d	e	f	g	action
1	1	1	1	1	1	0	0
0	1	1	0	0	0	0	1
1	1	0	1	1	0	1	2
1	1	1	1	0	0	1	3
0	1	1	0	0	1	1	4
1	0	1	1	0	1	1	5
0	0	1	1	1	1	1	6
1	1	1	0	0	0	0	7
1	1	1	1	1	1	1	8
1	1	1	0	0	1	1	9

d	f	e	a	b	c	g	action
1	1	1	1	1	1	1	8
1	1	1	1	1	1	0	0
1	1	1	0	0	1	1	6
1	1	0	1	0	1	1	5
1	0	1	1	1	0	1	2
1	0	0	1	1	1	1	3
0	1	0	1	1	1	1	9
0	1	0	0	1	1	1	4
0	0	0	1	1	1	0	7
0	0	0	0	1	1	0	1

If we use the table on the left directly for translation to code, then the first test would on `a`, the second on `b`, and so on. It would be better to test first on `d` because its column has 6 ones and 4 zeros, which is more balanced than the 7 ones and 3 zeros in the column for `a`. Because of considerations like this we end up with the ordering of columns shown in the table on the right.

In the table on the right not only the columns have been reordered, but the rows also: they have been sorted into reverse alphabetical order. As a result we can separate the top part of the table with a single test on `d`. Within that top part we can separate the top part by a single test on `f`. In this way the table has been arranged for a nested conditional expression that tests on `a, b, ..., g` in

the order as they occur in the table on the right. One can now transcribe the table into a conditional expression as in Program 7.1.

```
00 int digit( int a, int b, int c, int d, int e, int f, int g) {
02 // Purpose: return decimal digit on the seven-segment display
01 // Preconditions: all parameters are boolean values.
03 // if input valid; returns -1 otherwise.
04   return d ? f ? e ? a ? b ? (c ? (g ? 8 : 0) : -1)
05                             : -1
06                         : b ? -1
07                             : (c ? (g ? 6 : -1) : -1)
08                     : a ? b ? -1
09                             : (c ? (g ? 5 : -1) : -1)
10                         : -1
11                 : e ? a ? b ? (c ? -1 : (g ? 2 : -1))
12                             : -1
13                         : -1
14                     : a ? b ? (c ? (g ? 3 : -1) : -1)
15                             : -1
16                         : -1
17             : f ? e ? -1
18                     : a ? b ? (c ? (g ? 9 : -1) : -1)
19                             : -1
20                         : b ? (c ? (g ? 4 : -1) : -1)
21                             : -1
22                 : e ? -1
23                     : a ? b ? (c ? (g ? -1 : 7) : -1)
24                             : -1
25                         :(b ? (c ? (g ? -1 : 1) : -1) : -1)
26   ;
27 }
```

Figure 7.1: *OCR*: a function to recognize the digit on a seven-segment display.

7.9 Operations on bit vectors

Often the data that a computation acts on represent entities that exist outside the computer: measurements, images, sounds, accounting data, text, ... It is also the case that much programming effort goes into compilers, operating systems, or network protocols. In such programs it is common to regard the bits in memory not as the representation of a number, but as bit vectors.

A bit is similar to a Boolean in that both have two possible values. In the case of

a bit these values are denoted as `0` and `1`. As a result of this similarity, the Boolean operations have analogous counterparts among the bit operations:

- the Boolean *and* with operator `&&` is analogous to the bitwise *and* with operator `&`
- the Boolean *or* with operator `||` is analogous to the bitwise *or* with operator `|`
- the Boolean *not* with operator `!` is analogous to the bitwise *not* (also called *complement*) with operator `~`

In addition, the bitwise operations include an *exclusive or* with operator `^`.

x	y	x & y	x | y	x ^ y	~x
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Table 7.2: Definition of the bitwise operations `&` , `|` , `^`, and `~`.

Note in the table that, for any bit value `x`, we have that `x|0`, `x&1`, and `(x^y)^y` equal `x`. Also, `x&0` equals `0`, and `x|1` equals `1`, whatever the value of `x`. These identities are displayed in Table 7.3.

x	x|0	x&1	(x ^ y) ^ y	x|1	x&0	x^x
0	0	0	0	1	0	0
1	1	1	1	1	0	0

Table 7.3: Formulas for important identities. The expressions heading the first three columns are equal to `x`. The expressions heading the last three columns have values `1` and `0` as indicated.

In C the bitwise operations act not on a single bit, as shown in the table, but bit-by-bit on entire words. Accordingly, the parameters of the bitwise operators are of type `unsigned` or `int`.

Examples of bitwise operations Common operations on a bit in a given position in a bit vector include changing that bit to `1` ("setting the bit"), changing that bit to `0` ("resetting, or clearing, the bit"), and finding out whether the selected bit is `0` or `1` ("reading the bit").

To perform these operations, we use a "mask", an integer containing zeros everywhere and a one in the position where we want to set, reset, or read. We number the bits of a four-byte unsigned integer so that 0 and 31 are the positions of the

least and most significant bits, respectively. To set, reset, or read at position n the mask `m` is 2^n, which has zeros everywhere except at position n, for $n = 0, \ldots, 31$.

We use this mask for the desired operations as follows.

- To set the bit `b` of integer `x`, at position n, we change `b` to `b|1`, which is `1` according to Table 7.3. We get `b|0`, which is the unchanged `b`, at the other positions. That is, we change `x` to `x|m`.
- Similar reasoning tells us to reset the bit of integer `x` at position n by changing `x` to `x&~m`.
- To read the bit at position n we test `x&m`. The result is either a bit vector of all zeros or all zeros except for a `1` in position n. Which is the case is determined by evaluating `x&m == 0`.

See Table 7.4 for an example.

```
position   3322 2222 2222 1111 1111 1100 0000 0000
position   1098 7654 3210 9876 5432 1098 7654 3210
                           ^
m          0000 0000 0000 0000 0010 0000 0000 0000
x1         0011 0001 1010 1100 1010 0001 1011 1011
x2         0011 0001 1010 1100 1000 0001 1011 1011
x2|m       0011 0001 1010 1100 1010 0001 1011 1011 (set)
~m         1111 1111 1111 1111 1101 1111 1111 1111
x1&~m      0011 0001 1010 1100 1000 0001 1011 1011 (reset)
x1&m       0000 0000 0000 0000 0010 0000 0000 0000 (read)
x2&m       0000 0000 0000 0000 0000 0000 0000 0000 (read)
```

Table 7.4: Setting, resetting, and reading the bit at position 13. The two top lines number the positions of a 32-bit word: read vertically from $\begin{smallmatrix}3\\1\end{smallmatrix}$ on the left to $\begin{smallmatrix}0\\0\end{smallmatrix}$ on the right. The mask `m` has zeros everywhere except in position 13. Bit vectors `x1` and `x2` differ only in position 13.

Shift operations The bitwise operations in Table 7.2 are really operations on single bits, and they are extended bit-by-bit to an entire bit vector. The *shift operators* `<<` and `>>` are also classified among the bitwise operators, but they only make sense for an entire word. If `v` is a word, then `v << n` is the word where every bit of `v` is shifted `n` positions to the left. The vacated bits, the `n` least significant bits, are filled with zeros. Similarly `v >> n` shifts to the right. What the vacated bits are filled with depends on the implementation. Thus shifting right is the less useful operation. Integer divide by 2 is used instead if one needs to be assured of a defined result for the most significant bits.

Program 7.2, *Mask*, is a useful tool, as it provides a mask of any size in any position.

For an example of how masks are used, see Program 7.3, *Copy Bits*.

```
00 #include <stdio.h>
01
02 unsigned mask(short p, short n) {
03 // Purpose: return mask with 1 in positions
04 // p through p+n-1 and 0 elsewhere
05 // Preconditions: 0 <= p and 0 <= n and p + n < b
06 // where b is the number of bits in type unsigned.
07   return (~((~0) << n)) << p;
08 }
09 int main() {                                          Output:
10   printf("%08x\n", mask(11, 9));
11 }                                                     000ff800
```

Figure 7.2: *Mask*: returns mask of any size in any position.

7.10 Exercises

7.10.1 Nested less-than

In each row of Table 7.5, fill in the value of `x < y < z`.

x	y	z	x < y < z
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Table 7.5: Enter the values in the last column.

7.10.2 Converting between upper and lower case

Change Program 4.2 so that each input character is changed to be output in the opposite case.

7.10.3 Character packing

Write a program that inputs characters, packs them four to an integer (assume a one-byte `char` and a four-byte `int`), and outputs the resulting integers.

```
00 #include <stdio.h>
01 #include <assert.h>
02
03 unsigned mask(short p, short n);
04 void copyBits(short n, short p, unsigned x,
05               short q, unsigned* y) {
06 // Purpose: Copy n bits starting at position p in
07 // word x to position q in word y.
08 // Preconditions: 0 <= p,q,n,p+n-1,q+n-1 < b,
09 // where b is the number of bits in type unsigned.
10   unsigned rna = mask(p, n) & x;
11   // rna has bits p..p+n-1 from x and zero elsewhere.
12   // Shift rna to start nonzero bits at position q:
13   while (p<q) { rna *= 2; p++; }
14   while (p>q) { rna /= 2; p--; }
15   assert(p == q);
16   // rna shifted to start nonzero bits at position q
17   *y = *y & ~mask(q,n);
18   // recipient bit positions cleared
19   *y = *y | rna; // bits transferred to y
20 }
21 int main() {
22   unsigned x = 0x31acabbb, y = 0x9ac7b2bb;
23   copyBits(5, 7, x, 13, &y);                Output:
24   printf("%08x\n", y);
25 }                                           9ac6f2bb
```

Figure 7.3: *Copy Bits*: copy specified block of bits.

7.10.4 Unpacking characters

Write a program that inputs integers, which are assumed to contain four characters. It unpacks them, and outputs the resulting characters.

7.10.5 IP addresses

IP addresses are bit vectors of length 32. To make these easier to handle by humans, they are represented in *dotted decimal notation*: each group of four is written separately as an unsigned decimal literal. For example:

Bit vector	Equivalent dotted decimal
0x81340600	129.52.6.0
0xc0053003	192.5.48.3
0x0c020025	10.2.0.37
0x8080ff00	128.128.255.0

Write a program that inputs an `int` (assume it's four bytes) and outputs the equivalent dotted decimal.

7.10.6 Weight of bit vector

Write a program that prompts the user to input an integer and that outputs the number of `1`'s in the bit vector representing this integer.

7.10.7 Weight of bit vector?

What does the following program do?

```
#include <stdio.h>

int main() {
  int wt = 0;
  int i; scanf("%d\n", &i);
  while (i) { wt = wt + i%2; i = i/2; }
  printf("%d\n", wt);
}
```

Discuss its merits as a solution for problem 7.10.6.

7.10.8 Mystery program

What does the following program do?

```
#include <stdio.h>

int main() {
  int msk = 1, wt = 0, n = 0;
  int i; scanf("%d\n", &i);
  while (n < 32) {
    wt = wt + (i&msk);
    msk = msk << 1; n++;
  }
  printf("%d\n", wt);
}
```

7.10.9 Steganography

Encryption allows privacy in communication. It has the disadvantage that the existence of the message is in the clear. Steganography is an attempt at hiding the very existence of the message.

Digital images are an obvious medium for steganography. In images stored according to the 32-bit True Color format, each pixel is represented as four consecutive

one-byte fields. These are used for the intensities (starting with the least significant byte) for blue, green, and red, followed by the Alpha byte, which represents transparency.

Write a program that tests part of the code for a steganography method that degrades a given image by storing a message in the least significant halves of the fields of the pixels. The program is given an `int` p that stores a 32-bit pixel description and a `short` m with a size of 16 bits that contains part of a message. The program places each of the successive four four-bit fields of m in the least significant bits of the four fields of p, both in the order from less to more significant.

7.10.10 Circular shift

The type of shift effected by the operators `<<` and `>>` can be called "linear". In this type of shift, bits disappear off one end, while a fixed choice of bit appears at the other end. In a one-bit *circular* shift the bit that disappears at one end appears at the opposite end.

Write a program that contains code that effects a one-bit circular shift where, except for the most significant bit, each bit moves to the position of the next more significant bit.

7.10.11 Endianity conversion

There are two ways to store numbers in a computer memory: the least significant byte can have the lowest memory address or it can have the highest. If the processor assumes the former alternative, it is said to be *little-endian* (little end first); otherwise *big-endian* (big end first). The *endianity* of the processor indicates which is the case.

For example, the unsigned `0xbaddecaf` can be stored as

	little-endian	big-endian
address	content	content
12345	af	ba
12346	ec	dd
12347	dd	ec
12348	ba	af

Write a program that changes the endianity of an arbitrary (assumed four-byte) `unsigned` input.

7.10.12 OCR

In Program 7.1 most of the work is to ensure that -1 is returned for an invalid input. This entails corresponding effort in testing the function. Write and test a program that tests the function in Program 7.1 on all 128 possible inputs. If you have never written *for*-loops nested to a depth of seven, this is your opportunity.

Chapter 8

Control

Though not without limitations, computers are potentially enormously powerful. This power derives to a large extent from the fact that the programs that control them are typically more than mere to-do lists. Statements are often not executed in the order they are written. Depending on conditions, they may be skipped or repeated or retrieved from function bodies. The general topic of what gets executed or evaluated next is called *control*. The following sections treat the various aspects of control.

8.1 Compound statements

Long, unstructured lists of simple statements can usually be improved by identifying a small set of them that are related by having a common purpose and by using variables not needed elsewhere. The improvement then consists in making these statements into a *compound statement*, also called *block*. It usually has the form

{ *declarations statements* }

although declarations can also occur after one or more statements. The *declarations* create variables that only occur in *statements*. In this way we take a step towards *locality*, one of the goals to be aimed at in structuring a program. This goal is to create variables shortly before they are first used and to make them disappear as soon as they are no longer needed.

Suppose for example that we introduce a variable for the sole purpose of exchanging the values of two other variables, as in:

```
{ int x, y, temp;
    ...
    temp = x; x = y; y = temp;
    ...
}
```

In such a situation we can improve program structure by writing instead:

```
{ int x, y;
    ...
    { int temp = x; x = y; y = temp; }
    ...
}
```

In this way, `temp` only exists where needed.

8.2 Two-way decisions

As we saw in Section 3.4, during execution of a program a decision can be made according to the value of a condition being true or false. In an if-else statement a choice is made between two statements. Each of these can be any kind of statement, including an if-else statement. Thus such statements can be nested to any depth.

The need for such complex decisions often arises in practice. It requires considerable care to ensure that the code executes according to the specification. Several formalisms have been developed for writing such specifications. These include *decision tables*, *decision trees*, and *flowcharts*. Because they often get so complex that they are difficult to translate to code, software exists to automate this translation.

If-statements and if-else statements are so easy to get wrong that it is important to keep them simple. As soon as they are not, then one should use some form of table or tree to specify the decisions and translate these systematically to executable statements.

In Section 8.2.1 we saw a potential trap to be watched out for with nested `if`s and `else`s. In a statement of the form

```
if (C1) if (C2) S1a else S1b
```

we have to know which of the `if`s matches the `else`. One of the ways this can be answered is by means of a decision table:

C1	C2	action
1	1	S1a
1	0	S1b
0	1	nil
0	0	nil

Here `1` stands for *true* and `0` stands for *false*. When `C1` is *false*, the execution of both `S1a` and `S1b` is skipped and the entire statement has no effect. This is indicated by the action `nil`.

If we want the `else` to match the first `if`, then we write

```
if (C1) {if (C2) S1a} else S1b
```

This statement has as decision table

C1	C2	action
1	1	S1a
1	0	nil
0	1	S1b
0	0	S1b

8.2.1 The counterfeit coin

In this section we write a program that solves the problem of the counterfeit coin, one that I will explain presently. The reason for bringing the problem up now is that it demonstrates a more complex kind of selection than what we have seen so far.

Recall that in

`if (`C`)` S

S can be any statement whatsoever. For example, it can be an if-else statement. This way you get the structure:

`if (`C_0`) if (`C_1`)` S_{01} `else` S_1

The else goes with the second if. But if we want the else to go with the first if, then we need to force this by using braces:

`if (`C_0`) {if (`C_1`)` S_1`}` `else` S_0

Background *Three-sort* is so simple that one can hardly go wrong in the choice of comparisons to make and the order in which to make them. When sorting large sequences it matters a great deal at each stage what comparison to make. A policy that leads to good choices can result in vast performance differences.

Another example where efficiency depends on a good choice of comparison to make is the problem of the counterfeit coin. The problem is the following.

> *Suppose you have nine coins that all look the same. All but one are genuine, and have equal weight. The counterfeit coin is lighter. A balance is available with two trays. All it can do is compare the weights of the contents of the two trays. The comparison can have three outcomes: equal weights, left tray lighter, or right tray lighter.*
>
> *Find a sequence of weighings that identifies the counterfeit coin.*

Specification The program reads in nine numbers. All are equal, except one, which is smaller. The program prints one of the words 'one', ..., 'nine' to indicate which of the nine input numbers is the small one. The only allowable operation is to compare sums of non-overlapping subsets of the nine numbers.

Method Weighings should be arranged so that each divides the number of possibilities as much as possible into two equal halves. This rules out comparing single coins. At the other extreme, putting four coins on each tray leaves too many possibilities in the likely event that their weights are unequal.

Putting three coins on each tray strikes a good balance between these extremes. In the case of equal weights, the remaining three are identified as harbouring the culprit. Otherwise, we again have narrowed down the counterfeit coin to the set of three on the lighter tray. Either way, the second weighing gives the answer.

As each weighing has three possible outcomes, and as we need to identify one out of nine possibilities, we should not expect to be able to solve the problem with fewer than two weighings.

Pseudo code The weighing plan described above translates to the following nested selection statement.

if $w_1 + w_2 + w_3 < w_4 + w_5 + w_6$
  (w_1 or w_2 or w_3 lighter)
else if $w_1 + w_2 + w_3 > w_4 + w_5 + w_6$
  (w_4 or w_5 or w_6 lighter)
else
  (w_7 or w_8 or w_9 lighter)

The comments indicate the place for a similar nested statement for the second weighing. Whatever the outcome, no more than two conditions are evaluated.

Implementation See Program 8.1, *Counterfeit Coin*.

Running the program Suppose the coins are Kruger Rands, with a true weight of one ounce. Suppose the third coin is underweight at 0.95 oz. That suggests the following interaction:

```
Input nine weights; all equal except one, which is less:
   1.0 1.0 0.95 1.0 1.0 1.0 1.0 1.0 1.0
The rank of the light weight: three
```

8.3 Multi-way decisions

In Table 8.1 you see a common form of multi-way decision. The only way to implement such a table seems to be by splitting the multi-way decision into a chain of binary decisions, as shown in Program 8.2.

The following table could be translated in the same way. But because the decision is on a single value rather than on a range of values, we can use the *switch statement*, which is designed for such simple multi-way decisions. With a switch statement, the table could be translated as shown in Program 8.3: *Celebrities*.

In general, a switch statement has the form

```
00 #include <stdio.h>
01
02 int main() {
03   printf("Input nine weights; all equal except one, ");
04   printf("which is less:\n");
05   double w1, w2, w3, w4, w5, w6, w7, w8, w9;
06   scanf("%lf %lf %lf %lf %lf %lf %lf %lf %lf",
07          &w1,&w2,&w3,&w4,&w5,&w6,&w7,&w8,&w9);
08
09   printf("The rank of the light weight: ");
10   if ((w1+w2+w3) < (w4+w5+w6)) {
11      //w1 or w2 or w3
12      if (w1 < w2) printf("one");
13      else if (w1 > w2) printf("two");
14      else printf("three");
15   } else if ((w1+w2+w3) > (w4+w5+w6)) {
16      //w4 or w5 or w6
17      if (w4 < w5) printf("four");
18      else if (w4 > w5) printf("five");
19      else printf("six");
20   } else // w7 or w8 or w9
21      if (w7 < w8) printf("seven");
22      else if (w7 > w8) printf("eight");
23      else printf("nine");
24   printf("\n");
25 }
```

Figure 8.1: *Counterfeit Coin*: perform two tests and identify the underweight one among nine coins.

income x			*tax on income x*
$0 \leq$	x	≤ 37178	$0.155x$
$37178 <$	x	≤ 74357	$5763 + 0.22(x - 37178)$
$74357 <$	x	≤ 120887	$13942 + 0.26(x - 74357)$
$120887 <$	x		$26040 + 0.29(x - 120887)$

Table 8.1: A table for income tax.

`switch (` *expression* `)` *statement*

Here, *statement* generally has the form of a compound statement in the form of a list of *labeled statements*. In a switch statement, the labeled statement has one of

```
0 double tax(double x) {
1 // Purpose: return income tax according to table.
2 // Preconditions: x is non-negative.
3
4   if (x <= 37718) return 0.155*x;
5   else if (x <= 74357) return 5763  + 0.22*(x-37178);
6   else if (x <= 120887) return 13942 + 0.26*(x-74357);
7   else return 26040 + 0.29*(x-120887);
8 }
```

Figure 8.2: *Tax Time*: a function that computes income tax according to Table 8.1.

extension	*name*
5764	Colmerauer
5723	Kay
5727	Landin
5768	McCarthy
5719	Milner

Table 8.2: Telephone extensions of selected celebrities.

the following forms:

`case` *constant-expression* `:` *statements*
`default :` *statements*

The default statement, if any, occurs once and after all the case statements.

The switch statement is executed by evaluating *expression* and then executing *statement* starting at the first case statement, if any, where the value of the case equals that of the expression just evaluated. If the value of this expression is not equal to any of the *constant-expressions* in the case statements, then the default statement is executed, if present. If no default statement is present, then the switch statement is null.

The `break` statement causes control to be transferred to the statement following the switch statement. This statement is essential if one wants the switch statement to execute exactly one of the cases. In the absence of break statements the selected statement as well as all statements following in the switch statement are executed.

It is a common mistake to omit `break` statements. But their absence can be useful, as the following example shows.

8.3.1 Example: counting characters, words, and lines

Background An often-needed utility is the one that reports the numbers of characters, words, and lines in a text file. A space can separate two words, and so can

```
00 #include <stdio.h>
01
02 void listing(int extension) {
03 // Purpose: looks up and print according to given table.
04 // Preconditions: none.
05   switch (extension) {
06     case 5764: printf("Colmerauer\n"); break;
07     case 5723: printf("Kay\n"); break;
08     case 5727: printf("Landin\n"); break;
09     case 5768: printf("McCarthy\n"); break;
10     case 5719: printf("Milner\n"); break;
11     default:   printf("No Listing\n");
12   }
13 }
14 int main() {
15   int extension; scanf("%d", &extension);
16   listing(extension);
17 }
```

Figure 8.3: *Celebrities*, a program to translate telephone extensions to names of owners.

a tab character (`\t`). However, to count the number of word separators one cannot just count the number of space and tab characters in the file, as any contiguous sequence of space or tab characters also acts as a single word separator.

On this most utilities agree. There is a well-known one that counts every single new-line character (`\n`) as a line separator. Apparently it regards a line followed by a sequence of ten (`\n`)'s as containing ten lines, of which nine are empty. This example is how to write a function that computes the number characters, words, and lines in a text file, while correctly avoiding to count empty lines.

Specification It is required to write a function that reads a file as standard input and prints the number of characters, words, and lines in the file. A line is defined as a sequence of at least one character that does not contain a new-line symbol and cannot be extended without losing this property. Similarly, a word is defined as a sequence of at least one character that does not contain a space, a tab, or a new-line symbol and cannot be extended without losing this property.

Method We use a *finite-state automaton* as starting point. This is a device that receives inputs and performs actions on the basis of these inputs. The device is not merely a translator from each of a set of inputs to the corresponding action. For a given input, the resulting action depends on the *state* the device is in. That state depends on past inputs.

A particularly simple example is a toggle switch. When you press the button, it can have the effect of the light going on or going off. Which of these happens depends on the state it is in (being off or on). A vending machine is another example of a machine with state. Pushing the dispense button may or may not have the desired effect, depending on the state the machine is in.

The program that counts lines, words and characters can be built as a finite-state automaton with the characters of the file as inputs. The action taken as a result of receiving a word character as input has to depend on the state it is in: sometimes the word counter is incremented, but not every time. Roughly speaking, the action to be taken by the counting program depends not only on the character being read, but also on whether the character is in a word, is between words, or is between lines. Let us name these states and define them.

- The `red` state is the one in which the last character read is the new-line symbol `'\n'`. We are also in this state if we attempted to read a character and none was found.
- The `amber` state is the one in which the last character read is part of a line, but not of a word.
- The `green` state is the one in which the last character read is part of a word.

Pseudo code Instead of pseudo code in the usual format, we give the program's skeleton in the form of the following transition table for the finite-state machine. The table specifies for each state the action to be taken, depending on the last character read.

current state	*current character*	*next state*
red	`'\n'`	red
red	`' '` or `'\t'`	amber
red	word character	green
amber	`'\n'`	red
amber	`' '` or `'\t'`	amber
amber	word character	green
green	`'\n'`	red
green	`' '` or `'\t'`	amber
green	word character	green

Implementation To translate this table to code means to decide which line applies, depending on the state and the last character. If we have the state as the value of a variable, it is natural to use a switch statement on that variable. Within this state, a multi-way decision needs to be made on the last character read. Again, a natural for a switch statement. This suggests a nested switch statement.

However, here we have another new wrinkle: more characters than one require the same action. The fact that there are many word characters suggests that these are handled by a default statement. What do we do about the fact that both space and tab require the same action? We could of course ignore this commonality and

give each of these possibilities its own complete case statement. Notice how, in Program 8.5, this is more succinctly solved by merely omitting some `break` statements. Every call of the function `int getch(int *nc)` returns the next character read from the input, or `EOF` to indicate the end of input. In the first case it increments its actual parameter variable to indicate that a character has been read.

```
#include <stdio.h>

int getch(int *nc);
// Purpose: if standard input is at end of file, return EOF;
// otherwise, increment nc.
void linWrdChar( int *nl   // number of lines
               , int *nw   // number of words
               , int *nc); // number of characters
// Purpose: count number of lines, words, and characters
// from standard input.
// Preconditions: None.

int main() {
  int nl = 0, nw = 0, nc = 0;
  linWrdChar(&nl, &nw, &nc);
  printf("%d %d %d\n", nl, nw, nc);
}
```

Figure 8.4: *Lines, Words, and Characters*, a program to analyze a text file. See Program 8.5 for definitions of the auxiliary functions.

A more extreme example is in Program 8.6: *V-D-O*. With input

```
abc1dfegh2jkilm3npoqr4stuvw5xy
```

it gives as output

```
voodoovoodoovoodoovoodoovoodoo
```

8.4 Iteration statements

Iteration statements cause code to be repeatedly executed under control of a condition. There are three forms of iteration statement: the *while* statement, the *do* ... *while* statement, and the *for* statement.

The first of these is sufficiently familiar.

The second can be characterized by the fact that

`do` *statement* `while (`*expr*`);`

is equivalent to

statement `while (`*expr*`)` *statement*

The *do* ... *while* can be replaced by the *while*. The former is preferred when one wants to document that the controlled statement is executed at least once. In that case the *while* statement performs an unnecessary test.

The general form of the *for* statement is

`for (`$expr_1$ `;` $expr_2$ `;` $expr_3$ `)` *statement*

It is equivalent to

```
{ expr1 ;
  while (expr2 ) {
        statement
        expr3 ;
  }
}
```

Not all parts of the for statement need occur. Suppose we want to replace positive `p` and `q` by their greatest common denominator. This can be done by

```
for(; p != q;)
  if (p<q) q -= p;
  else
  if (q<p) p -= q;
```

There is no initialization because we assume that `p` and `q` already have their right values. There is no increment or decrement, as the required changes to `p` and `q` are not naturally expressed this way.

As an extreme example,

```
while ((c = getchar()) != EOF) putchar(c);
```

is sometimes written as:

```
for (;;) {
  if ((c = getchar()) == EOF) break;
  putchar(c);
}
```

```
#include <stdio.h>
typedef enum {red, amber, green} State;

int getch(int *nc) {
// Purpose: if standard input is at end of file, return EOF;
// otherwise, increment nc.
  int c = getc(stdin);
  if (c != EOF) (*nc)++;
  return c;
}
void linWrdChar(int *nl, int *nw, int *nc) {
// Purpose: count number of lines, words, and characters
// from standard input.
// Preconditions: None.
  State s = red; int c;
  while ((c = getch(nc)) != EOF) {
    switch (s) {
      case red:
        switch (c) {
          case '\n': break;
          case '\t': case ' ': (*nl)++; s = amber; break;
          default: (*nl)++; (*nw)++; s = green;
        } break;
      case amber:
        switch (c) {
          case '\n': s = red; break;
          case '\t': case ' ': break;
          default: (*nw)++; s = green;
        } break;
      case green:
        switch (c) {
          case '\n': s = red; break;
          case '\t': case ' ': s = amber; break;
} } }   }
```

Figure 8.5: *Finite-State Automaton*, a function to count the number of lines, words, and characters in a text file.

```
#include <stdio.h>
void processVowel(char ch) { printf("v"); }
void processDigit(char ch) { printf("d"); }
void processOther(char ch) { printf("o"); }
int main() {
  int ch;
  while ((ch = getchar()) != EOF) {
    switch (ch) {
      case 'a': case 'e': case 'i': case 'o': case 'u':
        processVowel(ch); break;
      case '0': case '1': case '2': case '3': case '4':
      case '5': case '6': case '7': case '8': case '9':
        processDigit(ch); break;
      default: processOther(ch);
  } }
  printf("\n");
}
```

Figure 8.6: *V-D-O*, a program to illustrate stacked-up cases in a switch statement.

8.5 Greatest common divisor

Background One of the oldest algorithms is the one by Euclid for computing the greatest common divisor (GCD) of two positive integers. It is a remarkable algorithm. At first sight it may seem to be necessary to find the prime factors of p and q. When we know these, we can see which are in common, and their product is $gcd(p, q)$. We will see that the GCD can be found without ferreting out any prime factors; even without performing any divisions.

Examples For any x and y we have that $gcd(x, y) = gcd(y, x)$; for any x it is true that $gcd(x, x) = x$. Pick arbitrary x and y and you have a good chance that that their GCD is 1; e.g. $gcd(1001, 10001) = 1$. But sometimes a surprisingly large common divisor is hidden, as the pair (1001, 637).

Specification Input: two positive integers x and y. Output: $gcd(x, y)$.

Method If $x = y$, then nothing needs to be done to solve the problem. Otherwise one of x and y is greater; suppose it is x. We reduce the problem to one that is easier to solve by using the fact that $gcd(x, y) = gcd(x - y, y)$. It is easier to find $gcd(x - y, y)$ when the greater of $x - y$ and y is less than the greater of x and y.

To compute $gcd(x, y)$, the GCD of x and y, we keep subtracting the smaller one from the larger until the two are equal. If x and y are equal already, then we don't have anything to do.

For example, the algorithm determines $gcd(34, 26) = 2$ by the steps shown in Table 8.3. The algorithm takes seven steps to complete. Every column shows one of these steps. In every column the smallest number of the previous column is repeated and the largest number of the previous column is replaced by the difference of the two.

step #	0	1	2	3	4	5	6	7
x	34	8	8	8	8	6	4	2
y	26	26	18	10	2	2	2	2

Table 8.3: Trace of Euclid's algorithm to find the gcd of 34 and 26.

Pseudo code

```
while x ≠ y
      if x < y then y ← y − x
      else x ← x − y
print x
```

Implementation See Program 18.2, *Iterative Euclid*.

```
0 int gcd(int x, int y) {
1 // Purpose: return the greatest common denominator of x and y.
2 // Preconditions: x and y are positive integers.
3   while (x != y)
4     if (x < y) y = y-x; else x = x-y;
5   return x;
6 }
```

Figure 8.7: *Iterative Euclid*: compute the greatest common denominator.

8.6 Example: Pythagorean triples

It is common for loops to be nested. For example the statement controlled by a `for`-statement can be a `for`-statement. Suppose we want to find Pythagorean triples. These are triples of natural numbers such that the square of third is the sum of the squares of the first two. The most familiar example consists of 3, 4 and 5. If we construct a triangle with these numbers as the lengths of the sides, then the triangle has a right angle, so that the theorem of Pythagoras applies. Hence the name.

As an example, consider a function that takes as input a positive integer N and prints all triples of positive integers x, y, and z such that $x^2 + y^2 = z^2$ and $z < N$. For $N = 20$ we should get the triples $\langle 3, 4, 5\rangle$, $\langle 6, 8, 10\rangle$, $\langle 5, 12, 13\rangle$, $\langle 9, 12, 15\rangle$, and $\langle 8, 15, 17\rangle$.

A brute-force way of find such numbers is to try all triples up to a certain limit. A natural way to do this is to use a `for` statement to let z run through all candidate values, for each of these let x and y run through all candidate values. That is, a `for` statement for y inside one for x and those two inside the one for z. See Program 8.8: *Pythagoras*.

```
00 #include <stdio.h>
01
02 void triples(int N) {
03 // Purpose: print all Pythagorean triples <x,y,z> with z < N.
04 // Preconditions: N is positive.
05   for(int z = 1; z < N; z++)
06     for(int x = 1; x < z; x++)
07       for(int y = 1; y < x; y++)
08         if (x*x + y*y == z*z)
09           printf("%d^2 + %d^2 = %d^2\n", x, y, z);
10 }
11 int main() {
12   printf("Input a positive integer.\n");
13   int N; scanf("%d", &N);
14   triples(N);
15 }
```

Figure 8.8: *Pythagoras*, a program that prints Pythagorean triples.

8.7 The comma operator

If E_1 is an expression and if E_2 is an assignment expression, then E_1, E_2 is an expression. The operator of this expression is the *comma operator*.

The value of the expression is the result of evaluating E_2 after the evaluation of E_1 is completed. This may be puzzling, as the value of E_1 is discarded. Yet the comma operator has its uses.

The comma operator can be useful in a situation where we want to execute two statements but there is only place for one. This happens for example in `for` statements:

```
void reverse(int a[], int n) {
  int p, q;
  for (p = 0, q = n-1; p < q; p++, q--) {
    int temp = a[p]; a[p] = a[q]; a[q] = temp;
  }
}
```

In many situations semicolons separating statements can be replaced by commas. This can be used as a way to emphasize that the statements belong together. In this way we would rewrite the swap of values of two variables from

```
temp = x; x = y; y = temp;
```

to

```
temp = x, x = y, y = temp;
```

As a final example of the comma operator, consider the function `main` in Program 8.5. In a more realistic context, the variables `nc`, `nw`, and `nl` are global and are used to communicate between different functions. Then we cannot be assured that they have been correctly initialized when `linWrdChar` is called. The comma operator allows the initialization to occur in the call:

```
linWrdChar((nl=0,&nl), (nw=0,&nw), (nc=0,&nc));
```

The commas separating the parameters have nothing to do with the comma operator.

8.8 The function as control mechanism

Background As the annual revolution of Earth around the sun takes 365.2422 days on average, any system of time reckoning has to alternate 365 (for "normal" years) with 366 (for leap years) as the number of days in a year. About twenty centuries ago, Julius Caesar introduced what is now known as the "Julian calendar", which determines the length of years as follows. Normal years are 365 days. Whenever the year number is divisible by four, there is a leap year. This gives an average length of year of 365.25 days, which is too long by

$$365.25 - 365.2422 = 0.0078,$$

which is about three quarters of a day per century.

The discrepancy was corrected by the Gregorian calendar, first introduced in certain jurisdictions in the sixteenth century. Now it is used worldwide. The Gregorian calendar follows the Julian calendar except in century years. In the Gregorian calendar, normal century years have 365 days. The exceptions are leap-century years, which occur when the year number is divisible by 400.

This gives a cycle of 400 years with a total of 146,097 days, making the average length of the year over a cycle 365.2425 days. As the average length of the year is 365.2422, this leaves a discrepancy of 3 days in 10,000 years.

Specification The program reads an integer equal to a year number and prints its number of days according to the Gregorian calendar.

Method As we have to test the year number for three conditions, we need a nested if-else statement. In

`if` (*C*) S_0 `else` S_1

both S_0 and S_1 can be selection statements. Such substitutions can be made to any depth. The resulting constructs may become difficult to understand. *A good strategy for complex decisions is to keep S*0 *a simple statement and only replace* S_1

by a selection statement. Here we have a complex decision: leap years are exceptions to normal years, normal century years are an exception to that exception, while every fourth century year is an exception to an exception to an exception. Can we keep this simple?

This question translates to: Is there a single condition that tells us the length of some year? This is not the case if we only know that the year number is divisible by four or by a hundred. But if we ask whether it is divisible by 400, then we know the year length, if the answer is Yes. Otherwise, we know that we do not have a leap-century year, and we have a less complex problem. In this latter case there is again a single condition with the same favourable property.

Pseudo-code

input year number y
if y is divisible by 400 then print 366 and halt
  (y is not divisible by 400)
if y is divisible by 100 then print 365 and halt
  (y is not divisible by 100)
if y is divisible by 4 then print 366 and halt
  (y is not divisible by 4)
print 365 and halt

Implementation See Program 8.9, *Leap Year*.

```
int yearLen(int y) {
  if ((y % 400) == 0) return 366;
  if ((y % 100) == 0) return 365;
  if ((y % 4) == 0) return 366;
  return 365;
}
```

Figure 8.9: *Leap Year*, a function to compute the length of any year with non-negative number y.

8.9 Jump statements

Selection statements affect control flow because they are replaced by one of their constituent statements under control of a condition. Iteration statements affect control flow because they are replaced by a repetition, under control of a condition, of suitably modified instances of their body. In neither case is there an explicit transfer of control.

To transfer control explicitly, C has four *jump statements*: `return`, `break`, `continue`, and `goto`.

8.9.1 Return

We already encountered the `return` as soon as we used functions. What still needs to be done is to point out a common weakness in code resulting from not realizing the full power of this statement. This weakness is exemplified by the first version in Section 8.8. This function is more clearly and more succinctly written as in the second version in that section.

In many functions this is an effective strategy: Isolate the simplest part of the function's task, do it, and get out. Repeat with the remaining part of the task.

8.9.2 Break

We already encountered the `break` in connection with the switch statement. The effect of `break` is to transfer control to immediately past the closest surrounding switch or iteration statement.

In an iteration statement, the condition normally determines when the loop terminates. Often, however, there is more than one such condition. Consider for example a program that copies a line:

```
void copyLine() {
  char c;
  while ((c = getchar()) != EOF && c != '\n')
    putchar(c);
  putchar('\n');
}
```

Here it is essential that what's on the left of the `&&` gets evaluated before the part on the right. This is only so because of an exception, "short-circuit condition evaluation", to the general rule that the order in which the subexpressions of an expression are evaluated is not defined.

It is worth knowing another approach to the problem of terminating a loop on multiple conditions. This other approach uses the `break` statement, and that is why it is introduced here.

```
void copyLine() {
  char c;
  while ((c = getchar()) != EOF) {
    if (c == '\n') break;
    putchar(c);
  }
  putchar('\n');
}
```

In this function it is clearer than before that the comparison with `'\n'` comes after the test for `EOF`.

8.9.3 Continue

The next jump statement is similar to `break` in that it modifies the behaviour of an iteration statement. It is called `continue`. Its effect is to transfer control to the end of the body of the iteration statement. Thus it skips only to the end of the current iteration without terminating the iteration statement, which is what `break` does. Here is an example

```
float addFile() {
  double x, sum = 0.0;
  while (scanf("%lf", &x) == 1) {
    if (x <= 0) continue; // only add positive numbers
    sum += x;
  }
  return sum;
}
```

You will see `continue` used less often than `break`, as the same effect can be more easily achieved by an if statement. An attraction of `continue` is that it avoids the need to indent the rest of the body of the iteration statement, which can be a long piece of code.

8.9.4 Goto

The final type of jump statement is the *goto statement*, which has the following form:

`goto` *identifier*`;`

This statement can occur if there is in the same function, outside of a switch statement, a labeled statement that has as label the one that is named by *identifier*.

The break- and continue statements can be viewed as specialized versions of the goto statement that transfer to a specific place. Because that place is determined by the location of the break- or continue statement, there does not need to be a labeled statement.

The above example of a break statement can be written with a `goto` as follows:

```
void copyLine() {
  char c;
  while ((c = getchar()) != EOF) {
    if (c == '\n') goto L;
    putchar(c);
  }
  L: putchar('\n');
}
```

For the goto equivalent of the above example of a continue statement, we only have to look at the while statement, which becomes:

```
float addFile() {
  double x, sum = 0.0;
  while (scanf("%lf", &x) == 1) {
    if (x <= 0) goto L; // only add positive numbers
    sum += x;
    L:;
  }
  return sum;
}
```

Notice that the statement labeled with `L` to which control transfers need not do anything. In fact, it *must* not do anything. Hence, what is labeled in

```
L:;
```

is the statement

```
;
```

This is the *empty statement*.

Clearly, goto statements can be used to implement the equivalent of iteration statements. But goto statements are not restricted this way. In fact, they are a versatile way of expressing any pattern of transfer of control. Some of these patterns have a name, such as *while*, *for*, and *switch*. There is no need to implement such a pattern with goto statements; use instead a while, a do-while, a for statement, or a switch.

8.10 Exercises

8.10.1 Counterfeit among ten coins

Modify Program 8.1 to detect a single underweight coin from among ten coins.

Chapter 9

Arrays and strings

There's more to arrays than what we saw of them in Chapter 3. Here I explain how to pass array data into and out of functions, the relations between arrays and pointers, and how to work with multi-dimensional arrays. Strings are a special kind of array, and so important that C accords them special status.

9.1 Arrays as sequences of variables

Consider the definition `int A[n]`. It defines an integer array of length n, a sequence `A[0]`, ..., `A[n-1]` of n integer variables stored contiguously in memory. Each of the n variables has an address. The address of `A[0]` is the address of the array `A`. This is not just figuratively speaking; it is literally the case: the array identifier `A` has the type of pointer-to-integer. Thus `A[1]`, `A[2]`, `A[3]`, are the same integer variables as `*(A+1)`, `*(A+2)`, and `*(A+3)`. By the same logic, `A[0]`, is the same integer variable as `*A`.

But, you may object, isn't it the case that the next address after a is $a + 1$? Assuming that integers occupy four bytes in memory, the address of `A[1]` should be `A+4` rather than `A+1`. The objection is understandable and shows that the situation is not quite as simple as just suggested, though more convenient for the programmer.

C allows for the possibility that integers have a size other than four bytes. The address calculation has to be adjusted accordingly. The address calculation does not need to be left to the programmer because the C compiler knows that `*A` has type pointer-to-integer and knows how many bytes there are to an integer on *this* machine. For that reason C defines that `*(A+n)` to be the n-th element of `A` independently of the size of the array elements.

Subtracting addresses that are far from each other can result in a large integer. If such a result is to be stored in a variable, what should its type be? Usually `unsigned` or even `int` suffices. But whether this is actually the case depends on the implementation of the C language. To make sure enough storage is allocated for such a variable, its type should be made `ptrdiff_t`, a type specifically introduced for this purpose.

```
00 #include <stdio.h>
01 #include <assert.h>
02
03 int main(){
04   int   a0[] = {0,31,28,31,30,31,30,31,31,30,31,30};
05   short a1[] = {0,31,28,31,30,31,30,31,31,30,31,30};
06   int len = sizeof(a0)/sizeof(a0[0]);
07   assert(&a1[len-1] - &a1[0] == len-1);
08
09   printf("%p  %p\n%p  %p\n",
10     (void*)&a0[0],(void*)&a0[1],
11     (void*)&a1[0],(void*)&a1[1]);
12 }
```

Output:

```
0x7fff56490b60  0x7fff56490b64
0x7fff56490b40  0x7fff56490b42
```

Figure 9.1: A program to illustrate array basics.

9.2 Arrays as function parameters

The programs so far were in a style that could be called "array-oriented". The fact that array elements can be accessed by means of pointers give rise to the "pointer-oriented" style of programming preferred by many programmers. As a result even those who prefer to write in the array-oriented style, read a lot of pointer-oriented code. See Program 9.2 *Array Copy* for both array-oriented and pointer-oriented versions.

You may wonder why we supply both the name of the array and its length as actual parameters to function calls that act on an array. Can't we determine that length by the expression

```
sizeof(a)/sizeof(a[0]).
```

This only works in the block where the array is defined. In the function body `sizeof(a)` returns the size of `a` as a pointer rather than as an array.

Both `copy0` and `copy1` have as purpose to copy `b[0..n-1]` into `a[0..n-1]`. Both have as precondition that `a[0..n-1]`, `b[0..n-1]`, and `c[0..n-1]` be allocated. To say that `a[p..q]` is allocated means that this sequence of variables is part of memory allocated to an array. They do not have to be all of an array. For example, in *Array Copy* we can add a function call

```
copy0(c+n-n/2, a, n/2);
```

to copy the first half of `a[0..n-1]` to the second half of `c[0..n-1]`.

```
00 #include <stdio.h>
01
02 void copy0(int a[], int b[], int n) {
03   for(int i = 0; i < n; i++) a[i] = b[i];
04 }
05 void copy1(int *a, int *b, int n) {
06   for(int i = 0; i < n; i++) *a++ = *b++;
07 }
08 int main() {
09   int a[] = {0,1,2,3,4,5,6,7,8,9},
10       n = sizeof(a)/sizeof(a[0]),
11       b[n], c[n]; // same size as a
12   int *ap = a, *bp = b, *cp = c;
13   copy0(b, a, n);
14   // copy0(bp, ap, n); // works as well
15   copy1(c, a, n);
16   // copy1(cp, ap, n); // works as well
17   for(int i = 0; i < n; i++)
18     printf("%d %d, ", i, b[i] == c[i]);
19   printf("\n");
20   copy0(c+n-n/2, a, n/2);
21   for(int i = 0; i < n; i++)
22     printf("%d %d, ", i, c[i]);
23   printf("\n");
24 }
```

Figure 9.2: *Array Copy*, array-oriented and pointer-oriented versions of a function that copies one array to another.

9.3 Strings

The reasons for having arrays of numbers, or arrays of arrays of numbers, is that we want to represent n-dimensional vectors as used in science and engineering, or rows and columns of financial data. A kind of data that occurs in all types of computer application is *text*. Text can be regarded as a array of characters. In programming such arrays are called "strings". C provides *string literals*, written as sequences of characters between double quotes:

```
"This is a string."
```

9.3.1 Character arrays versus strings

Why do we need "string" as a separate concept when arrays can have characters as elements? The answer is that, though strings are indeed such arrays, a distinction

needs to be made.

To understand the distinction, let us consider the two ways of specifying the length of a sequence: *extrinsic* and *intrinsic*. In the extrinsic method, there is an integer, independent of the sequence, that gives the length. In the intrinsic method, there is a designated value, the *sentinel*, that does not occur in the sequence. Accordingly, if we store the sequence in an array or file, then the first occurrence of the sentinel value can be used to signal the end of the sequence.

If we do not want to reserve a value for the exclusive use as a sentinel, then we need to use the extrinsic method. This is why, so far, our functions that act on arrays have had separate parameters for the array and for its length.

There are types of data where we don't need to allow for all possible values. The excluded values are candidates for playing the role of sentinel and we can use the intrinsic method for sequence representation. Examples:

- An important class of data are nonnegative numbers. In this case there are plenty of sentinel candidates. Conventionally, one uses -1.
- Memory addresses have plenty of excluded values, reflecting the fact that the operating system does not allow user programs to access all of memory. A null pointer is a pointer variable that has an excluded address as value. If we want to store a sequence of addresses, then it is natural to use a null pointer to indicate the end of the sequence.
- ASCII text has the property of not containing the character code 0. Accordingly, this code is used as sentinel to indicate the end of text. It should be noted that we don't get code 0 when writing the character literal `'0'`, because its ASCII code is 48, as we saw in Figure 4.3. The character literal for NUL, the ASCII name for the symbol with code 0, is `'\0'`. When discussing code we follow the common convention of referring to NUL as "the null character".

> *The character array* `s` *is a legal string if every character in* `s[0..n]` *is legally addressable, where* `s[n]` *is the leftmost null character. The length of the string is* `n`.

A difficulty with the intrinsic sequence representation chosen for strings is that, at the time the array is declared, the length of the string is often not known. Code that fills the array with a string has to watch simultaneously for the end of the array and for the end of the string. This is often not done correctly, so that string characters are written beyond the area allocated to the array. This is called "buffer overflow". Such code can fail with disastrous consequences after years of satisfactory performance for millions of users. Sometimes the failure is caused on purpose by a malicious user; this is called an "attack", or an "exploit".

The function `strcopy` in Program 9.3 demonstrates several features of string processing. The length of the string is not a parameter, as the code assumes that the string is correctly terminated by the null character. In this way the end of string `s` is detected. The function adds the null character to the copy being created. In function `main` we print `s` as an integer array, so we can see the terminating null

```
00 #include <stdio.h>
01
02 char* strcopy(char dest[], char src[]) {
03 // Purpose: return dest after copying contents of src to dest,
04 // including the terminating null character.
05 // Preconditions: src is a legal string of length n and
06 // dest[0..n-1] is allocated.
07   int i;
08   for(i=0; src[i] != '\0'; i++) dest[i] = src[i];
09   dest[i] = '\0';
10   return dest;
11 }
12 int main() {
13   char t[] = "abc";
14   const int n = 4; char s[n];
15   // One extra location for terminating null character.
16   strcopy(s, t);
17   for(int i=0; i<n; i++) printf("%d ", s[i]);
18   printf("\n%s\n", s);
19 }
```

Figure 9.3: A beginner's string copy function. The return value is conventional for a string copy function. It is often left unused.

character. It is also printed as a string, and then the null character does not appear.

As C programmers gain experience, they tend to write the body of `strcopy` in Program 9.3 more succinctly. By using pointers, the array index can be avoided. The body then becomes

```
for(; *t != '\0'; s++, t++) *s = *t;
*s = '\0';
```

By combining the assignment with the test, the body of the function becomes

```
for(; (*s = *t) != '\0'; s++, t++);
```

The body of the `for` statement has disappeared. Now the last assignment is the one in the test that causes the for statement to terminate. The null character terminating `s` is thereby copied from `t`.

As a further abbreviation we can include the increments into the assignment. In this way the body becomes

```
for(; (*s++ = *t++) != '\0';);
```

By now the for-statement has become quite degenerate: two of its three fields are empty. It looks better as a `while` statement:

```
while((*s++ = *t++) != '\0');
```

The condition `false` is coded as zero, which happens to be the code for the null character. This is exploited in the next simplification, which changes the body of the function to

```
while(*s++ = *t++);
```

This allows the function to be defined as:

```
void strcopy(char s[], char t[]) {while(*s++ = *t++);}
```

Copying a string is such a commonly required function that it occurs in the C standard library. It is accessed by including `string.h`. Though not officially defined like this, typical use can assume the declaration to be

```
char *strcpy(char* s, char* t);
```

The ordering of the parameters and the return type make it look like the assignment statement: the function copies from `t` to `s` and it returns a pointer to the first element of `s`. This function is one of many string functions available from the standard library.

9.3.2 Defense against infinite strings

Short and sweet as we may have been able to make the body of `strcopy`, the function relies on two crucial assumptions: that `t` is terminated by a null character and that `s` has enough memory allocated to it. It may have disastrous consequences to call `strcpy` when these conditions are not met. While the code cannot be made totally fool- and villain-proof, at least something can be done to make the function less fragile.

An array of characters that is not terminated by a null, can be thought of as an infinite string. Such a string can not be copied into any amount of memory allocated for the string being copied to. To ensure that copying a string does not cause any writing to memory that is not allocated for the purpose, an additional parameter can be added to the function.

This can be done in several ways. The library function `strncpy` includes an integer parameter that can be used as defense against attempting to copy an infinite string. Though not officially defined like this, typical use can assume the declaration to be

```
char *strncpy(char *s, char *t, int n);
```

writes `n` characters in `s`. This is independent of the number k of characters in `t` preceding the first null character. The function copies the first k characters of `t` into `s` followed by $n - k$ null characters if k is less than n. Otherwise, it copies the

first n characters of `t` into `s`, which does not in that case become a null-terminated string.

The function for comparing two strings likewise comes in two varieties: `strcmp` and `strncmp`. The same is true in the case of string concatenation, for which the standard library has `strcat` and `strncat`.

9.4 Multi-dimensional arrays

The elements of an array can themselves be arrays. We think of such an array as *two-dimensional*. See for example Program 9.4: *Row-major or column-major?* Here `A` has two elements, `A[0]` and `A[1]`, each of which is an array of three elements. Its first dimension is 2; its second dimension is 3. Accordingly, it is created by the definition `int A[2][3]`. The output of Program 9.4 is:

```
10 11 12
11 12 13

10 11 12 11 12 13
```

We can define a three-dimensional array; it has two-dimensional arrays as elements. And so on to any number of dimensions. Although there is no limit to this number, in practice one finds mostly one- and two-dimensional arrays.

```
00 #include <stdio.h>
01
02 int main() {
03   int A[2][3] = {{10,11,12},{11,12,13}}, *ap;
04   int n = sizeof(A)/sizeof(A[0][0]);
05   int i, j;
06   for(i=0; i<2; i++) {
07     for(j=0; j<3; j++) printf("%d ", A[i][j]);
08     printf("\n");
09   }                                              Output:
10   printf("\n");
11   for(i=0, ap=&A[0][0]; i<n; i++)                10 11 12
12     printf("%d ", *(ap++));                      11 12 13
13   printf("\n");
14 }                                                10 11 12 11 12 13
```

Figure 9.4: *Row-major or Column-major?*

Arrays, of whatever dimension, are a fiction maintained by the programming language. The underlying reality is random-access memory, which is one-dimensional. As a result, when a two-dimensional array is stored, either rows or columns have

to be broken up. If rows are left intact, then the array is said to be stored in "row-major" order; otherwise, in "column-major order". In C, arrays are stored in row-major order. This is can also be expressed by saying that the second index varies most rapidly when scanning along the array elements in memory. For example, array `A` is stored as:

```
A[0][0] A[0][1] A[0][2] A[1][0] A[1][1] A[1][2]
```

The second loop in the Program 9.4 demonstrates this.

The initialization of the two-dimensional array in this program also has to be in row-major order:

```
int A[2][3] = {{10,11,12},{11,12,13}};
```

I started out by saying that `A` is an array of arrays: that `A[0]` is `{10,11,12}`, and `A[1]` is `{11,12,13}`. This is true only to a certain extent. Consider the code snippet

```
int A[2][3] = {{10,11,12},{11,12,13}};
printf("size of A: %d\n", sizeof(A)/sizeof(A[0][0]));
printf("size of A[0]: %d\n", sizeof(A[0])/sizeof(A[0][0]));
```

which causes as output

```
size of A: 6
size of A[0]: 3
```

The first line shows `A` as a single sequence consisting of the rows concatenated. According to the second line, `A[0]` does seem to exist and it even has the length of the rows of `A`.

Array `A` is only one way of storing six numbers arranged like

```
10  11  12
11  12  13
```

Let us compare `A` with an array that really is an array of arrays, as `B` is in the code snippet:

```
int A[2][3] = {{10,11,12},{11,12,13}};
int* B[2];
int B0[] = {10,11,12};
int B1[] = {11,12,13};
B[0] = (int*) B0;
B[1] = (int*) B1;
printf("B[1][2]: %d\n", *(*(B+1)+2));
printf("B[1][2]: %d\n", B[1][2]);
assert(&B[1][2] == *(B+1)+2);
```

The definition of `A` allocates space for six integers; that of `B` only allocates space for two pointers to integers. These pointers point to first elements of arrays. Obviously we can access the element in the second row and third column with `*(*(B+1)+2)`. But we can write equivalently `B[1][2]`! As `A` is a two-dimensional array of integers, the compiler translates `A[i][j]` to `*(A+i*r+j)`, where `r` is the length of the rows of `A`. As `B` is a one-dimensional array of pointers to integers, the compiler translates `B[i][j]` to `*(*(B+i)+j))`. This will turn out to be important later when we program with matrices.

In discussing definitions of one-dimensional arrays, I advocated the use of an incomplete array type so that the size of the array would only specified in a single place. Accordingly, I would advocate

```
int A[][] = {{10,11,12},{11,12,13},{12,13,14}};
```

as the size information is already available in this way. However, this is an error: the C language only allows the leftmost dimension to be left unspecified.

This restriction not only affects array definitions, but also array parameters of functions. To review the options we have in C, let us consider a function that adds the diagonal elements of square array. In this example, it would return

```
A[0][0] + A[1][1]+ A[2][2]
```

The declaration of such a function should be

```
double diagSum(double A[][], int n);
```

The requirement that all dimensions but the first be specified prevents this. We even have to put constants for the second dimension, as in

```
double diagSum(double A[][3], int n);
```

and then make sure that this function is only called with 3 as value for `n`. This goes against the basic idea of a function: to parametrize the code in body*.

This is why we should not represent a matrix by a two-dimensional array of numbers, but by a one-dimensional array of pointers to one-dimensional arrays of numbers that represent the rows of the matrix†. Thus we can define `diagSum` as in Program 9.5.

The function `diagSum` is as general as we want it: it will work for square matrices of any size `n`. We can't expect the array definition to be general, as every particular array has a particular size. So the array *definitions* have constants for their bounds. Still, it would be preferable to be able to use a variable to specify that the three occurrences of an array bound have to be the same: C does not allow

*The constant in the second dimension can be made symbolic by means of a macro (see Appendix D), as is done in Program 9.5. This does not solve the problem, as the function is still only usable with one size of array.

†*Numerical Recipes in C* by Brian P. Flannery, Saul A. Teukolsky, William T. Vetterling.

```
00 #include <stdio.h>
01 #define N 3
02
03 double diagSum(double** A, int n) {
04   double sum = 0.0;
05   for(int i=0; i<n; i++) sum += A[i][i];
06   return sum;
07 }
08
09 int main() {
10   double data[N][N] = {
11     {1,2,3},
12     {3,2,1},
13     {2,3,1}};
14   double* A[N];
15   for(int i=0; i<N; i++) A[i] = &data[i][0];
16   printf("trace: %lf\n", diagSum(A,N));
17 }
```

Figure 9.5: Two-dimensional array `A` as array of pointers to arrays.

```
const int N = 3;
double data[N][N] = {
  {1,2,3},
  {3,2,1},
  {2,3,1}};
double* A[N];
```

9.5 Exercises

9.5.1 Evaluating a polynomial

A expression of the form

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

is called a *polynomial* in x with coefficients $a_0, \ldots, a_{n-1}$. Write a function

`double evalPol(double a[], int n, double x);`

that evaluates the polynomial with coefficients in array `a` of length `n` for the value `x`.

An efficient algorithm is suggested by *Horner's Scheme*:

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} =$$
$$a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-2} + x a_{n-1}) \cdots)$$

9.5.2 Gaps

Given integer array `a` of length n containing nonnegative integers in increasing order. The *gaps* of such an array are all nonnegative integers less than `a[n-1]` that do not occur in `a`. Write the definition of `printGaps` that is missing in the program in Figure 9.6.

```
#include <stdio.h>

void printGaps(int a[], int n);
int main() {
  int a[] = {1,3,5,9,10,14,19,20};
  printGaps(a, sizeof(a)/sizeof(a[0]));
}
/* Output:
0 2 4 6 7 8 11 12 13 15 16 17 18
*/
```

Figure 9.6: A main program for function `printGaps`.

9.5.3 Purifying input

When entering a password, some software allows one to correct typing mistakes by backspacing over them. Others don't: they apparently record the backspace characters as part of the attempted password. This exercise is to see how difficult it is to implement correction of a typed string by means of backspaces. Consider Program 9.7. Function `capture` is the primitive version that captures keystrokes from standard input without doing anything special for backspaces. Function `pureCap` lets each backspace cancel the most recent uncanceled character. If a backspace is entered when are no (more) uncanceled characters, then it is ignored.

Here is a sample run of the program:

```
abc^H^H^H^Hdef
number of characters captured: 10
97 98 99 8 8 8 8 100 101 102
abc^H^H^H^Hdef
number of characters captured: 3
100 101 102
```

The lines `abc^H^H^H^Hdef` have been typed by the user when the program waits for input. Each `^H` represents a backspace character. Note that there are more backspaces than the preceding normal input. Accordingly, the last backspace is ignored by `pureCap`.

```
00 #include <stdio.h>
01
02 int capture(char arr[], int n) {
03 /* places characters from standard input into arr */
04   int c, i;
05   for(i=0; i<n; i++) {
06     if ((c = getchar()) == '\n') break;
07     arr[i] = c;
08   }
09   return i;
10 }
11 int pureCap(char arr[], int n);
12 int main() {
13   const int n = 200; char arr[n];
14   int i, j;
15   j = capture(arr, n);
16   printf("number of characters captured: %d\n", j);
17   for(i = 0; i < j; i++) printf("%d ", arr[i]);
18   printf("\n");
19   j = pureCap(arr, n);
20   printf("number of characters captured: %d\n", j);
21   for(i = 0; i < j; i++) printf("%d ", arr[i]);
22   printf("\n");
23 }
```

Figure 9.7: A function for capturing standard input.

9.5.4 Tallying

In the program below, add definitions of functions so that a call to `tally` prints the different elements of the given array followed by the number of different elements.

```
#include <stdio.h>
void tally(int a[], int n);
int main() {
  int a[] = {4,2,6,7,4,5,9,7,4,7,1,7,5,4,8,8};
  tally(a, sizeof(a)/sizeof(a[0]));
}
```

We the data shown the output should be:

```
1 2 4 5 6 7 8 9
8
```

9.5.5 Relative frequencies in a character file

Supply the function definition missing from the following program:

```
int freqDist(double freqs[], int n);
// Purpose: place the the relative frequencies of the printable
// characters of the text file in standard input in the array
// freqs of length n.
// Preconditions: n >= 0 and freqs[0..n-1] allocated.
int main() {
  double freqs[128];
  int sz = sizeof(freqs)/sizeof(freqs[0]);
  int i;
  printf("number of characters: %d\n", freqDist(freqs, sz));
  printf("character frequencies in percent:\n");
  for(i=0; i<sz; i++) {
    if (freqs[i] != 0)
      printf("%c:%1.2f ", (char)i, 100*freqs[i]);
  }
}
```

For example, for with the input

```
the quick brown fox jumps over the lazy dog
```

we get an unusually flat frequency distribution; the output is (some newlines inserted to improve readability):

```
number of characters: 44
character frequencies in percent:

:2.27  :18.18
a:2.27 b:2.27 c:2.27 d:2.27 e:6.82 f:2.27 g:2.27 h:4.55 i:2.27 j:2.27
k:2.27 l:2.27 m:2.27 n:2.27 o:9.09 p:2.27 q:2.27 r:4.55 s:2.27 t:4.55
u:4.55 v:2.27 w:2.27 x:2.27 y:2.27 z:2.27
```

The first frequency, 2.27, is that of the newline character, which is printed as an empty line. The second frequency is that of the space character, so it is printed as a ... space!

When the input is "Moby Dick", a big novel, the output is (some newlines inserted to improve readability):

```
number of characters: 1220151
character frequencies in percent:

:1.88  :16.24 !:0.14 ":0.25 $:0.00 &:0.00 ':0.24
(:0.02 ):0.02 *:0.00 ,:1.58 -:0.49 .:0.62
0:0.01 1:0.01 2:0.00 3:0.00 4:0.00 5:0.00 6:0.00 7:0.00 8:0.00 9:0.00
::0.02 ;:0.34 ?:0.08
A:0.22 B:0.12 C:0.09 D:0.06 E:0.10 F:0.07 G:0.05 H:0.12 I:0.29 J:0.02
K:0.01 L:0.07 M:0.06 N:0.10 O:0.08 P:0.09 Q:0.03 R:0.07 S:0.18 T:0.20
U:0.02 V:0.01 W:0.11 X:0.00 Y:0.03 Z:0.00
[:0.00 ]:0.00 _:0.00
a:6.17 b:1.27 c:1.75 d:3.07 e:9.50 f:1.64 g:1.65 h:5.04 i:5.07 j:0.07
k:0.65 l:3.43 m:1.85 n:5.28 o:5.60 p:1.33 q:0.10 r:4.21 s:5.08 t:7.01
u:2.17 v:0.69 w:1.71 x:0.08 y:1.36 z:0.05
```

The frequency of newlines suggests an average line length of about fifty; that of the spaces an average word length of around six. As in most English texts, the letter e is the most frequent one, although apparently not among the capital letters.

9.5.6 Cyclic shift

A cyclic shift of a sequence moves every element k places to the left; for the purpose of this operation, the rightmost element is considered to be to the left of the leftmost element. Here is an example with $k = 3$.

$$0\,1\,2\,3\,4\,5\,6\,7\,8\,9 \Rightarrow 3\,4\,5\,6\,7\,8\,9\,0\,1\,2$$

The function to perform cyclic shifting has as declaration:

```
void shift(int a[], int n, int k);
```

In general we define cyclic shifting as follows. If on entry the elements of `a` are $a_0, \ldots, a_{n-1}$, then on exit from the function they are $a_{i_0}, \ldots, a_{i_{n-1}}$, where $i_j = (j + k) \bmod n$ for $j = 0, \ldots, n - 1$.

For example, if on entry to the function $a_0, \ldots, a_{n-1}$ are $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$ and $k = 3$, then on exit they are $3, 4, 5, 6, 7, 8, 9, 0, 1, 2$.

One solution is as follows:

```
void shift(int a[], int n, int k) {
  int b[n], i;
  for(int j=0; j<n; j++)
    b[(i = j-k) < 0 ? i+n : i] = a[j];
  for(int j=0; j<n; j++) a[j] = b[j];
}
```

To make this work, we use n extra storage locations in the form of the array `b`. It is possible to do with fewer:

```
void shift(int a[], int n, int k) {
  while(k-- > 0) {
    int temp = a[0];
    for(int i=1; i<n; i++) a[i-1] = a[i];
    a[n-1] = temp;
  }
}
```

This needs only one extra storage location, but takes kn assignments. For most values of k, this is a lot more than the previous solution.

The following solution is better than both previous ones: no temporary storage to speak of, and fewer assignments for most values of k. This solution consists of applying a number of reversals to the array. Example:

$$\underbrace{0\,1\,2}_{\texttt{rev}}\underbrace{3\,4\,5\,6\,7\,8\,9}_{\texttt{rev}} \Rightarrow \underbrace{2\,1\,0\,9\,8\,7\,6\,5\,4\,3}_{\texttt{rev}} \Rightarrow 3\,4\,5\,6\,7\,8\,9\,0\,1\,2$$

Believe it or not, this works every time.

```
void rev(int a[], int i, int j) { /* reverses a[i]..a[j] */
  for(; i<j; i++, j--) {
    int temp = a[i]; a[i] = a[j]; a[j] = temp;
  }
}
void shift(int a[], int n, int k) {
  rev(a, 0, k-1); rev(a, k, n-1); rev(a, 0, n-1);
}
```

The number of swaps in the calls to `rev` is approximately, for large k and n, $k/2 + (n-k)/2 + n/2 = n$. Each of these swaps takes three assignments, so the shifting takes about $3n$ assignments, however large or small k is. Doing the reversing trick is therefore more efficient than the first two solutions.

After these preliminaries, it is time to state the exercise:

Implement `void shift(int a[], int n, int k);` *using as little extra storage as the reversing trick, but using a significantly smaller number of assignments.*

9.5.7 Adding line numbers

A surprising number of useful functions can be written as variants of the file copy program[‡]:

```
#include <stdio.h>

void fileCopy() {
  int ch;
  while ((ch = getchar()) != EOF) putchar(ch);
}
```

[‡] "Software Tools" by Brian Kernighan and P. J. Plauger. Addison-Wesley, 1976.

Use this function as starting point for a program that reads a file as standard input and copies it to standard output with line numbers added.

9.5.8 String length

Implement the function

```
int strnlen(char *s, int n);
```

that returns the number of characters in `s`, not including the terminating null character, unless this number exceeds `n`, in which case the function returns -1.

9.5.9 Number of occurrences

Implement the function

```
int charCount(char *s, char c, int n);
```

that return the number of occurrences of `c` in `s`, unless the length of the string exceeds `n`, in which case the function returns -1.

9.5.10 Character mapping

Implement the function

```
int toLower(char *s);
```

that changes to the corresponding lowercase letter each occurrence of an uppercase letter in the string that starts at `s`. It returns the number of characters changed.

9.5.11 Palindromes

Implement the function

```
int palindrome(char *s);
```

that returns 1 if the string that starts at `s` is a palindrome, and 0 otherwise. Make sure it works when the length of the string is 0 or 1.

9.5.12 Preventing buffer overflow

It is often required to store a string in a character array of fixed length `n` (a "buffer"), where the string is given as a pointer to its first character. To prevent buffer overflow it should be checked whether the string fits into the array. Write a function that serves this purpose.

9.5.13 Integer to string

Write a function with header `int itos(int i, char* str)` (`itos`: "integer to string") that places the digits of the decimal representation of `i` in `str` and returns the number of characters so placed. The decimal representation must not have leading zeros. What does this imply in the case of `i` equal to 0?

9.5.14 Anagrams

Write a program that finds anagrams of a given word that may occur in a given list words. For example suppose we want anagrams of "protein" and suppose we have available a file called `wordlist` consisting of words in lower-case letters separated by white space. Suppose the executable of our program is in the file `prog`. Then we use it as follows:

```
%prog protein < wordlist
%pointer
%protein
```

9.5.15 File of text to array of pointers to lines

The function

```
int lineList(char *arr[], int n, int lnLen);
```

assumes that standard input is a character file and that `arr` has length `n`. The function returns `m` and ensures that concatenating all strings `arr[i]` for $i = 0, \ldots, m-1$ results in a string containing the characters of `arr` broken up into lines separated by a single `'\n'`. Each line contains the words of `arr` separated by a single space. Moreover, each line is such that adding the first word of the next line would make the length of the line longer than `lnLen`[§].

9.5.16 Guessing game

There are several guessing games on the following pattern. Player A invites player B to select a secret number N. A asks B a few seemingly innocent questions about N. On the basis of the answers A hopes to impress bystanders by revealing the value of N.

A computer-based version of such a game goes as follows.

[§]This is the well-known "Telegram Problem" attributed to P. Naur but also to M.E. Conway.

```
Select an integer N in 0..7.
If N is one of the following four, then enter 1, otherwise enter 0.
0 1 2 3
    0
If N is one of the following four, then enter 1, otherwise enter 0.
- - - -
    1
If N is one of the following four, then enter 1, otherwise enter 0.
- - - -
    0
N is 6
```

B's answers are indented. The first set of numbers presented by the computer (playing *A*) is $\{0, 1, 2, 3\}$. The other two sets of four numbers in $\{0, 1, 2, 3, 4, 5, 6, 7\}$ are blanked out. Part of the exercise is to find three suitable sets of numbers—you can, but do not need to, include $\{0, 1, 2, 3\}$ among the three sets of numbers.

In case you need a hint for finding three suitable sets of numbers, here is one. A regular octahedron has eight faces and three equators. By "equator" of an octahedron I mean a set of four vertices that lie in a flat plane.

Chapter 10

Structures and unions

So far the array was the only way to aggregate an existing data structure into a composite one. An array is accessed by means of a numerical index. This has as advantage that the aggregate can be made arbitrarily large. Such flexibility requires all elements to be of the same type. This limitation is removed by the *structure*, which is an aggregate where the elements are accessed by names chosen by the programmer and where the elements do not have to be of the same type. These advantages come of course at a cost: it is not convenient to create structures with more than a small number of elements.

10.1 An example of using structures

Recall from Exercise 3.8.16 the unwieldy function header

```
double area(double ux, double uy,
            double vx, double vy,
            double wx, double wy)
```

in which the formal parameters describe a triangle by listing its vertices as being the endpoints of the vectors u, v, and w, where each of these vertices is in turn spelled out by means of its x and y coordinates. Thus six numbers are given, whereas we have only a single concept in mind. Structures can help to get code and concepts closer to each other.

A missing concept is that of the *vector*, as distinct from two numbers specifying the coordinates of its endpoint. Consider

```
struct vec {double x; double y;} u, v, w;
```

Here the new keyword `struct` introduces the definition of a structure. The identifier `vec` is the *tag* of the structure. The structure has two *components* named `x` and `y`, which are double-length floating-point numbers. The structure thus defined is the type of the variables `u`, `v`, and `w`. Another way of getting the same effect is to write instead

```
struct vec {double x; double y;};
struct vec u, v, w;
```

The first line reserves no storage; it introduces `vec` as the tag of a structure, together with the names and types of its components. The second line introduces the variables of the specified type and reserves storage for them.

There is variant of the second style of `struct` definition that relies on `typedef`. Here `vec` is introduced as a `typedef` rather than as a tag:

```
typedef struct {double x; double y;} vec;
vec u, v, w;
```

As a result, the definition of the variables is simplified.

Whichever of the three styles is used, there is no difference as far as the resulting variables `u`, `v`, and `w` are concerned: the components of `u` are available as the variables `u.x` and `u.y`, and similarly for `v` and `w`.

Structures are similar to arrays in the way they are initialized. We could have created some of the above structures with an initialization:

```
typedef struct {double x; double y;} vec;
vec u = {1.0, 2.0}, v = {1.0, 2.0}, w;
w.x = u.x + v.x;
w.y = u.y + v.y;
```

The uninitialized structure `w` is made equal to the sum of `u` and `v` by means of assignments.

In this example there is structure also above the level of vectors: the input to the function `area` is conceptually a triangle. That suggests defining a `struct` accordingly.

In Program 10.1: *Area of Triangle* we switch from vector terminology to the ancient one of Euclid: the `struct`s represent points, lines, and triangles. Points are pairs of numbers, in deference to a 17th-century invention.

Pointers to structures In principle pointers to structures are like other pointers. However, they come with special notation.

We could have done the above example with pointers:

```
typedef struct {double x; double y;} vec;
vec u = {1.0, 2.0}, v = {1.0, 2.0}, w;
vec *up = &u, *vp = &v, *wp = &w;
(*wp).x = (*up).x + (*vp).x;
(*wp).y = (*up).y + (*vp).y;
```

The parentheses are necessary because the dot operator has higher priority than the dereferencing operator. Because this combination of operators is so common, they can be replaced by the single operator `->`, as in the example below:

```
00 #include <stdio.h>
01 #include <math.h>
02
03 typedef struct {double x; double y;} point;
04 typedef struct {point A; point B;} line;
05 typedef struct {point A; point B; point C;} triangle;
06
07 double len(line l) {
08   double dx = (l.A).x - (l.B).x;
09   double dy = (l.A).y - (l.B).y;
10   return sqrt(dx*dx + dy*dy);
11 }
12 double heron(triangle T) {
13   point A = T.A, B = T.B, C = T.C;
14   line a = {B,C}, b = {C,A}, c = {A,B};
15   double la = len(a), lb = len(b), lc = len(c) ;
16   double s = (la+lb+lc)/2;
17   return sqrt(s*(s-la)*(s-lb)*(s-lc));
18 }
19 int main() {
20   point A = {2,2}, B = {3,1}, C = {4,3};
21   triangle ABC = {A,B,C}; line c = {A,B};
22   printf("distance between A and B: %lf\n", len(c));
23   printf("area: %lf\n", heron(ABC));
24 }
```

Figure 10.1: *Area of Triangle*, code to illustrate use of structures.

```
distance between A and B: 1.414214
area: 1.500000
```

Figure 10.2: Output of Program 10.1.

```
typedef struct {double x; double y;} vec;
vec u = {1.0, 2.0}, v = {1.0, 2.0}, w;
vec *up = &u, *vp = &v, *wp = &w;
wp -> x = up -> x + vp -> x;
wp -> y = up -> y + vp -> y;
```

10.2 Properties of structures

Arrays and structures have in common that they are aggregates of data of which each element is accessed by means of an index. In the case of arrays the index is an integer; in case of structures it is an identifier selected by the programmer. Arrays and structures also have in common that they allow initializers. In spite of these similarities, there are fundamental differences.

Consider for example a definition of a function that subtracts one vector from another:

```
vec sub(vec u, vec v) {
  vec temp = {u.x - v.x, u.y - v.y};
  return temp;
}
```

The analogous function for arrays would be

```
double *sub(double u[], double v[], int n) {
  double temp[n];
  for(--n; n >= 0; n--) temp[n] = u[n]-v[n];
  return temp; /* dangling pointer */
}
```

In function `double* sub` a local array `temp` is created and a pointer to it is returned. This is wrong in the sense of leading to code of which the effect is undefined at best and typically disastrous.

Why is that so? Consider a call to this function:

```
double u[] = {2, 2}, v[] = {3, 1}, *diff = sub(u, v, 2);
```

All variables local to a function disappear on exit from the function. Therefore, the array `temp` no longer exists when evaluation of the expression `sub(u, v, 2)` is completed. As a result, `diff` points to the area of memory that has been occupied by `temp`. After exit from `sub` its contents are no longer defined. This is an example of a dangling pointer.

The function for vector subtraction with structures is different. Consider

```
vec u = {2, 2}, v = {3, 1}, w = {4, 3},
    diff = sub(u, v);
```

Here the expression `sub(u, v)` has as value the structure itself, not a pointer to it. The left-hand side of the assignment, `diff` is itself a structure. The initialization of `diff` causes an entire structure to be copied, component by component, to another structure of the same type.

This point is illustrated by the code:

```
vec u = {2, 2}, v; v = u;
```

Such an assignment is not possible between arrays.

```
double u[] = {2, 2}, v[2]; v = u;
```

is not possible because `v` is not a variable. In the following:

```
void foo(double u[], double v[], int n) { v = u; }

double w0[] = {2, 2}, w1[2];
foo(w0, w1, 2);
```

`v` *is* a variable (because it is a pointer to the first element of the array that is the actual parameter), so the assignment `v = u` is legal. But because the parameters are passed according to the call-by-value rule, only the local copy of `v` is affected by the assignment. As a result, the call `foo(w0, w1)` has no effect.

10.3 Modeling objects with structures

Structures help us get concepts closer to code. Suppose we are working with triangles and that the attributes of interest are the three vectors that are the vertices and that additional attributes of interest are the angles and the area of a triangle. We can store these seven items in a structure as in

```
typedef struct {double x; double y;} vec;
typedef struct
   {vec u; vec v; vec w;
    double angleU; double angleV; double angleW;
    double area;
   } triangle;
```

To create a new `triangle`, we should only need to specify the three vertices, as the other four attributes can be computed from them. This suggests the function `mkTri` ("make triangle"):

```
triangle mkTri(vec u, vec v, vec w) {
  triangle t;
  t.u  = u; t.v  = v; t.w  = w;
  t.angleU = angle(sub(v,u), sub(w,u));
  t.angleV = angle(sub(u,v), sub(w,v));
  t.angleW = angle(sub(u,w), sub(v,w));
  t.area = area(t);
  return t;
}
```

which needs the functions `angle` for computing the angle between two vectors, `sub` for subtracting one vector from another, and `area`. See Program 10.3: *Triangles with Vectors*, where it all comes together. The example is a triangle laid out on a building site to ensure a right angle. According to an old carpenter's trick, you make the sides of lengths 3, 4, and 5. The output verifies the right angle.

```
00 #include <stdio.h>
01 #include <math.h>
02 const double pi = 3.1415926535;
03 typedef struct {double x; double y;} vec;
04 typedef struct {
05   vec u; vec v; vec w;
06   double angleU; double angleV; double angleW;
07   double area;
08 } triangle;
09 double len(vec v);
10 vec sub(vec u, vec v);
11 double heron(double a, double b, double c);
12 double area(triangle T);
13 double angle(vec u, vec v);
14 triangle mkTri(vec u, vec v, vec w);
15 int main() {
16   vec u = {0,0}, v = {3,0}, w = {0,4};
17   triangle T = mkTri(u,v,w);
18   printf("angles: %lf %lf %lf\narea: %lf\n",
19          T.angleU, T.angleV, T.angleW, T.area);
20 }
```

Figure 10.3: *Triangles with Vectors*, code to illustrate creation of triangle structures. See Program 10.5 for function definitions.

10.4 Unions

C is a typed language. This means that each variable has a type, which determines how the content of the variable's memory location (a bit pattern) is translated to the variable's value. The type system is a powerful help in avoiding programming errors. Suppose the language would allow us to treat a `float` variable with value 0.1 by mistake as an `int`. Then the integer we would get is not 0 or 1, but 1,036,831,949. However, sometimes the type system is too rigid. *Unions* can be used to provide the desired relief.

10.4.1 Typical use of unions

As an example of use of unions, consider the situation where we have a large collection of positive numbers of which some are in floating-point format, some are normal-size integers, and some are small integers in the unsigned short format. The `union` keyword allows us to define a type, say, `fltIntChar` that covers all these types. That is, if we define a variable `x` of type `fltIntChar`, then a value of any of these three types can be stored in `x`. After having stored a value of one of the

```
angles: 90.000000 53.130102 36.869898
area: 6.000000
```

Figure 10.4: Output of Program 10.3.

```
 1 double len(vec v) {
 2   return sqrt(v.x*v.x + v.y*v.y);
 3 }
 4 vec sub(vec u, vec v) {
 5   vec temp = {u.x - v.x, u.y - v.y};
 6   return temp;
 7 }
 8 double heron(double a, double b, double c) {
 9   double s = (a+b+c)/2;
10   return sqrt(s*(s-a)*(s-b)*(s-c));
11 }
12 double area(triangle T) {
13   return
14     heron(len(sub(T.u, T.v)),
15           len(sub(T.v, T.w)), len(sub(T.w, T.u)));
16 }
17 double angle(vec u, vec v) {
18   return (180.0/pi) * /* degrees per radian */
19          acos((u.x*v.x + u.y*v.y)/(len(u)*len(v)));
20 }
21 triangle mkTri(vec u, vec v, vec w) {
22   triangle t;
23   t.u  = u; t.v  = v; t.w  = w;
24   t.angleU = angle(sub(v,u), sub(w,u));
25   t.angleV = angle(sub(u,v), sub(w,v));
26   t.angleW = angle(sub(u,w), sub(v,w));
27   t.area = area(t);
28   return t;
29 }
```

Figure 10.5: Function definitions for Program 10.3.

types, we can still store a value in it of any of the other types allowed by the union declaration.

Suppose now that we want to print the value of `x`. To do this we need to know whether to call printf with `%f` or with `%d` as formatting code. So far, it was always

known of a variable what its type is. With a union type this is no longer the case. To remedy this situation, union types are defined in a format similar to that of structures. In this case,

```
typedef union {float F; int N; unsigned short US;} fltIntShrt;
fltIntShrt x;
```

This definition allocates a storage area large enough for any of the three types. In a commonly occurring case this is the four bytes needed for the `float` and `int` alternatives. Though this is more than the single byte needed for the `unsigned short` alternative (again, in this commonly occurring case; it is not mandated by C), all four bytes are allocated for any variable of type `fltIntShrt`.

Now we can indicate what type of value to expect in `x`. For each of the three possibilities we write `printf("%f", x.F)`, `printf("%d", x.N)`, or `printf("%d", x.US)`.

However, how does the programmer know which of `x.F`, `x.N`, `x.US` to choose? One cannot tell the type from the content of `x`, which is just a bit pattern. The only way this information can be supplied is by whoever stored the value in `x`. It is typically so that the only way to do this is to reserve a variable for this purpose. As the value of such a variable can be restricted to three possibilities, it is natural to make its type an enumeration, for example:

```
typedef enum {flt, intgr, ush} type;
```

Somehow the variable of the union type and the variable indicating its current type have to be kept together. A natural way to do this is to put them both in a struct:

```
typedef struct{fltIntShrt num; type t;} miscNum;
```

The desired large collection of numbers can then be an array of such structs. See Program 10.6. This program adheres to the usual discipline for unions:

> *A union type must be disambiguated according to the type of the value last stored in it.*

10.4.2 Abuse of unions

An interesting property of the C programming language is that the rule just introduced is only guideline for the programmer. The language does not enforce it. We can choose not to follow the guideline occasionally and in the process discover interesting facts about the way C is implemented.

Consider the declaration

```
union fltInt {float F; int N; unsigned char s[4];} x;
```

The declaration allows the variable `x` of type `union fltInt` to be converted to variables of three types: `float`, `int`, and `unsigned char[]`.

In this declaration, on a computer where `int` and `float` occupy 32 bits and where `char` occupies 8 bits, the three derived variables correspond to the same area of memory.

```
00 #include <stdio.h>
01
02 typedef enum {flt,   // float
03               intgr, // integer
04               ush}   // unsigned short
05   type;
06 typedef union {float F; int N; unsigned short US;}
07   fltIntShrt;
08 typedef struct{fltIntShrt num; type t;} miscNum;
09
10 void print(fltIntShrt x, type t) {
11   switch(t) {
12   case flt:
13     printf("%f ", x.F); break;
14   case intgr:
15     printf("%d ", x.N); break;
16   case ush:
17     printf("%hu ", x.US); break;
18   }
19 }
20 int main() {
21 const int n = 100000;
22 miscNum vec[n]; // collection of miscellaneous numbers
23   vec[0].num.F  = 0.12345; vec[0].t = flt;
24   vec[1].num.N  = 12345; vec[1].t = intgr;
25   vec[2].num.US  = 123; vec[2].t = ush;
26   for (int i = 0; i<3; i++)
27     print(vec[i].num, vec[i].t); printf("\n");
28 }
```

Figure 10.6: Typical use of unions.

Here is a program that uses the union declaration:

```
#include <stdio.h>

int main() {
  union fltInt {float F; int N; unsigned char s[4];} x;
  x.F = 0.1;
  printf("%f %d %x\n", x.F, x.N, x.N);
  for(int i=0; i<4; i++) printf("%u ", x.s[i]);
  printf("\n");
}
```

On a system with a big-endian processor it gives the following output:

```
0.100000 1036831949 3dcccccd
61 204 204 205
```

In the little-endian case the output is:

```
0.100000 1036831949 3dcccccd
205 204 204 61
```

The printing of `x.F` should present no surprise. The rest of the output allows us to have a look behind the scenes to see how the computer that produced this output represents a floating-point number.

The output `3dcccccd` of `x.N` in hexadecimal shows us bit by bit the content of `x`:

```
   3    d    c    c    c    c    c    d
0011 1101 1100 1100 1100 1100 1100 1101
```

The top row shows the hexadecimal digits from the output. The bottom row shows the corresponding bits.

The output

```
61 204 204 205
```

of `x` aliased as an array should give the same bits in the form of four eight-bit integers. So `3d` in hexadecimal should have the same bits as the integer denoted by the decimal numeral 61. Hmm, let's see ... `3d` is $3 * 16 + 13$, which is indeed 61. Next, `cc` is $12 * 16 + 12$, which is 204. And of course `cd` is one more. So that checks OK.

We have peeked at the bits of 0.1 inside the computer and we double-checked. To make sense of these bits, we first convert 0.1 to a binary floating-point numeral, using pencil and paper.

Observe that any number f can be written in binary as $\ldots b_2 b_1 b_0 b_{-1} b_{-2} \ldots$ such that

$$f = \ldots + b_2 2^2 + b_1 2^1 + b_0 2^0 + b_{-1} 2^{-1} + b_{-2} 2^{-2} + \ldots$$

When $f = 0.1$ it is clear that b_0, b_1, b_2, ... are all zero. The number of halves (0 or 1) that f has is indicated by b_{-1}; the number of quarters by b_{-2}, and so on.

Thus we have

$$\begin{aligned}
0.1 &= b_{-1}2^{-1} + b_{-2}2^{-2} + b_{-3}2^{-3} + \ldots &\Rightarrow\quad & b_0, b_1, b_2 \text{ all zero} \\
0.2 &= b_{-1} + b_{-2}2^{-1} + b_{-3}2^{-2} + \ldots &\Rightarrow\quad & b_{-1} = 0 \\
0.4 &= b_{-2} + b_{-3}2^{-1} + b_{-4}2^{-2} + \ldots &\Rightarrow\quad & b_{-2} = 0 \\
0.8 &= b_{-3} + b_{-4}2^{-1} + b_{-5}2^{-2} + \ldots &\Rightarrow\quad & b_{-3} = 0 \\
1.6 &= b_{-4} + b_{-5}2^{-1} + b_{-6}2^{-2} + \ldots &\Rightarrow\quad & b_{-4} = 1 \\
0.6 &= b_{-5}2^{-1} + b_{-6}2^{-2} + \ldots &\Rightarrow\quad & b_{-4} \text{ has been subtracted}
\end{aligned}$$

$$\begin{aligned} 1.2 &= b_{-5} + b_{-6}2^{-1} + b_{-7}2^{-2} + \ldots &\Rightarrow\quad & b_{-5} = 1 \\ 0.2 &= b_{-6}2^{-1} + b_{-7}2^{-2} + \ldots &\Rightarrow\quad & b_{-5} \text{ has been subtracted} \\ 0.4 &= b_{-6} + b_{-7}2^{-1} + b_{-8}2^{-2} + \ldots &\Rightarrow\quad & b_{-6} = b_{-2}, b_{-7} = b_{-3}, \text{ and so on} \end{aligned}$$

Because the value of 0.4 on the last line repeats an earlier one, we now know all further lines, hence all the infinitely many binary digits of 0.1:

$$0.00011\ 0011\ 0011\ldots$$

We rewrite this to ensure that the part before the power of two is between 1 and 2:

$$1.1\ 0011\ 0011\ldots \times 2^{-4}.$$

This is always possible by adjusting the exponent. Any fractional number, however big or small, can be specified by the exponent and the digits after the fractional point.

In principle this last form is the one that is stored as a floating-point number in the computer. To understand the actual bits we need to know that the computer on which the program was run uses for the C type `float` the *single-length IEEE-standard floating-point format*.

A bit of research into this format reveals that the 32 bits are used as follows:

```
1:        33222222222211111111110000000000
2:        10987654321098765432109876543210
3:
4:        seeeeeeeeffffffffffffffffffffff
5:        00111101110011001100110011001100...
6:           3   d   c   c   c   c   c   c
```

The top two rows 1 and 2 should be read downwards to give the serial numbers of the bit positions: from right to left we read $\begin{smallmatrix}0\\0\end{smallmatrix}$, meaning 00 through $\begin{smallmatrix}3\\1\end{smallmatrix}$, meaning 31. In row 4, the `s` in bit position 31 means that this bit determines the sign (0 for positive, 1 for negative). The `e` in bit positions $30\ldots23$ are eight bits for the exponent. The `f` in bit positions $22\ldots0$ are the 23 bits for the bits after the fractional point.

In row 5, we see the bits themselves. Because 0.1 is positive, the sign bit in position 31 is 0. The eight bits for the exponent in positions $30\ldots23$ can range from binary 00000000 to 11111111 and represent 0 through 255. According to the floating-point standard 127 is subtracted, so that both positive and negative exponents can be represented. So -4 shows up as 123; in bits this is `01111011`, which is what we see in row 5 below the `e`'s. The fractional bits are below the `f`'s. In row 6 the hexadecimal equivalents of the groups of four in row 5 are shown.

As a result we get `3dcccccc`, which is 0.1 truncated to fit into the 23 bits available for the fractional part. What we found is `3dccccd`, which is the result of rounding upwards. As the discarded bits start with `11`, we can tell that the upward-rounded version is a better approximation. Kudo to the `stdio` library for getting this right!

10.5 Exercises

10.5.1 Returning multiple results

So far, a function that computes multiple results had to return each in a separate parameter, as, for example, in

```
void minMax(int a[], int n, int *min, int *max) {
  int i;
  *min = *max = a[0];
  for(i=1; i<n; i++) {
    if (a[i] < *min) *min = a[i];
    if (a[i] > *max) *max = a[i];
  }
}
```

Structures allow such multiple outputs to be returned as a single value. Supply the missing definition in:

```
#include <stdio.h>

typedef struct {int min; int max;} pair;

pair minMax(int a[], int n);
int main() {
  int a[] = {3, 4, 2, 2, 9, 4};
  pair p = minMax(a, sizeof(a)/sizeof(a[0]));
  printf("%d %d\n", p.min, p.max);
}
```

10.5.2 Packaging an array

When an array is passed as an actual parameter to a function, it is rarely the case that anything can be done with the array without knowing its length. This information is then added in another parameter. The calling program has to make sure it passes the right number as the length. Wouldn't it be better if arrays would have their lengths built-in?

C is not designed that way. However, it is designed to be flexible enough to allow the programmer to do the right thing. Accordingly, in this exercise you supply the missing definition in the program in Program 10.7: *Array with Length*.

10.5.3 File of text to array of pointers to words

The function

```
int wordList(char *arr[], int n);
```

```
 1 #include <stdio.h>
 2
 3 typedef struct {int min; int max;} pair;
 4 typedef struct {int* array; int len;} arLen;
 5
 6 pair minMax(arLen ar);
 7 int main() {
 8   int a[] = {3, 4, 2, 2, 9, 4};
 9   arLen ar = {(int *)a, sizeof(a)/sizeof(a[0])};
10   pair p = minMax(ar);
11   printf("%d %d\n", p.min, p.max);
12 }
```

Figure 10.7: *Array with Length*: `struct` that pairs array with its length.

assumes that standard input is a character file and that `arr` has length `n`. The function ensures that the i-th word of the input file becomes a string and that `arr[i]` is a pointer to the first character of that string, for $i = 0, \ldots, m-1$, where `m` is the number of words if that number is less than `n`. In that case `arr[m]` contains the null pointer. If $m \geq n$, then `arr[n-1]` contains the null pointer and `arr[0..n-2]` point to the first $n-1$ words of the input.

The function returns `m` or `n-1`, whichever is less. You may truncate any word in the input that has more than a thousand characters to that length.

Chapter 11

Memory allocation

To run a program, memory needs to be allocated for the variables and the functions. *Static memory* is memory of which it is known at compile time that it will be needed at run time. This memory is allocated at compile time. There is also memory that can only be allocated at run time. This consists of areas known as "the stack" (for automatic allocation) and "the heap" (for dynamic allocation).

See Program 11.1: *Allocation Demo*, for some example allocations. Let us look at the output of the program on two different operating systems.

```
g++ under Mac OS:           g++ under Linux:

&f:       0x10cecddd0       &f:          0x4005c4
&a:       0x10cece078       &a:          0x601040
&b0:  0x7fff6caccb60        &b0:  0x7fffcec109d8
&b1:  0x7fff6caccb58        &b1:  0x7fffcec109d0
p0:       0x10d000890       p0:          0x73d010
p1:       0x10d100000       p1:   0x2b278ffd0010
```

The addresses are virtual addresses, not the physical addresses in the random-access memory of the machine. Both machines probably have a 64-bit architecture, so that virtual addresses can in principle go all the way up to `0xffffffffffffffff` (16 `f`s), hexadecimal for $2^{64} - 1$. The fact that the larger addresses have twelve hex digits suggest that the virtual address space is only 2^{48} bytes.

Memory for functions and for global variables is allocated at compile time, in static memory. The fact that `f` and `a` have similar, and small, addresses suggest that static memory is at the low end of the virtual address space. The fact that `b0`, the first local variable to be allocated, has the largest address suggests that the stack is at the high end of the virtual address space. The fact that `b1` is adjacent, and has a lower address, shows that the stack grows "downwards": from the high end towards the low end. The dynamically allocated addresses, those for `p0` and

p1, are in between, which suggests that the heap is between static memory and the stack.

```
#include <stdio.h>
#include <stdlib.h>
00 #include <stdio.h>
01 #include <stdlib.h>
02 typedef unsigned long nat;
03
04 nat a;
05 void f() {}
06
07 int main() {
08   nat b0, b1;
09   // Dynamic storage allocation:
10   nat *p0 = (nat*)malloc(1*(sizeof(nat)));
11   nat *p1 = (nat*)malloc(124567890*(sizeof(nat)));
12     printf("\n");
13     printf("&f: %16p\n&a: %16p\n"
14           , (void*)&f, (void*)&a);
15     printf("&b0:%16p\n&b1:%16p\np0: %16p\np1: %16p\n"
16           , (void*)&b0, (void*)&b1, (void*)p0, (void*)p1);
17 }
```

Figure 11.1: *Allocation Demo*, a program showing the effects of typical allocations.

When a function has an array as parameter, it needs in addition a parameter for the size of the array. Each array element is implemented as a pointer. It is the caller's responsibility to ensure that the entire contiguous sequence of pointers implied by the actual parameters can be dereferenced. In preconditions of functions I summarize this requirement by saying that the implied memory area is *allocated*.

For example the call to `sort` in

```
1 #include <stdio.h>
2
3 void sort(int c[], int n);
4 int main() {
5   int a[] = {0,1,2,3,4};
6   int b[] = {5,6,7,8,9};
7   sort(b, 10);
8 }
```

is syntactically correct, but its behaviour is undefined. Yet we should not be surprised when it sorts the concatentation of `b[0]..b[4]` and `a[0]..a[4]`. It is not guaranteed by C that `a[]` and `b[]` are allocated in adjacent areas. The precondition

to `sort` should specify that `c[0..n-1]` be allocated as a *single entity*. This is meant by being "allocated".

11.1 Automatic memory allocation

Strictly, variables local to a function should be declared not only with a type but also with either the keyword `auto` or `static`. In practice, as in this book, it is unusual for either to appear. This is because a local variable is, almost always, to be allocated "automatically" and the compiler takes the absence of `static` to imply automatic allocation.

Memory for a local variable is allocated at runtime on the stack on entry to a function and is de-allocated on exit from the function. Between these two events another function may be called and this typically causes additional automatic allocations. At any point in run time, any number of functions may be entered and not exited from. All of these may have caused memory to be allocated that is somehow released by the time the program normally terminates.

This potentially complex situation is simplified by the fact that whenever a execution of a function terminates, it is the function that was most recently entered. Thus functions obey the rule: last activated, first de-activated. This translates to the rule for automatic memory allocation: last allocated, first de-allocated. As this is reminiscent of the way plates in a cafeteria are stored, the area for automatic memory allocation is called "the stack". In conformance with the plate analogy we call the end at which allocations and de-allocations happen the "top", and the other end the "bottom", of the stack. Because the bottom of the stack stays in the same place in memory, it is convenient to place it at one of the ends of virtual memory space.

11.2 Static memory allocation

A consequence of automatic memory allocation of local variables is that their values are not retained from one call to a function to the next call to the same function. As this is not always convenient, C provides the possibility of declaring a local variable "static". Its storage is allocated at compile time in the static memory area.

However, this is an exceptional use of static memory: as we noted, most programs have no static locals. All programs define at least one function (`main`). As all functions are known at compile time, their code can be allocated in static memory. The same is true for global variables (those not declared in a function). Because they *can* be allocated in static memory, they *are*, because it is simple and efficient.

Because nothing is added to or deleted from statically allocated storage during run time, it is convenient to place it at the end of virtual memory space opposite to that of the stack.

11.3 Dynamic memory allocation

An advantage of automatic memory allocation is the automatic *de*-allocation: it happens automatically on exit of the function on whose entry the memory was allocated. For situations in which automatic storage allocation does not satisfy a programmer's needs there is dynamic storage allocation. Allocation of this type of storage is not triggered by the definition of a variable, but is allocated in the heap by a call to the function `malloc`, or one of its variants*. Such a call can occur wherever the programmer is allowed to write a function call. This gives a great deal of flexibility. It has the disadvantage that dynamically allocated memory needs to be de-allocated explicitly. This is easy to forget, so that it may happen that execution needs to be abnormally terminated because a request for storage cannot be granted for lack of room in the heap.

For an example call to `malloc`, see Program 11.2. The storage obtained can be accessed by means of the pointer returned by `malloc`, which contains the address of the first byte of the storage area allocated. If the attempt at allocation is not successful (the requested amount of memory may not be available), then the null pointer is returned. When the library function `free` is called with as actual parameter a pointer obtained by a call to `calloc` or `malloc`, then the storage allocated by that call is de-allocated.

The functions `malloc`, `calloc`, and `free` are made accessible by the standard library `stdlib`.

Let us consider an example of dynamic storage allocation. The example is motivated by a consequence of the array mechanism of C: a function cannot tell from its array parameter what size it is. It has to be supplied with this size in a separate parameter. The major problem with this is that the caller may not have supplied the right value. A minor problem is the clutter caused by the additional parameter. With the data structure for two-dimensional arrays we found necessary in Section 9.4 there is an additional problem. This data structure consists of a one-dimensional array of pointers to one-dimensional arrays containing the rows of a matrix. It is convenient to allocate this storage dynamically. However, to free it we need to know the numbers of rows and of columns. These considerations suggest we regard vectors and matrices as objects containing as components a pointer to storage for their elements as well as integers for the dimensions.

Let us call the vector type `vec` and make it a `struct` with two components: a pointer to an array containing the vector elements and, as second component, the length of this array:

```
typedef struct{double* V; int n;} vec;
```

The matrix type represents square matrices: those with an equal number of rows

*`calloc` is specifically for arrays. It takes two parameters, the number of elements and the size of each element, as measured by the `sizeof` operator. `calloc` sets every bit of the allocated area to zero. A general-purpose version of `calloc` is `malloc`, which takes as it its sole parameter the number of bytes to be allocated. `malloc` cannot be counted on to initialize the area allocated by it, so the caller cannot assume any particular contents. To be able to use these functions, include the `stdlib` part of the standard library.

and columns. The `struct` that implements a matrix object assumes that the matrix is stored as an array of pointers to arrays of `double`. Accordingly, in addition to an integer for the number of rows and columns, the `struct` has as component a pointer to pointer to `double`:

```
typedef struct{double** M; int n;} mat;
```

Here follow descriptions of the functions supporting vector and matrix objects.

Vector functions The use of vector objects is supported by the following functions.

- The function to create a vector needs the length of the vector to be created. The function allocates the required amount of dynamic storage and places a pointer to it in the appropriate component of the newly created vector. As the vector object itself is created in dynamic storage, the function returns a pointer to this object. Hence the declaration of this function:

  ```
  vec* mkVec(int n);
  ```

  The counterpart of this function is

  ```
  void freeVec(vec* v);
  ```

  The argument `v` needs to be a pointer returned by a call to `mkVec`. The effect of `freeVec` is to deallocate the heap area allocated by that call to `mkVec`.

- Upon creation of a vector we cannot count on it to contain any useful values. Function `fillVec` serves to place the values we need:

  ```
  void fillVec(vec* v);
  ```

- Sundry utility functions such as `copyVec` and `printVec`.

See Program 11.2: *Vector Functions*.

Matrix functions To support the use of matrix objects we have the following functions.

- In analogy to the vector case we have:

  ```
  mat* mkMat(int n);
  void fillMat(mat* A);
  ```

- In analogy to vectors we have

  ```
  void freeMat(mat* A);
  ```

```
 1 // file vectors.h
 2 #include<stdio.h>
 3 #include<stdlib.h>
 4
 5 typedef struct{double* V; int n;} vec;
 6
 7 vec* mkVec(int n);
 8 void fillVec(vec* v);
 9 void copyVec(vec* v, vec* w);
10 void printVec(vec* v);
11 void freeVec(vec* v);
```

```
 1 // file vectors.c
 2 #include"vectors.h"
 3
 4 vec* mkVec(int n) {
 5   vec* v = (vec*) malloc(sizeof(vec));
 6   v -> V = (double*)malloc(n*sizeof(double));
 7   v -> n = n;
 8   return v;
 9 }
10 void fillVec(vec* v) {
11   for(int i=0; i < (v -> n); i++)
12     scanf("%lf", &(v -> V[i]));
13 }
14 void copyVec(vec* v, vec* w) {
15   for(int i=0; i < (v -> n); i++)
16     (v -> V)[i] = (w -> V)[i];
17 }
18 void printVec(vec* v) {
19   for(int i=0; i < (v -> n); i++)
20     printf("%2.2lf ", v -> V[i]);
21   printf("\n\n");
22 }
23 void freeVec(vec* v) { free(v -> V); free(v); }
```

Figure 11.2: *Vector Functions*, functions of the `vector` module. For the meaning of line 2 in the lower listing, see Chapter 12.

The function for freeing the storage allocated to a vector, `freeVec`, is simple: `mkVec` only executes a single call to `malloc`. Therefore this storage can be de-allocated by a single call to `free`. In the case of a matrix, de-allocation is more interesting: `mkMat(n)` executes $n+1$ calls to `malloc`; n for the rows and

an additional one for the array of pointers to rows. For each of these there has to be a corresponding call to `free`.

- There is a function to compute the product of a matrix and a vector

  ```
  void matVec(mat* A, vec* x);
  ```

  It computes $y = Ax$ according to the formula

  $$y_i = \sum_{j=0}^{n-1} A_{ij} x_j$$

  for $i = 0, \ldots, n-1$.

  The resulting vector replaces the actual parameter for `x`. Note the abence of a formal parameter for the dimension of `A` and `x`.

- Function `matMult` multiplies two matrices. Note the absence of parameters for the dimensions of the matrices to be multiplied. This is possible because we have made the matrices into `struct`s that carry this information with them. It is not always necessary to create a third matrix to hold the product. Accordingly, `matMult` does not create it nor does it return such a third matrix. Instead the actual parameter `A` is overwritten with this product. If it is desired to save the contents of `A`, then the caller can save `A` in a copy beforehand.

  The matrix product $C = AB$ is formed according to the formula

  $$C_{ik} = \sum_{j=0}^{n-1} A_{ij} B_{jk}$$

  for all $i = 0, \ldots, n-1$ and $k = 0, \ldots, n-1$. From this formula it is clear that at least temporarily a third matrix for the product needs to exist. For this purpose `matMult` defines a matrix `C` in automatic storage. This has the advantage that allocation of such storage is faster and that de-allocation is automatic (by exiting from the function) as well as faster. Before function exit the product matrix is copied into the actual parameter for `A`.

```
 1 // file matrix.h
 2 #include"vectors.h"
 3
 4 typedef struct{double** M; int n;} mat;
 5 mat* mkMat(int p);
 6 void freeMat(mat* A);
 7 void printMat(mat* A);
 8 void fillMat(mat* A);
 9 void matVec(mat* A, vec* x);
10 void matMult(mat* A, mat* B);
```

```
 1 // file matrix1.c
 2 #include"matrix.h"
 3
 4 mat* mkMat(int p) {
 5   mat* result = (mat*)malloc(sizeof(mat));
 6   result -> n = p;
 7   result -> M = (double**)malloc(p*sizeof(double*));
 8   for (int i=0; i<p; i++) {
 9     result -> M[i] = (double *)malloc(p*sizeof(double));
10   }
11   return result;
12 }
13 void freeMat(mat* A) {
14   for(int i; i<A -> n; i++) free(A -> M[i]);
15   free(A -> M); free(A);
16 }
17 void printMat(mat* A) {
18   for(int i=0; i < A -> n; i++) {
19     for(int j=0; j < A -> n; j++)
20       printf("%2.2lf ", A -> M[i][j]);
21     printf("\n");
22   }
23   printf("\n");
24 }
25 void fillMat(mat* A) {
26   double x;
27   for(int i=0; i < A -> n; i++) {
28     for(int j=0; j < A -> n; j++) {
29       scanf("%lf", &x); A->M [i][j] = x;
30 } } }
```

Figure 11.3: *Matrix Functions, Part I*, some of the functions of the matrix module. For the meaning of the lines 2 in the listings, see Chapter 12.

```
 1 // file matrix2.c
 2 #include"matrix.h"
 3
 4 void matVec(mat* A, vec* x){
 5 // Purpose: write in x the product Ax.
 6 // Preconditions: A and x are created by mkMult
 7 // and mkVec and are of equal dimension.
 8   int n = x -> n; double y[n];
 9   for(int i=0; i < n; i++) {
10     double sum = 0.0;
11     for(int j=0; j < A -> n; j++)
12       sum += (A -> M[i][j])*(x -> V[j]);
13     y[i] = sum;
14   }
15   for(int i=0; i < n; i++) x -> V[i] = y[i];
16 }
17 void matMult(mat* A, mat* B) {
18 // Purpose: write in A the matrix product AB.
19 // Preconditions: A and B are created by mkMult
20 // and are of equal dimension.
21   int n = A -> n; double C[n][n];
22   for (int i=0; i < n; i++) {
23     for (int k=0; k < n; k++) {
24       double sum = 0.0;
25       for (int j=0; j < n; j++)
26         sum += (A->M [i][j])*(B->M [j][k]);
27       C[i][k] = sum;
28     }
29   }
30   for (int i=0; i < n; i++)
31     for (int j=0; j < n; j++)
32       A->M [i][j] = C[i][j];
33 }
```

Figure 11.4: *Matrix Functions, Part 2*, the remaining functions of the matrix module. See Chapter 12 for the meaning of line 2.

Chapter 12

Multi-file programs

12.1 Why programs get big

Programs get big because we want them to do many things and because we want them to be easy to use. For the study of large-scale program structure it is most interesting to look at programs that started out small and ended up, if not big, at least much less small. Examples of programs with such a history are many of the Unix tools. They started as a preliminary version thrown together in an afternoon. As they became more widely used, they acquired features and became easier to use. By the time they stabilized into the versions we know and love, they had turned into sizeable programs by significant effort on the part of several programmers. As this kind of growth is such a useful and natural phenomenon, let us look more closely at the process of program growth by small increments.

12.2 How programs get big

In C all executable code is in a function. A small increment typically takes the form of a function getting bigger. But functions are best when short, so excessive function growth is checked by the spawning of new functions. After a stage of this kind of growth, the program, though not containing any functions of excessive length, has too many pages of listing for a programmer to understand or takes too long to compile. This means that the program needs to be split into several parts. As a requirement for these parts is that they be compiled separately, they are called *translation units*. In most operating environments these reside in files.

Large program size causes several problems. Such programs tend to be complex, hence difficult to understand. To complete such programs in a reasonable amount of time requires a group rather than an individual programmer. Decomposing the program into *modules* helps to maintain an overview of the entire program and prevents the programmers working at cross purposes.

A module is motivated by a concept used by the designer of the overall system. The vectors and matrices in Chapter 11 are simple examples of such concepts. A

concept must be representable as a data structure and must be characterized by operations operations on instances of the concept. These operations are implemented by functions. The names of these functions and the information needed for calling them constitute the module's *interface*. The code defining the functions is the module's *implementation*.

12.3 Separate compilation

The essence of the module concept is the separation between a module's interface and the module's implementation. The separation is realized in C by writing the interface in a header file, which is C code consisting of type definitions and *declarations* of the module's functions. The header file does not contain any definitions of the functions. We adhere to the convention of giving header files names ending in "`.h`".

By the nature of modularization, the program contains more than a single module. Often a module A depends on another module B. Thus the user of B needs not just the interface for B. In such a situation the `.h`-file for B begins by including the `.h`-file for A.

Modularization not only makes the design of the program conceptually clearer, but also facilitates the generation of the executable code. This is done by organizing the code so that each module corresponds to a *translation unit* that can be compiled separately. In our example the translation unit for the vector module consists of the file `vector.h` for the interface and the file `vector.c` for the module's implementation. These are listed in Program 11.2. Similarly the matrix module is in a translation consisting of the files `matrix.h`, for the interface (see Program 11.3), and the files `matrix1.c` and `matrix2.c` for the implementation (see Program 11.3 and Program 11.4).

Program 12.1 is an example of user code that relies on the vector and matrix modules. Hence the inclusion in line 2 of the matrix interface, which, in turn, includes the vector interface.

Running the program is prepared by three separate compilations: of the vector module, of the matrix module, and of the file in Program 12.1. After these compilations their resulting object files are linked. Finally the result of linking is executed. One of the tasks of integrated development environments is to automate these steps so that the user only sees the output appear.

Sample interaction with Program 12.1 (input indented):

```
   0 1 2
0.00 1.00 2.00

   0 1 2
   1 2 0
   2 0 1
6.00 9.00 12.00
```

```
 1 // file main.c
 2 #include"matrix.h"
 3
 4 int main() {
 5   int n = 3;
 6   vec *v = mkVec(n), *w = mkVec(n);
 7   fillVec(v); copyVec(w, v); printVec(w);
 8   mat* A = mkMat(n);
 9   fillMat(A);
10   matVec(A, v); matVec(A, v); printVec(v);
11   matMult(A, A); matVec(A, w); printVec(w);
12   // pro forma:
13   freeVec(v); freeVec(w); freeMat(A);
14 }
```

Figure 12.1: Main program for matrix-vector multiplication. Line 13 is marked *pro forma* because the freeing of heap space does not make a difference in this situation, when the entire program is about to be terminated and all its storage returned to the operating system.

```
6.00 9.00 12.00
```

Part III

Algorithms

Chapter 13

Search

The purpose of *search*, in the sense of computing, is to find an item in a collection. Typically the collection is large. Algorithms for search vary widely: they depend on the nature of the item, on the structure of the collection, and on how the item is specified. In this chapter the collection is a sequence and the item is completely specified.

We first consider search in sequences that are stored and that are randomly accessible. We also consider search in sequences of which the elements are not stored, but are computed on demand.

In the programs of this chapter, the items are numbers. The programs can be readily modified to work for other types that are totally ordered. "Total order" means that, for any two unequal items, one is either earlier or later than the other in the ordering.

13.1 Search in a randomly accessible sequence

If we do not know whether the sequence to be searched is sorted, then every item has to be examined before we can conclude that it does not occur in the collection. In case it does, on average a sizeable proportion of items has to be examined before one is found. See Program 13.1: *Find*.

This algorithm uses the randomly accessible sequence only sequentially. As a consequence, it can, with suitable modifications, be used to search for an item in a file.

Binary search In Chapter 1 we considered, and left only partially solved, the problem of searching for a card with a specified name in an ordered stack of cards. We could not finish the solution because we lacked a formalism that is sufficiently precise. Now that we have such a formalism, in the form of C, we can complete the solution, provided we are willing to represent the ordered stack of cards with names by a sorted array of numbers.

```
00 int find(int a[], int n, int x) {
01 // Purpose: return -1 if x does not occur in a,
02 // otherwise return an i such that a[i] == x.
03 // Preconditions: n >= 0 and a[0..n-1] allocated.
04   int i = 0;
05   // x does not occur in a[0..i-1]
06   for(; i < n; i++) // x does not occur in a[0..i-1]
07     if (a[i] == x) return i;
08   // x does not occur in a[0..i-1] and i == n
09   return -1;
10 }
```

Figure 13.1: *Find*, a function that searches an array of which we do not assume that it is sorted.

We make the task precise by defining the *insertion point* of x with respect to a sorted sequence $a_0, \ldots, a_{n-1}$. The insertion point is the greatest i such that $a_i \leq x$. To ensure that the insertion point exists even when $a_0 > x$, we imagine a fictitious $a_{-1} = \infty$. In this case the insertion point is -1.

It is common to need to search in arrays of millions of items. When such a search needs be performed many times, it takes too long to have to examine every item. But if the array is sorted, then for an array of size one million one only needs to examine about twenty elements. At every stage of the search there is a remaining segment of the array that remains to be searched, and this segment can be reduced to half its size by the comparison of x with the middle of the segment.

See Program 13.2: *Find Ordered*.

13.2 Search in computed sequence

Integer square root Suppose one wants to compute the "integer square root" of a nonnegative n, which can be expressed as $\lfloor\sqrt{n}\rfloor$. Here I use the "floor" function: $\lfloor x \rfloor$ is the greatest integer not greater than x*.

One can consider the non-negative integers to be an ordered sequence. But this sequence does not need to be stored in an array: it can be computed on demand. See Program 13.3: *Integer Square Root* for a function that applies the idea of binary search to the ordered sequence of non-negative numbers.

Solving equations by binary search The sequence in which an algorithm searches can also be the floating-point numbers. Consider for example the sequence $f(x)$, where x is a real number. If reals a and b are such that $f(a)$ and $f(b)$ have different signs and if f is continuous, then there is an x such that $f(x) = 0$.

*The "ceiling" of x is the least integer not less than x, written as $\lceil x \rceil$.

```
00 int findOrd(int a[], int n, int x) {
01 // Purpose: return the insertion point IP for x w.r.t. a[0..n]
02 // Preconditions: n >= 0 and a[0..n-1] allocated.
03   if (x < a[0]) return -1; // 0 <= IP <= n-1
04   if (x >=  a[n-1]) return n-1; // 0 <= IP < n-1
05   int lb = 0, ub = n-1, m; // lb <= IP < ub
06   while (1) {
07   // lb <= IP < ub
08     if (lb+1 == ub) return lb;
09     // lb+1 < ub
10     m = lb + (ub-lb)/2;
11     // lb < m < ub and lb <= IP < ub
12     if (x < a[m]) ub = m; else lb = m;
13     // lb <= IP < ub
14   }
15 }
```

Figure 13.2: *Find Ordered*, a function that searches a sorted array.

This situation is another one where binary search is useful. When we bisect an interval $[a, b]$ with middle m into $[a, m]$ and $[m, b]$, then we can continue search with guaranteed success in one of these sub-intervals. Which one to search depends on the sign of $f(m)$.

In other words we can use binary search to solve non-linear equations in one variable. As an example we compute the square root of 2 by solving $x^2 - 0.5 = 0$. See Program 13.4: *Solve by Bisection*.

```
00 int sqRoot(int n) {
01 // Purpose: return greatest integer
02 // of which the square is not greater than n.
03 // Preconditions: n >= 0
04   if (n == 0 || n == 1) return n;
05   // n > 1
06   int lb = 1, ub;
07   for(ub = 1; ub*ub <= n; ub +=ub)
08     ; // empty body
09   // lb < ub && lb*lb <= n && n < ub*ub
10   while (lb+1 < ub) {
11     int m = lb+(ub-lb)/2;
12     // lb < m < ub
13     if (m*m <= n) lb = m; else ub = m;
14     // lb < ub && lb*lb <= n && n < ub*ub
15   }
16   // lb+1 == ub && lb*lb <= n && n < ub*ub
17   return lb;
18 }
```

Figure 13.3: *Integer Square Root*, a function that, given n, computes the greatest i such that $i^2 \leq n$.

```
 1 #include <stdio.h>
 2 #include <math.h>
 3
 4 double f(double x, double c) { return x*x - c; }
 5 double bisect(double a, double b, double fa, double fb,
 6               double (*f)(double, double),
 7               double c, float delta) {
 8 // Purpose: return x such that f(x',c) = 0 for an x'
 9 // such that |x-x'| <= delta.
10 // Precondition: f(a,c) and f(b,c) differ in sign
11 // and f continuous in x.
12   // x in [a,b]
13   double m, fm;
14   while (b-a > delta) {
15     m = a+(b-a)/2.0; fm = f(m, c);
16     if (fa*fm < 0) { b = m; fb = fm; }
17     else           { a = m; fa = fm; }
18   }
19   return m;
20 }
21 int main() {
22   double a = 0, b = 1;
23   printf("%lf",
24     bisect(a, b, f(a, 0.5), f(b, 0.5), &f, 0.5, 1e-6));
25   printf("  sqrt(0.5): %lf\n", sqrt(0.5));
26 }
```

Figure 13.4: *Solve by Bisection*, a program to solve an equation by bisection search. The output is `0.707107 sqrt(0.5):  0.707107`

The function `bisect` ignores the fact that floating-point numbers in a computer are not reals and ignores the fact that, translated to floating-point numbers, f is not continuous. The example is instructive because it shows that, in spite of this questionable translation, useful results are obtained. In fact, numerical analysis, a major application of computers, relies on similarly questionable translations.

13.3 Making a search space linear

When the search space is not a sequence, then searching it becomes more difficult. Even when one has to resort to brute-force search, that is, examining every item in the search space, it is sometimes possible to reduce the search effort by making the search space into a sequence.

Pythagorean triples Opportunities for making a search into a sequence sometimes crop up in unexpected corners. Let us reconsider the problem of generating Pythagorean triples, as solved by Program 8.8: *Pythagoras*. Let us consider it in conjunction with the problem of rendering curves in raster graphics. The curve is a continuous line while in raster graphics one is constrained to render with the cells of a rectangular grid. This juxtaposition suggests viewing Pythagorean triples as the points where a circle intersects points of a rectangular grid. This makes the search space into a sequence and results in an algorithm similar to Bresenham's Circle Algorithm[†] for rendering circles in raster graphics.

Remember that Pythagorean triples are triples of positive integers x, y, and r such that $x^2 + y^2 = r^2$. Examples are $\langle 3, 4, 5\rangle$, $\langle 6, 8, 10\rangle$, $\langle 5, 12, 13\rangle$, $\langle 9, 12, 15\rangle$, and $\langle 8, 15, 17\rangle$.

What may come to mind first is to fix r and regard all points with integer coordinates ("grid points") in the XY-plane as the search space, and test every one of these for $x^2 + y^2 = r^2$. The amount of work required is proportional to r^2. That is, when we increase r by a factor of 10, we need to do 100 times as much work. This is the brute-force approach taken in Program 8.8: *Pythagoras*.

Can we do better? The graph of $x^2 + y^2 = r^2$ is a circle in the XY-plane with centre at the origin and with radius r. The points in the XY-plane with integer coordinates form a grid. For every intersection of the circle with a grid point $\langle x, y\rangle$, there is a Pythagorean triple $\langle x, y, r\rangle$. For some values of r there are intersections, for others there are none.

Because of the symmetry of the circle, every intersection with a grid point exists in eight variants, of which only one is interesting. Whenever $\langle x', y'\rangle$ satisfies $x^2 + y^2 = r^2$, this is also the case for $\langle y', x'\rangle$, $\langle -x', y'\rangle$, $\langle y', -x'\rangle$, $\langle x', -y'\rangle$, $\langle -y', x'\rangle$, $\langle -x', -y'\rangle$, and $\langle -y', -x'\rangle$. Geometrically this means that we can restrict the search to a single octant of the circle, namely the $\langle x, y\rangle$ such that $0 < x < y < r$.

Brute-force search, as in Program 8.8: *Pythagoras*, corresponds geometrically to sweeping the entire two-dimensional area $0 < x < y < r$. The number of grid

[†]US patent 4,371,933. See *Computer Graphics*, 2nd ed., by Foley, van Dam, Feiner, and Hughes. Addison-Wesley, 1990; pp. 81–87.

points in this area is proportional to r^2. This suggests an algorithm that traces a path in the form of a sequence of grid points in the XY-plane that closely tracks the arc of the circle. The number of grid points on such a path is proportional to r.

The algorithm in Program 13.5, *Curve-Tracking Pythagoras*, constructs a path that I describe in terms of compass points. North (South) is increasing (decreasing) X coordinate; East (West) is increasing (decreasing) Y coordinate. The path starts at the Northwest end of the octant just outside the circle. From there it moves South until it is no longer outside the circle. This means that it may be on the circle, in which case it has found a Pythagorean triple. In that case it prints it and takes one more step South, ensuring that it ends up inside the circle. From there is moves East until it is no longer inside, which is the situation we started from. The net result is a macro-step consisting of a positive number of steps South followed by a positive number of steps East.

```
 1 #include <stdio.h>
 2
 3 void triples(int N) {
 4 // Purpose: print all Pythagorean triples <x,y,r> with r < N.
 5 // Precondition: N is positive.
 6   for(int r = 1; r < N; r++) {
 7     int x = 1, y = r;
 8     while (y > x) {
 9       // x*x + y*y >= r*r
10       while (x*x + y*y > r*r) y--;
11       if (x*x + y*y == r*r) {
12         printf("%d^2 + %d^2 = %d^2\n",
13                x, y, r);
14         y--;
15       }
16       // x*x + y*y < r*r
17       while (x*x + y*y < r*r) x++;
18       // x*x + y*y >= r*r
19     }
20   }
21 }
22 int main() {
23   printf("Input a positive integer.\n");
24   int N; scanf("%d", &N);
25   triples(N);
26 }
```

Output and input (indented):

```
Input a positive integer.
   50
3^2 + 4^2 = 5^2
6^2 + 8^2 = 10^2
5^2 + 12^2 = 13^2
9^2 + 12^2 = 15^2
8^2 + 15^2 = 17^2
12^2 + 16^2 = 20^2
7^2 + 24^2 = 25^2
15^2 + 20^2 = 25^2
10^2 + 24^2 = 26^2
20^2 + 21^2 = 29^2
18^2 + 24^2 = 30^2
16^2 + 30^2 = 34^2
21^2 + 28^2 = 35^2
12^2 + 35^2 = 37^2
15^2 + 36^2 = 39^2
24^2 + 32^2 = 40^2
9^2 + 40^2 = 41^2
27^2 + 36^2 = 45^2
```

Figure 13.5: *Curve-Tracking Pythogoras*, a program that performs linear search for Pythagorean triples by tracking a circle.

13.4 Exercises

13.4.1 Returning an interval

In Program 13.4 a solution is returned in the form of the midpoint of the first interval that gives a sufficiently accurate approximation. This exercise is to modify the program so that it returns the approximation in the form of the interval itself.

13.4.2 Variant of bisection search (1)

```
int find(int a[], int n, int x);
// Purpose: return i such that a[i] == x if such an i exist
// and return -1 otherwise.
// Preconditions: n > 0 and a[0..n-1] is sorted.
```

13.4.3 Variant of bisection search (2)

```
typedef struct{int lb; int ub;} intv;
intv find(int a[], int n, int x);
// Purpose: return {lb, ub} such that a[i] == x
// for all i in {lb, ub-1} if such an i exists.
// Otherwise return {lb, lb} where lb is the insertion point.
// Preconditions: n > 0 and a[0..n-1] is sorted.
```

13.4.4 Variant of bisection search (3)

```
int find(int a[], int n, int x);
// Purpose: return the i closest to 0 such that
// a[j] <= x for all j in [0,i] and
// a[j] >= x for all j in [i,n-1].
// Preconditions: n > 0 and a[0..n-1] is sorted.
```

13.4.5 Integer cube root

```
int intCubeRt(int n);
// Purpose: returns greatest positive x such that x^3 <= n.
// Preconditions: n >= 0.
```

Chapter 14

Conversion between numeral bases

In the early days of digital computers it was often asked whether a particular machine was decimal or binary. The first kind had hardware that represented numbers as decimal numerals. It was soon found that it is so easy for software to convert between numerals of different bases that decimal machines disappeared. In this chapter we look at such conversions.

From a mathematical point of view base conversion is similar to the problem to be solved in designing a vending machine. Among the functions such a machine provides is to return change in the form of coins. This requires the conversion of the amount due (specified in cents) to the number of quarters, dimes, and nickels equivalent to this amount. Another problem related to base conversion is that of converting a time period specified in seconds to conventional time units, such as minutes, hours, and weeks.

14.1 Converting numerals to an arbitrary base

A common task for software is to convert a decimal numeral, say,

```
123456789
```

into octal

```
726746425
```

or into binary

```
111010110111100110100010101.
```

"Number" is an abstract concept — "numeral" is concrete: it is a representation of a number as a sequence of digits. The digits can be binary, decimal, or from some

other base. Thus we can have binary, octal, decimal, and hexadecimal numerals, just to mention the numeral bases used around computers.

To print a number stored in a computer implies converting this number to a numeral, hence implies a choice of base for this numeral.

For example the result of executig

```
printf("%d\n", 3*5*7*11*13*17);
```

is that it displays

```
255255,
```

which is like a machine dispensing that number of dollars as 5 singles, 5 tens, 2 hundreds, and so on, until it dispenses 2 hundred-thousand dollar bills. But what it really is, is expressing a number as a decimal numeral.

Executing

```
printf("%d\n", 3*5*7*11*13*17);
```

is like setting n equal to $3 \times 5 \times 7 \times 11 \times 13 \times 17$ and solving the equation

$$n = d_0 + 10 \times (d_1 + 10 \times (d_2 + 10 \times (d_3 + 10 \times (d_4 + 10 \times d_5))))$$

where d_0, d_1, $d_2, \ldots$ are the numbers of ones, tens, hundreds, ... Of course six unknowns are not determined by a single equation. The information we use in addition is that d_0 has to be as large as possible under the constraint that it be at most 9. Given the value of d_0, the same has to hold for d_1. And so on for d_2, d_3, d_4, and d_5. See Program 14.1 *Base Conversion*.

The hexadecimal digits are the sixteen symbols

```
0 1 2 3 4 5 6 9 A B C D E F
```

The same system naturally extends to the 62 symbols

```
0 1 2 3 4 5 6 9 A B ... X Y Z a b ... x y z
```

This is why the precondition allows the base to be up to 62. Of all these possible bases only 2 (binary), 8 (octal), 10 (decimal), and 16 (hexadecimal) are widely used.

The conversion problem is simplified by the fact that each numeral position is the remainder to the same base. Some problems are base conversion problems disguised by the different positions indicating remainders to different bases.

The output of Program 14.1 is Figure 14.2.

14.2 Making change

When a transaction leaves a vending machine with an amount owing, it needs to convert that amount to an equivalent set of coins. Such a conversion can be expressed mathematically by the equation

$$a = 5n + 10(d + 25q)$$

where a, n, d, and q are the amount to be changed and the numbers of nickels, dimes, and quarters, respectively.

See Program 14.3, *Making Change*.

```
 1 #include <stdio.h>
 2 #include <string.h>
 3
 4 void reverse(char s[], int m, int n) {
 5 // Purpose: reverse order of characters in s[m..n].
 6 // Precondition: s[m..n] is allocated.
 7   char temp;
 8   for (; m<n; ++m, --n) {
 9     temp = s[m]; s[m] = s[n]; s[n] = temp;
10 } }
11 char conv(int d) {
12 // Purpose: convert digit d to ASCII character.
13 // Preconditions: 1 < d <= 62.
14   return d<10 ? '0' + d :
15                 d<36 ? 'A' + d - 10 : 'a' + d - 36;
16 }
17 int numConv(char s[], int x, int b) {
18 // Purpose: write in s[] the base-b numeral for x  as a null-
19 // terminated string with the most significant digit first,
20 // without leading zeros, preceded by the sign, except when x==0.
21 // In that case the string consists of the digit 0 only.
22 // Return the length of the numeral without the sign character.
23 // Preconditions: 62 >= b > 1, string length of s at least 33.
24   int i;
25   if (x == 0) { s[0] = '0'; s[1] = '\0'; return 1; }
26   if (x < 0) { s[0] = '-'; x = -x; } else s[0] = ' ';
27   for (i = 1; x > 0; i++, x /= b) s[i] = conv(x%b);
28   s[i] = '\0';
29   reverse(s, 1, strlen(s)-1);
30   return i-1; // not counting the sign character
31 }
32 int main() {
33   int n = 33; char s[n];
34   int m = 6; int b[] = {2,8,10,16,60,62};
35   for (int i = 0; i<m; i++)
36     printf("base: %d; length: %d; numeral: %s\n"
37           , b[i], numConv(s, 123456789, b[i]), s
38           );
39 }
```

Figure 14.1: *Base Conversion*, a function to convert a given number to a numeral in a given base.

```
base: 2; length: 27; numeral:  11101011011110011010010101
base: 8; length: 9; numeral:  726746425
base: 10; length: 9; numeral:  123456789
base: 16; length: 7; numeral:  75BCD15
base: 60; length: 5; numeral:  9VXX9
base: 62; length: 5; numeral:  8M0kX
```

Figure 14.2: Output of Program 14.1.

```
0 #include <stdio.h>
1
2 int main() {
3   printf("Input a positive multiple of 5.\n");
4   int amt; scanf("%d", &amt);
5   printf("%d cents changed to %d quarters ", amt, amt/25);
6   amt %= 25; printf("%d dimes ", amt/10);
7   amt %= 10; printf("%d nickels\n", amt/5);
8 }
```

Figure 14.3: *Making Change*, a program to compute change owing.

14.3 Numerals in heterogeneous base

When we read that Valerie Radcliffe of the UK won the London Marathon in 2003 in the (then) record time of 2:15:25, we are looking at a base-60 numeral. For longer time periods our conventional way of naming time periods becomes a heterogeneous base, because there are 24 hours to a day and seven days to a week. Beyond weeks the system becomes worse than heterogeneous: the bases even become variable.

Given a nonnegative integer n, the program we have in mind prints the equivalent of n seconds in s seconds ($0 \leq s < 60$), m minutes ($0 \leq m < 60$), h hours ($0 \leq h < 24$), d days ($0 \leq d < 7$), and w weeks ($0 \leq w$).

As there are 7 days in a week, d days and w weeks is $d + 7w$ days. Similarly, h hours and d days is $h + 24d$ hours, m minutes and h hours is $m + 60h$ minutes, and s seconds and m minutes is $s + 60m$ seconds.

Putting this all in a single formula gives

$$n = s + 60(m + 60(h + 24(d + 7w))).$$

As we want s to be less than 60, we have that s is the remainder on division of n by 60. We also have

$$(n - s)/60 = m + 60(h + 24(d + 7w)).$$

So, after we assign to s the remainder on dividing n by 60 and then assign to n the integer part of $n/60$, as in

```
s = n%60; n = n/60;
```

we have

$$n = m + 60(h + 24(d + 7w))$$

and we are left with a less complex problem. Moreover this simpler problem is similar to the original. In turn, we determine m, h, d, and w in the same way.

```
00 #include <stdio.h>
01
02 int main() {
03   printf("Input the number of seconds:\n");
04   int n; scanf("%d", &n);
05   printf("%d seconds is equivalent to:\n", n);
06   printf("%d second(s)\n", n%60);
07   printf("%d minute(s)\n", (n /= 60)%60);
08   printf("%d hour(s)\n", (n /= 60)%24);
09   printf("%d day(s)\n", (n /= 24)%7);
10   printf("%d week(s)\n", n/7);
11 }
```

Figure 14.4: *Time Period*, a program to print n seconds as seconds, minutes, days, hours, and weeks.

Suppose we have estimated that some program takes, for a certain input, a million seconds to execute. Does that leave us time to go out for lunch? We consult Program 14.4, *Time Period*. User input is indented.

```
Input the number of seconds:
   1000000
1000000 seconds is equivalent to:
40 second(s)
46 minute(s)
13 hour(s)
4 day(s)
1 week(s)
```

14.4 Exercises

14.4.1 Time Period

Program 14.4 first prints out the number of seconds, then the minutes, and so on until the number of weeks is printed last. This is the wrong order: the *first* thing

we want to know is the number of weeks, and the number of seconds last, if at all. Modify accordingly.

14.4.2 Numerals in English

```
void num(int n);
// Purpose: Print n in English.
// Preconditions: 0 < n <= 999,999
```

Examples (user responses indented):

```
Enter a number in 0..999999 in decimal digits.
     0
N in English:
zero

Enter a number in 0..999999 in decimal digits.
     999999
N in English: nine hundred ninety-nine thousand nine hundred ninety-nine

Enter a number in 0..999999 in decimal digits.
     20
N in English: twenty

Enter a number in 0..999999 in decimal digits.
     21
N in English: twenty-one

Enter a number in 0..999999 in decimal digits.
     100001
N in English: one hundred thousand one

Enter a number in 0..999999 in decimal digits.
     10011
N in English: ten thousand eleven
```

Chapter 15

Numerics

15.1 Numerical differentiation

Differentiation and integration are the fundamental operations of the calculus. There we learn rules for these operations on the basis of the nature of the function to be differentiated or integrated. Sometimes the rules are difficult to apply. Sometimes the rules fail and we have to be quite ingenious, as, for example, when we want to evaluate $\int_0^\infty e^{-x^2} dx$.

In calculus the result of differentiation or integration is always a *formula*. This formula can then be used to get any numbers we may need. Sometimes we do not need the result as a formula, but are only interested in the numerical value of a derivative or integral at a given point. In the case of the derivative we speak of *numerical differentiation*. We use as starting point the definition of the derivative of a function f at parameter value x as

$$\lim_{h \to 0} \frac{f(x+h) - f(x)}{h}.$$

We cannot evaluate this for $h = 0$. What we do instead is choose such a small value of h that we get a good approximation of the limit, and thereby of the value of the derivative at x.

Because we take a finite value of h we are not approximating the derivative at x but rather at some point between x and $x + h$. To avoid this bias we use the definition

$$\lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h}.$$

This is equivalent for well-behaved functions.

Hence the function definition

```
double deriv(double (*f)(double), double x, double h) {
  return ((*f)(x+h) - (*f)(x-h))/(2*h);
}
```

This definition leaves it up to the user to choose a suitable value for h. This is not easy. If we choose h much too small we will find that `x+h` or `x-h` evaluates to the same number as `x`. Even when this phenomenon does not occur, a too small value of h will cause rounding errors in the evaluation of `f` to have an unnecessarily large influence on the result. On the other hand, if we choose h too large, we are approximating the derivative at a point that is unnecessarily far away from x.

Program 15.1 computes approximations for the derivative of $\sin(1/x)$ at $x = 0.01$ for various values of h.

```
 1 #include <stdio.h>
 2 #include <math.h>
 3
 4 double f1(double x) { return(-cos(1.0/x)/(x*x)); }
 5 double f(double x) { return(sin(1.0/x)); }
 6
 7 double deriv(double (*f)(double), double x, double h) {
 8 // Purpose: return approximation to the derivative of f at x.
 9 // Preconditions: f differentiable around x and h > 0.
10   return ((*f)(x+h) - (*f)(x-h))/(2*h);
11 }
12 int main() {
13   printf("analytic value:      %4.15lf\n", f1(0.01));
14   double h; int n;
15   for (h = 1.0e-1, n = 1; n<20; h /= 10, ++n)
16     printf("- log h: %2d; deriv: %22.15lf\n",
17            n, deriv(&f, 0.01, h));
18 }
```

Figure 15.1: Numerical differentiation of $f(x) = \sin(1/x)$ at $x = 0.01$ for values of h ranging from too large to too small. The analytically determined derivative $f_1(x) = -\cos(1/x)/x^2$ is evaluated at $x = 0.01$ for comparison.

Numerical differentiation works best when the function changes slowly. To highlight the potential difficulties I have chosen as function to be differentiated $\sin(1/x)$, which changes more and more rapidly as x approaches 0. In Figure 15.2 we see the output of Program 15.1.

Because of the rapid fluctuation of the function it is important that h be chosen small enough: for the larger values the result is nowhere near the correct value. This is because h spans several cycles of $\sin(1/x)$. As h approaches 10^{-9} agreement with the correct value improves. Further decrease in h results in a decrease in accuracy due to rounding errors. As h approaches zero, $(f(x+h) - f(x-h))/2h$ approaches the undefined expression $0/0$. When $x = h$ the formula gives division by 0, which is an error. This shows up as `nan` instead of a numeral; `nan` stands for "Not A Number".

```
analytic value:      -8623.188722876839165
- log h:  1; deriv:     -3.328161668207064
- log h:  2; deriv:                    nan
- log h:  3; deriv:    555.383031209733531
- log h:  4; deriv:  -7298.881615648508159
- log h:  5; deriv:  -8609.337986153235761
- log h:  6; deriv:  -8623.050153572479758
- log h:  7; deriv:  -8623.187337153391127
- log h:  8; deriv:  -8623.188708684057929
- log h:  9; deriv:  -8623.188719358851813
- log h: 10; deriv:  -8623.188639811371104
- log h: 11; deriv:  -8623.189434731057190
- log h: 12; deriv:  -8623.195546508808548
- log h: 13; deriv:  -8623.219915904199297
- log h: 14; deriv:  -8623.346481329006565
- log h: 15; deriv:  -8614.775559578902175
- log h: 16; deriv:  -8700.817843987351807
- log h: 17; deriv:  -8582.023980352460057
- log h: 18; deriv: -12212.453270876721945
- log h: 19; deriv:      0.000000000000000
```

Figure 15.2: Output of Program 15.1. The right-hand column shows numerical approximations to the derivative of $\sin(1/x)$ at $x = 0.01$ for various values of h, which are indicated by $-\log_{10} h$. Apparently the best value for h is 10^{-9}: the numerical derivative agrees with the analytic value over the first nine digits.

15.2 Integration by Monte Carlo simulation

Suppose we have plotted on a sheet of paper the graph of a positive function $f(x)$ between x-values a and b. The lower edge of the paper is the X-axis. The sides of the paper are the lines $x = a$ and $x = b$. The top of the plot just hits the top of the paper. Suppose furthermore that we throw darts at the paper in such a way that every point on the sheet is equally likely to get hit by a dart. In this set-up we can estimate $\int_a^b f(x)dx$, which is the area under the graph of f, from the proportion of dart hits that fall under the graph of f.

This is the idea behind Program 15.3, *Monte Carlo*. To simulate the behaviour of the darts, we use a random-number generator. Its functionality is supplied by the functions `rand` and `srand` from the library that we include by the line

```
#include <stdlib.h>
```

Every call to `rand` delivers a randomly selected integer in the range from 0 to `RAND_MAX`. The function is implemented in such a way that each integer in this range has an equal probability of being the result.

```
00 #include <stdio.h>
01 #include <math.h> // for sqrt()
02 #include <stdlib.h> // for rand() srand() RAND_MAX
03
04 double f(double x) { return sqrt(1 - x*x); }
05 const double maxRand = (double)RAND_MAX;
06
07 double M_C(double a, double b, double bound,
08            long count, double (*f)(double)) {
09 // Purpose: approximate integral of f over interval [a,b]
10 // by Monte-Carlo simulation.
11 // Preconditions: a < b, bound > maximum of f over [a,b],
12 // and count is number of samples
13   double x, y; int hits = 0;
14   for (int i = 0; i < count; i++) {
15     x = a + (b-a)*(rand()/maxRand);
16     y = bound*(rand()/maxRand);
17     if (y < (*f)(x)) hits++;
18   }
19   return ((double)hits/count)*(b-a)*bound;
20 }
21 int main() {
22   srand(12345);                                    Output:
23   printf("%lf\n", 4*M_C(0,1.0,1.0,1e6,&f));
24   printf("%lf\n", 4*M_C(0,1.0,1.0,1e6,&f));        3.141120
25 }                                                  3.142524
```

Figure 15.3: *Monte Carlo*, using Monte Carlo simulation to estimate $\pi = 4\int_0^1 \sqrt{(1-x^2)}dx$.

A simulation program obtains random numbers by successive calls to `rand`. However, the numbers are not truly random. Each is in fact determined by the previous one. The result of the first call to `rand` is determined in the same way by the parameter of the last call to `srand` (from *set random*) preceding that first call. In this way we are assured that every execution of the program yields the same result, even though for the purposes of estimating the integral the successive values can be considered random.

The function `M_C` recognizes that it may be difficult to know the exact maximum of the function over the interval $[a, b]$. In effect, it gives the paper that we throw darts at a height given by the formal parameter `bound`. As long as we ensure that its value exceeds all function values in $[a, b]$, we get the correct result in the limit as we increase the number of throws of darts.

In the function `M_C` the x-coordinate of the place where a dart hits is stored

in the variable `x`. The y-coordinate goes into `y`. Whether the hit falls in the area under the graph is determined by the test `y < f(x)`.

The Monte Carlo method is only recommended for nested integrals, like

$$\int_{a_0}^{b_0}\int_{a_1}^{b_1}\int_{a_2}^{b_2} f(x,y,z)dxdydz.$$

For a single integral like $\int_a^b f(x)dx$ more accurate methods are feasible. These include the trapezoidal rule and Simpson's formula.

15.3 Numerical integration by Simpson's formula

The integral $\int_a^b f(x)dx$ from a to b of a function f of one variable can be visualized as the area between a and b under the graph of the function. $\int_a^b f(x)dx = I(b) - I(a)$, where I is the anti-derivative of f. Sometimes it is easy to find a formula for I. For example, if $f(x)$ is x^2, then $I(x)$ is $x^3/3$, so that we get $7/3$ for $\int_1^2 f(x)dx$.

Here we have obtained the numerical value by means of *analytical* (also called *symbolic* integration). If we don't want to, or are unable to, find a formula for I, then we can compute a numerical approximation of the required area, using only a formula for f. Such a computation is called *numerical* integration.

The crudest approximation is a straight line connecting $f(a)$ and $f(b)$ in the graph. The integral $\int_a^b f(x)dx$ is then approximated by

$$T_1(a,b) = (f(a) + f(b)) * (b-a)/2.$$

This is called T_1 because the area bounded by the straight line is a trapezoid. The subscript 1 is because we have spanned the entire area to be integrated by a single trapezoid. We get a better approximation $T_2(a,b)$ by doing the same for the left and right halves of the interval from a to b. This gives $T_2(a,b) = T_1(a,(a+b)/2) + T_1((a+b)/2,b)$.

T_2 uses three function values, which is enough to uniquely determine a quadratic polynomial to go through these points. Hence it should be possible to get the approximation exact when f is a quadratic polynomial. But T_2 does not live up to this ideal.

Paradoxically, we can get a better approximation by using not T_2 only, but by combining it with the worse approximation T_1. If we do this in the right way, we get $S_1 = T_2 + (T_2 - T_1)/3$, where $(T_2 - T_1)/3$ is the correction term to be applied to T_2. S_1 not only gives an exact result for quadratic polynomials, but throws in exactness for cubic polynomials as a bonus. S_1 is named that way because it is equivalent to Simpson's integration formula.

With three function values, we not only get the superior approximation S_1, but also the correction term $(T_2 - T_1)/3$, which usually tells us how good an approximation S_1 is. This suggests the following problem reduction for numerical integration of function f from a to b:

1. If the correction term is below a certain predetermined tolerance τ, then accept S_1 as the result with estimated accuracy of τ.

2. Otherwise, the result is the sum of the results of the same procedure applied to the left half of the interval $[a, b]$ and the result of applying it to the corresponding right half, each with tolerance $\tau/2$.

The more a function differs from a cubic polynomial, the more deeply such problem reductions are nested. As long as the function is continuous, the intervals ultimately become small enough that the difference with a cubic is negligible and then the correction falls below the tolerance so that further splitting is not necessary.

The attraction of problem reduction here is that we do not have to determine in advance how narrow the ultimate intervals are going to be. Problem reduction ensures that subdivision only happens as far as necessary. It adapts to the local behaviour of the function. This method could therefore be called *adaptive* Simpson integration*.

The property of being adaptive may result in considerable savings in computation. This is illustrated by the call to `adSimp` shown in Figure 15.4. For this integration the smallest interval in some areas had a width of 2^{-4}, while in other areas this width was 2^{-18}. If the entire integration interval from 0 to 1 had to be covered by this narrowest width of interval, then 2^{18}, which is about a quarter million, calls to `as` would be needed. In this example that number was 109.

The function on which the problem reduction is based has as header

```
double as(double a, double b,
          double fa, double fb,
          double tau, double (*f)(double))
```

The integration problem is completely specified by parameters `a`, `b`, `tau`, and `f`. To avoid re-computing function values, we have made `fa` and `fb` parameters as well.

In Figure 15.4 we see the problem reduction expressed in the function

```
double as(double a, double b,
          double fa, double fb,
          double tau, double (*f)(double)) {
  ...  T2 = ...    corr = ...
  if (corr < tau && -corr < tau) return T2+corr;
  return as(a, m, fa, fm, tau/2.0, f) +
         as(m, b, fm, fb, tau/2.0, f);
}
```

The function `adSimp` is only a shell around function `as`. The shell is introduced so that the user has to enter no more than the minimum required to specify the integration problem.

*But this term usually means a related procedure.

```
 1 #include <stdio.h>
 2 #include <math.h> // for sqrt
 3
 4 double f(double x) { // circle with unit radius
 5   return sqrt(1.0 - x*x);
 6 }
 7 double trap(double a, double b, double fa, double fb) {
 8 // area of trapezoid
 9   return (b-a)*(fa + fb)/2.0;
10 }
11 double as(double a, double b,
12           double fa, double fb,
13           double tau, double (*f)(double)) {
14 // function auxiliary to adSimp()
15   double T1 = trap(a,b,fa,fb);
16   double m = (a+b)/2.0;
17   double fm = (*f)(m);
18   double T2 = trap(a,m,fa,fm) + trap(m,b,fm,fb);
19   double corr = (T2-T1)/3.0;
20   if (corr < tau && -corr < tau) return T2+corr;
21   return as(a, m, fa, fm, tau/2.0, f) +
22          as(m, b, fm, fb, tau/2.0, f);
23 }
24 double adSimp(double a, double b,
25               double tau, double (*f)(double)) {
26 // Purpose: return integral of f  from a to b with tolerance tau
27 // Preconditions: a < b, 0 < tau, f continuous
28   return as(a, b, (*f)(a), (*f)(b), tau, f);
29 }
30 int main() {                                              Output:
31   printf("%lf\n", 4*adSimp(0.0, 1.0, 1.0e-4, &f));
32 }                                                         3.141593
```

Figure 15.4: *Adaptive Simpson*, integration based on the trapezoidal approximation.

15.4 Numerical Algebra

An important class of problems leads to a system of n linear equations in n unknowns. It is not uncommon for n to run into many thousands. For some applications n is whatever the capacity of the computer system will carry.

Solution methods are studied by means of linear algebra. Consider a system of equations such as

$$a_{00}x_0 + a_{01}x_1 + a_{02}x_2 \quad = \quad b_0$$

$$\begin{aligned} a_{10}x_0 + a_{11}x_1 + a_{12}x_2 &= b_1 \\ a_{20}x_0 + a_{21}x_1 + a_{22}x_2 &= b_2 \end{aligned} \tag{15.1}$$

where x_0, x_1 and x_2 are the unknowns. The a_{ij} are known quantities, the *coefficients*. The b_i are known quantities, the *right-hand sides*. The system (15.1) can be written as $Ax = b$, where A is a matrix with n rows and n columns; x and b, are column vectors each with n elements. When discussing the general n-dimensional case, we display the examples with $n = 3$. This saves a lot of tiresome occurrences of "...". The vectors are stored as one-dimensional arrays; the matrix as a two-dimensional array.

As a concrete example consider the network of resistors shown in Figure 15.5. Suppose we maintain a voltage V at one terminal and ground the other. What are the voltages at `v0` and `v1` and what are the currents through each of the resistors?

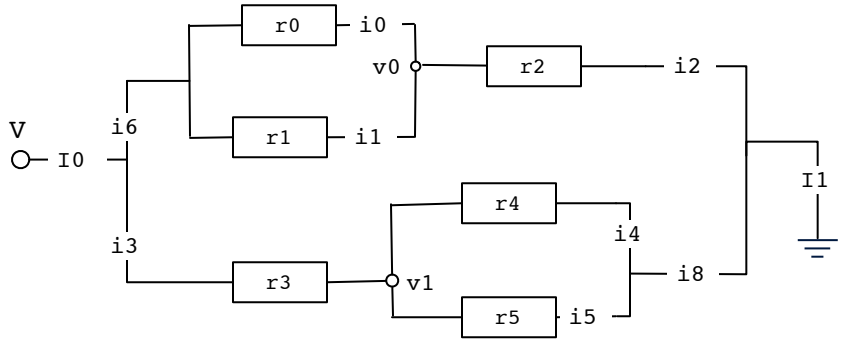


Figure 15.5: A network of resistors to be modeled as a system of linear equations. V, v_0, and v_1 are the voltages at the accompanying points; $r_0, \ldots, r_5$ are resistances given in kiloOhm. The problem is to determine the currents $I_0, I_1, i_8, i_0, \ldots, i_6$ and the voltages v_0 and v_1, given V and $r_0, \ldots, r_5$.

When I start by writing equations expressing immediately obvious facts such as $I_0 = I_1$, $i_2 = i_6$, and $i_3 = i_8$, then I get into trouble by ending up with too many unknowns, or too few. Even with small examples like this we need to follow a systematic approach; for large networks this is the only way.

The systematic way is to apply Kirchhoff's Current Law at every junction, which gives:

$$I_0 - i_6 - i_3 = 0 \tag{15.2}$$
$$i_6 - i_0 - i_1 = 0 \tag{15.3}$$
$$i_1 + i_0 - i_2 = 0 \tag{15.4}$$
$$i_3 - i_5 - i_4 = 0 \tag{15.5}$$

$$i_4 + i_5 - i_8 = 0 \quad (15.6)$$
$$i_2 + i_8 - I_1 = 0 \quad (15.7)$$

and Kirchhoff's Voltage Law at every voltage drop, which gives:

$$V - v_0 = i_0 r_0 \quad (15.8)$$
$$V - v_0 = i_1 r_1 \quad (15.9)$$
$$V - v_1 = i_3 r_3 \quad (15.10)$$
$$v_0 = i_2 r_2 \quad (15.11)$$
$$v_1 = i_4 r_4 \quad (15.12)$$
$$v_1 = i_5 r_5 \quad (15.13)$$

In this way we get 12 independent equations in 12 unknowns. Such a system has, according to linear algebra, a unique solution. In linear algebra such a system is written as $Ax = b$, where A is a 12-by-12 matrix and x and b are vectors of size 12. A contains the coefficients, x contains the unknowns, and b contains the right-hand-sides.

In this example the correspondence between $x_0, \ldots, x_{11}$ and the physical quantities of Figure 15.5 is as follows:

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}
I_0	i_0	i_1	i_2	i_3	i_4	i_5	i_6	i_8	v_0	v_1	I_1

I_0	i_0	i_1	i_2	i_3	i_4	i_5	i_6	i_8	v_0	v_1	I_1	rhs	eq. no.
1	0	0	0	-1	0	0	-1	0	0	0	0	0	15.2
0	-1	-1	0	0	0	0	1	0	0	0	0	0	15.3
0	1	1	-1	0	0	0	0	0	0	0	0	0	15.4
0	0	0	0	1	-1	-1	0	0	0	0	0	0	15.5
0	0	0	0	0	1	1	0	-1	0	0	0	0	15.6
0	0	0	1	0	0	0	0	1	0	0	-1	0	15.7
0	r_0	0	0	0	0	0	0	0	1	0	0	V	15.8
0	0	r_1	0	0	0	0	0	0	1	0	0	V	15.9
0	0	0	0	r_3	0	0	0	0	0	1	0	V	15.10
0	0	0	$-r_2$	0	0	0	0	0	1	0	0	0	15.11
0	0	0	0	0	$-r_4$	0	0	0	0	1	0	0	15.12
0	0	0	0	0	0	$-r_5$	0	0	0	1	0	0	15.13

Figure 15.6: Matrix A of coefficients of the linear equations modeling the resistor network in Figure 15.5. The column with the heading "rhs" shows the right-hand sides of the equations.

Let us now write a function that solves $Ax = b$; a function that, with A and b as input, produces x as output. Solving such systems is one of the most intensively

researched areas in all of computer science. Widely available software packages embody the result of this research. Use such a package whenever possible. The only reason for including the topic in this book is to enhance your programming skill: to be able to translate to code something you know how to do with pencil and paper. In case you don't know how to do it with pencil and paper, I'll start by showing you how.

We use the simplest method for solving a system of linear equations: *Gaussian elimination*. This method is based on the fact that when you add a multiple of an equation to another, the solutions to the system do not change. With a suitable choice of this multiple one of the variables has been eliminated from the resulting equation. Thus system (15.1) is transformed to

$$\begin{aligned} a_{00}x_0 + a_{01}x_1 + a_{02}x_2 &= b_0 \\ a'_{11}x_1 + a'_{12}x_2 &= b'_1 \\ a'_{21}x_1 + a'_{22}x_2 &= b'_2 \end{aligned} \tag{15.14}$$

This system is obtained by adding $-a_{10}/a_{00}$ times the first equation to the second, $-a_{20}/a_{00}$ times the first equation to the third, and so on. In matrix form system (15.14) is $A'x = b'$. This has the same solutions as $Ax = b$, but is easier to solve because the first column has zeros from the first row on down. The same method can be used to create zeros in the second column below the second row. After a third, similar, step we have $A''x = b''$ with the same solutions as $Ax = b$, where A'' has zeros everywhere below the diagonal. Written out in full, $A''x = b''$ is

$$\begin{aligned} a_{00}x_0 + a_{01}x_1 + a_{02}x_2 &= b_0 \\ a'_{11}x_1 + a'_{12}x_2 &= b'_1 \\ a''_{22}x_2 &= b''_2 \end{aligned} \tag{15.15}$$

This form is called *upper-triangular*; the first stage of Gaussian elimination is transformation to upper-triangular form.

The advantage of this form is that such equations are easy to solve: it is immediate that $x_2 = b''_2/a''_{22}$. We note this result and we remove the last equation from the system. The value just found for x_2 is substituted into the second equation, so that it becomes

$$a''_{11}x_1 = b''_1 - a''_{12}b_2/a''_{22}.$$

Similar substitutions remove the terms with x_2 from the two equations. This transformation of a three-dimensional upper-triangular system to a two-dimensional upper-triangular one is called a *back substitution* step. As the resulting system is upper-triangular, back substitution can be applied again. In this way repeated back substitution steps yield the values of all the three unknowns x_0, x_1, and x_2, thus completing the solution by Gaussian elimination.

The fact that Gaussian elimination is composed of two stages, transformation to upper-triangular form followed by back substitution, suggests that the function for solving a linear system have the following definition:

```
void linSol(double** ab, int n, double x[]) {
  uppTri(ab, n);
  backSubst(ab, n, x);
}
```

where `uppTri` is defined as in Program 15.7: *Upper Triangular*. Program 15.8: *Back Substitution* shows the function `backSubst`.

```
 1 void uppTri(double** ab, int n) {
 2 // Purpose: transform linear system Ax = b to upper triangular form.
 3 // Preconditions: n>1 and ab[0..n-1][0..n] contains the coeff.
 4 // in the first n columns and has b in the last column.
 5    for(int i=0; i<n; i++) {
 6    // there are zeroes below the diagonal in columns 0..i-1
 7    // sweep column i to zeroes
 8      for(int j=i+1; j<n; j++) {
 9      // j marches down the i-th column from the diagonal downwards
10      // add multiple of i-th row to j-th row
11        double mult = -ab[j][i]/ab[i][i];
12        for(int k=i; k <= n; k++) {
13        // sweep elements of j-th row
14          ab[j][k] += mult*ab[i][k];
15 } } } }
```

Figure 15.7: *Upper Triangular*, a function for transformation to upper-triangular form.

We now return to the equations generated by the network in Figure 15.5. In Program 15.9: *Resistors*, the array `data[12][13]` contains in its first 12 columns the coefficients of the A in $Ax = b$ while the 13th column contains the vector b of the right-hand sides. A call to `linSol` produces the vector x of values for the unknowns:

```
// I0   i0   i1   i2   i3   i4   i5   i6   i8   v0    v1   I1
   1.3  0.2  0.1  0.2  1.1  1.0  0.1  0.2  1.1  11.2  9.7  1.3
```

where we added as annotation the names of the unknowns in Program 15.9 and Figure 15.5.

Before leaving this topic I should confess to some cheating to make the solving of linear equations seem easier than it actually is. You may have wondered about the idiosyncratic order of the rows in the array `data[12][13]`. The reason is that there is a good chance that it does not work if you choose an order that does not jump

```
 1 void backSubst(double** ab, int n, double x[]) {
 2 // Purpose: perform back substitution on linear system Ax = b.
 3 // Preconditions: n>1 and ab[0..n-1][0..n] contains the coeff.
 4 // in the first n columns and has b in the last column.
 5 // A is in upper-triangular form.
 6   for(int i=n-1; i >= 0; i--) {
 7   // Find x[i] from i-th equation.
 8     x[i] = ab[i][n]/ab[i][i];
 9   //  Substitute this value in equations 0,...,i-1
10   //  and subtract from right-hand side.
11     for(int j=i-1; j >= 0; j--) {
12     // Right-hand side is in ab[j][n]
13       ab[j][n] -= x[i]*ab[j][i];
14     }
15   // From now on, forget about column i above diagonal.
16 } }
```

Figure 15.8: *Back Substitution*, a function for back substitution in a system of linear equations in upper-triangular form.

all over the diagram. "Does not work" means specifically that line `10` in `uppTri` in Program 15.7 encounters the value zero for `ab[i][i]` (called the "pivot"). At the least, `uppTri` should not always use the next row, but search candidate rows for one with a nonzero value.

Not only zero values cause `uppTri` to go awry: values that are too close to zero can lead to solutions that range from merely inaccurate to nonsense. The least a proper version of `uppTri` should do is to search for a row that has an entry in the `i`-th column with greatest absolute value. This strategy is called "partial pivoting". The "partial" suggests that one can be more ambitious in this respect.

At least one widely available software package can be relied on to give you the use of a state-of-the-art version of Gaussian elimination. Chances are that whatever system of linear equations you encounter in practice requires some other algorithm: perhaps Gauss-Jordan elimination, LU decomposition, Gauss-Seidel iteration, or Successive Overrelaxation.

```
 1 void uppTri(double** ab, int n);
 2 void backSubst(double** ab, int n, double x[]);
 3 void linSol(double** ab, int n, double x[]) {
 4 // Purpose: return in x[0..n-1] solution of linear system Ax = b.
 5 // Precondition: ab[n][n+1] contains the coefficients A in the
 6 // first n columns and b in the last column.
 7   uppTri(ab, n);
 8   backSubst(ab, n, x);
 9 }
10 int main() {
11   double const V=12,  // Volt
12     r0=4.7, r1=10, r2=47.7, r3=2.2, r4=10, r5=100; // kOhm
13   double data[12][13] = {
14 // 12 unknowns, followed by Right-Hand Side:              eq. no.
15 //  I0, i0, i1, i2, i3, i4, i5, i6, i8, v0, v1, I1, RHS
16   {  1,  0,  0,  0, -1,  0,  0, -1,  0,  0,  0,  0,  0}, //15.2
17   {  0, -1, -1,  0,  0,  0,  0,  1,  0,  0,  0,  0,  0}, //15.3
18   {  0,  0, r1,  0,  0,  0,  0,  0,  0,  1,  0,  0,  V}, //15.9
19   {  0,  0,  0,  1,  0,  0,  0,  0,  1,  0,  0, -1,  0}, //15.7
20   {  0,  0,  0,  0,  1, -1, -1,  0,  0,  0,  0,  0,  0}, //15.5
21   {  0,  0,  0,  0,  0,-r4,  0,  0,  0,  0,  1,  0,  0}, //15.12
22   {  0,  0,  0,  0,  0,  0,-r5,  0,  0,  0,  1,  0,  0}, //15.13
23   {  0,  1,  1, -1,  0,  0,  0,  0,  0,  0,  0,  0,  0}, //15.4
24   {  0,  0,  0,  0,  0,  1,  1,  0, -1,  0,  0,  0,  0}, //15.6
25   {  0, r0,  0,  0,  0,  0,  0,  0,  0,  1,  0,  0,  V}, //15.8
26   {  0,  0,  0,  0, r3,  0,  0,  0,  0,  0,  1,  0,  V}, //15.10
27   {  0,  0,  0,-r2,  0,  0,  0,  0,  0,  1,  0,  0,  0}};//15.11
28   const int n = 12; double x[n]; double* ab[n];
29   for(int i=0; i<n; i++) ab[i] = &data[i][0];
30   linSol((double**)ab, n, x);
31   for (int i = 0; i < n; i++) printf("%2.1lf ", x[i]);
32   printf("\n");
33 }
```

Figure 15.9: *Resistors*, a program for determining the currents and voltages in Figure 15.5. Compare the initialization of `data[12][13]` with the matrix in Figure 15.6. Rows needed to be permuted because of the absence of partial pivoting in `uppTri`.

15.5 Exercises

15.5.1 Simpson's method

Section 15.3 is based on the trapezoid approximation for the integral, as expressed in

```
double trap(double a, double b, double fa, double fb) {
// area of trapezoid
  return (b-a)*(fa + fb)/2.0;
}
```

where `fa` and `fb` are the function values at `a` and `b`, respectively.

If one adds `fm`, the function value at the point midway between `a` and `b`, then one can obtain a more accurate estimate by means of Simpson's formula, which is given by

```
double simpson(double a, double b,
               double fa, double fm, double fb) {
  return (b-a)*(fa + 4.0*fm + fb)/6.0;
}
```

In this way we get an estimate S_1 for the interval $[a, b]$. If we call S_2 the estimate obtained with Simpson's formula applied to the left and the right halves of this interval separately, then the optimal correction is $(S_2 - S_1)/15$.

Modify Program 15.4 to incorporate these improvements.

15.5.2 Testing with the Fundamental Theorem

Let f' be the derivative of a real-valued function f of a real variable. According to the Fundamental Theorem of Calculus we have $\int_a^b f'(x)dx = f(b) - f(a)$ and $\frac{d}{dx}\int_a^x f(y)dy = f(x)$. This exercise is to use these equalities to test numerical differentiation and numerical integration together. In Program 15.10 add a suitable definition of function `f`. You can use the definitions of `trap`, `as`, and `adSimp` defined elsewhere in this chapter.

In the definition of `fPrime` we have followed the recommendation of the experts[†] for the best value of `h` in function `deriv`.

15.5.3 Linear combination of two vectors

The *linear combination* of two n-dimensional vectors a and b is the n-dimensional vector c defined as $c_i = \mu a_i + \nu b_i$ for all $i = 0, \ldots, n-1$, where μ and ν are real valued coefficients, the *weights*; think of c as the *weighted average* of a and b. In Program 15.11 add a definition of the function `linComb` that replaces its parameter `a` with the linear combination of `a` and `b` with weights `mu` and `nu`.

[†] *Numerical Recipes in C* by Brian P. Flannery, Saul A. Teukolsky, William T. Vetterling.

```
00 #include <stdio.h>
01 #include <math.h> // for fabs, sqrt
02 #include <float.h> // for DBL_EPSILON
03
04 double f(double x);
05 double adSimp(double a, double b,
06               double tau, double (*f)(double));
07
08 double deriv(double (*f)(double), double x, double h) {
09   return ((*f)(x+h) - (*f)(x-h))/(2.0*h);
10 }
11 double fPrime(double x) {
12   return deriv(&f, x, sqrt(DBL_EPSILON));
13 }
14 double f1(double x) {
15   return adSimp(0.0, x, 1.0e-8, &fPrime);
16 }
17 int main() {
18   printf("%lf\n", adSimp(0.0, 0.5, 1.0e-4, &fPrime));
19   printf("should be: %lf\n", f(0.5) - f(0.0));
20   printf("%lf\n", deriv(&f1, 0.5, 1.0e-4));
21   printf("should be: %lf\n", fPrime(0.5));
22 }
```

Figure 15.10: For Exercise 15.5.2.

```
00 #include <stdio.h>
01
02 void linComb(double a[], double b[], int n,
03              double mu, double nu);
04 int main() {
05   double a[] = {0,1,2,3,4,5,6,7,8,9},
06          b[] = {9,8,7,6,5,4,3,2,1,0};
07   int n = sizeof(a)/sizeof(a[0]), i;
08   linComb(a, b, n, 0.5, 0.5);
09   for(i = 0; i<n; i++) printf("%1.1f ", a[i]);
10   printf("\n");
11 }
```

Figure 15.11: A function for the linear combination of two vectors.

15.5.4 Inner product

For two vectors a and b of length n, the *inner product* $a \cdot b$ of a and b is defined by the formula $a \cdot b = \sum_{i=0}^{n-1} a_i b_i$. The length $|a|$ of a vector a is $\sqrt{(a \cdot a)}$, the square root of the inner product of a with itself. The angle φ between a and b satisfies $\cos\varphi = a \cdot b/(|a| * |b|)$. Program 15.12 uses these formulas for a case that can easily be verified with pencil and paper. The interesting thing about vectors is that they exist in any number of dimensions, and the function `iProd` works for all them: two (as in this call), three, four (already difficult to visualize), ten, a thousand, or any finite number of dimensions.

Add the definition of `iProd` to Program 15.12.

```
 1 #include <stdio.h>
 2 #include <math.h>
 3
 4 double iProd(double a[], double b[], int n);
 5 // Purpose: return inner product of a and b of length n.
 6 // Preconditions: n > 0.
 7 const double degRad = 180.0/M_PI;
 8 // Number of degrees in a radian.
 9 // M_PI is the library's value for number pi.
10 int main() {
11   double a[] = {1.0,0.0}, b[] = {1.0, sqrt(3.0)};
12   int n = sizeof(a)/sizeof(a[0]);
13   double lenA = sqrt(iProd(a, a, n));
14   double lenB = sqrt(iProd(b, b, n));
15   printf("lengths of a and b: %f %f\n", lenA, lenB);
16   printf("angle between a and b: %f degrees\n",
17          degRad * // convert radians to degrees
18          acos(iProd(a, b, n)/(lenA * lenB)));
19 }
```

Figure 15.12: Example use of a function for the inner product of two vectors.

15.5.5 Evaluating a polynomial

Use inner product for a function that evaluates a polynomial

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

Chapter 16

Sorting

A common task programming task is to re-arrange a sequence of data into increasing or decreasing order. This is called *sorting*. The data need not be numerical: it often happens that we need to sort words or lines of text in alphabetical order. As sequences often take the form of an array, it is useful to be able to sort an array.

16.1 Selection sort

Let us consider how to sort an array `a` of integers. One possibility is to start by determining the location of a least element of the array and to exchange it with `a[0]`. Next, we determine the location of a least element of the remaining part of the array, and exchanging that element with `a[1]`. Continuing this way we end up with all of the array in sorted order. This method is called "selection sort". Of course, one can also start at the other end of the array, and then work downwards, as is done in Program 16.1: *Selection Sort*.

How long does selection sort take? If the array becomes twice as long, then the outer loop is executed about twice as many times. But the number of times the inner loop is executed also doubles. Thus, selection sort takes about four times as long when the length of the array is doubled. In computing terminology this is expressed by saying that selection sort has *quadratic complexity*: the amount of time it takes to sort an array of length n is proportional to n^2.

16.2 Quicksort

We next consider the "quicksort" method of sorting, which has the property that the amount of time it takes is proportional to $n \log n$ to sort an array of length n. This method was invented by C.A.R. Hoare.

Quicksort begins by finding the correct place for one element, the *pivot*, in the sorted version of the array. This preliminary stage is called *partitioning*. Once that is done, all one needs to do is to sort the part of the array to the left of the

```
00 #include <stdio.h>
01
02 void randPerm(int a[], int n);
03 // Purpose: place a random permutation of 0,1,...,n-1 in
04 // a[0..n-1].
05 // Preconditions: n >= 0 and a[0..n-1] allocated.
06
07 // Selection sort
08 void sort(int a[], int n) {
09 // Purpose: to sort a[0..n-1] in non-decreasing order.
10 // Preconditions: n >= 0 and a[0..n-1] allocated.
11   for(--n; n > 0; n--) {
12     int imax = n;
13     for(int i = n-1; i >= 0; i--)
14       if (a[i] > a[imax]) imax = i;
15     int temp = a[imax]; a[imax] = a[n]; a[n] = temp;
16   }
17 }
18 int main() {
19   int numErr = 0, n = 20000, a[n];
20   for(int i = 0; i < n; i++) a[i] = i;
21   srand(4321); randPerm(a, n);
22   sort(a, n);
23   for(int i = 0; i < n; i++) if (a[i] != i) numErr++;
24   printf("number of errors: %d\n", numErr);
25 }
```

Figure 16.1: *Selection Sort*, a function for sorting an array.

pivot independently from sorting the part of the array to the right of the pivot. Partitioning reduces each sorting problem to two smaller sorting problems. These smaller problems are also solved by quicksort, if there is anything left that needs sorting.

Partition Partitioning acts on the segment of the array `a` from index `p` up to but not including the element at index `q`. We denote this segment by `a[p..q-1]`. Here is an example of such an array segment.

```
array content: 4 1 9 2 11 10 3 6 8 0 7 5
array index:   p                        q
```

The element 4 at index `p` is taken as the pivot. We call an element "small" if it is less than or equal to the pivot; "big" if it is greater than or equal to the pivot.

The aim of partitioning is to perform element exchanges in such a way that to the left of the pivot there are only small elements; to the right only big ones.

The sorting algorithm usually does not find this ideal situation in the array. But something like it can be counted on to exist: we can always find indexes `i` and `j` are such that `a[p..i]` contains no big elements; `a[j..q-1]` no small ones. We choose `a[p]` as the pivot*, so we can initialize `i` to be equal to `p`. Similarly, we can initialize `j` to be equal to `q`. Because there are no elements in `a[q..q-1]`, we can safely say that there are no small ones in this segment. After this initialization, we try to increment `i` and decrement `j` while maintaining the property that `a[p..i]` contains no big elements and `a[j..q-1]` contains no small ones.

Because of their role, we call indexes `i` and `j` *cursors*. Partitioning is complete when we get the cursors next to each other. In the situation depicted below, the cursors can only be moved further inward after exchanging `a[i+1]` and `a[j-1]`.

We place `-` or `+` above an element according to whether it has been determined to be small or big, respectively. There is a question mark above it if the element has not yet been examined. The pivot, being neither small nor big, is marked by a question mark.

```
                0 - + ?  ?  ? ? ? ? - + +
array content: 4 1 9 2 11 10 3 6 8 0 7 5
array index:   p i                   j   q
```

After performing this exchange, we have

```
                0 - - ?  ?  ? ? ? ? + + +
array content: 4 1 0 2 11 10 3 6 8 9 7 5
array index:   p i                   j   q
```

The cursors can move inward some more, until they get stuck in the following situation:

```
                0 - - -  +  ? - + + + + +
array content: 4 1 0 2 11 10 3 6 8 9 7 5
array index:   p     i         j         q
```

Cursor `i` is blocked from moving inward by the big element 11; `j` is blocked from moving inward by the small element 3. An exchange of these blocking elements is called for:

```
                0 - - - -  ?  + + + + + +
array content: 4 1 0 2 3 10 11 6 8 9 7 5
array index:   p     i         j         q
```

In this way, partitioning alternates between moving the cursors inward as much as possible and exchanging when they are stuck. But all the time we need to check whether they run into each other, as they do now:

*The names `p` and `q` were chosen to serve as a nice pair to delimit the segment to be partitioned; `p` has nothing to do with the word "pivot".

```
              0 - - - -  +  + + + + + +
array content: 4 1 0 2 3 10 11 6 8 9 7 5
array index:   p       i j              q
```

Exchanging `a[p]` and `a[i]` results in the desired situation of having only small ones to the left of the pivot element 4; only big ones to the right. Partitioning is complete.

```
              - - - - 0  +  + + + + + +
array content: 3 1 0 2 4 10 11 6 8 9 7 5
array index:   p       i j              q
```

Program 16.2, *Partition*, is a function to effect partitioning.

```
 1 void swap(int a[], int i, int j);
 2 // Exchanges a[i] and a[j].
 3
 4 int partition(int a[], int p, int q){
 5 // Purpose: partition a[p..q-1]
 6 // Precondition: p < q-1 and a[p..q] allocated.
 7   if (a[p] > a[q-1]) swap(a, p, q-1);
 8   //a[p] <= a[q-1]
 9   int i = p, j = q-1;
10   while (1) {
11     //a[p..i] <= a[p] <= a[j..q-1] && i < j
12     while (a[i+1] < a[p]) i++;
13     // i < j && a[i+1] >= a[p]
14     if (i+1 == j) {swap(a, p, i); return i;}
15     while (a[j-1] > a[p]) j--;
16     if (i+1 == j) {swap(a, p, i); return i;}
17     // i+1 < j && a[i+1] >= a[p] >= a[j-1]
18     if (i+1 == j-1) { // a[i+1] == a[j-1] == a[p]
19       i++; swap(a, p, i); return i;
20     } // i+1 < j-1
21     i++; j--; // i < j
22     swap(a, i, j);
23 } }
```

Figure 16.2: *Partition*, a function for partitioning an array. The comment `a[p..i] <= a[p]` means that no element of the segment `a[p..i]` is greater than `a[p]`.

The code for quicksort The function `partition` acts on an arbitrary array segment `a[p..q-1]` and returns the index, let us call it `m`, of the pivot. By also making sure that to the left of the pivot there are no big elements, nor any small

elements to the right, the problem of sorting the whole array is reduced to sorting independently of each other the segments `a[p..m-1]` and `a[m+1..q-1]`. This fact can be recognized in the definition of `qsort1` in Program 16.3.

Quicksort is more complicated than selection sort. Averaged over all permutations of a sufficiently long array, it is also more efficient. A telling observation: the selection sort shown here took about the same time to sort an array of length 20,000 as it took quicksort to sort an array of length 2,000,000.

```
 1 int partition(int a[], int p, int q);
 2 // partitions a[p..q]
 3 void qSort1(int a[], int p, int q) {
 4 // sorts a[p..q-1]
 5   if (p >= q-1) // one element or fewer; do nothing
 6     return;
 7   int m = partition(a, p, q); // a[m] is in place
 8   qSort1(a, p, m); // sort a[p..m-1]
 9   qSort1(a, m+1, q); // sort a[m+1..q-1]
10 }
11 void qSort(int a[], int n) { // sorts a[0..n-1]
12   qSort1(a, 0, n);
13 }
14 void randPerm(int a[], int n);
15 // randomly permutes a[0..n-1]
16 int main(){
17   const int n = 2000000; int a[n], numErr = 0;
18   for (int i = 0; i < n; i++) a[i] = i;
19   srand(4321); // sets random-number generator
20   randPerm(a, n); qSort(a, n);
21   for (int i = 0; i < n; i++) if (a[i] != i) numErr++;
22   printf("number of errors: %d\n", numErr);
23 }
```

Figure 16.3: *Quicksort*, a program for sorting by the quicksort method.

16.3 Quicksort with an explicit stack

The very idea of the quicksort algorithm is expressed in terms of problem reduction: after partitioning the array segment `p..q-1` at index `m`, it is sufficient to sort segments `p..m-1` and `m+1..q-1` separately. An advantage that most programming languages have is that the problem reduction can be expressed by solving the two subproblems as function calls; see Program 16.3, *Quicksort*.

It will be instructive to see what kind of administration is involved in execut-

ing a program in which a function contains calls to itself. Let us follow a call `qSort1(a,0,10)`. Suppose that partition returns 5. This causes `qSort1(a,0,10)` to be replaced by `qSort1(a,0,5)` and `qSort1(a,6,10)`. Execution of the first of these starts next, so that we now have two unfinished function calls. Suppose that, in the course of executing `qSort1(a,0,5)`, partition returns 3. That will result in the following list of pending calls:

```
qSort1(a,0,3) qSort1(a,4,5) qSort1(a,6,10)
```

What is returned by partition will depend every time on the contents of `a`. The contents could be such that the next list of pending calls is:

```
qSort1(a,0,1) qSort1(a,2,3) qSort1(a,4,5) qSort1(a,6,10)
```

Every time execution of no more than one call can be started, while the runtime system has to remember somehow the remaining pending calls.

Now for the first time the call being executed does not result in calls to `qSort1`, giving as next state of the list:

```
qSort1(a,2,3) qSort1(a,4,5) qSort1(a,6,10)
```

With certain contents of `a`, the next states of the list could be:

```
qSort1(a,4,5) qSort1(a,6,10)
qSort1(a,6,10)
qSort1(a,6,8) qSort1(a,9,10)
qSort1(a,6,7) qSort1(a,8,8) qSort1(a,9,10)
qSort1(a,8,8) qSort1(a,9,10)
qSort1(a,9,10)
```

This last call does not give rise to further calls, so that after 11 recursive calls to `qSort1`, the original call `qSort1(a,0,10)` is completed[†].

The power of the function mechanism is that we only have to state a problem reduction for the computer to solve the problem without the programmer needing to be concerned about storing and scheduling pending function calls. This keeps the quicksort function simple.

The pending calls to the recursive qSort1 function represent a collection that grows and shrinks as time progresses. It does not matter which call is selected from the collection when the time comes to execute a function call. By following the details of the above example we saw that it always selects the most recently added call to be executed next.

In computing terminology it is said that the collection of pending calls is managed by the *last-in, first-out discipline*. Such a collection is called a *stack*, because this is the way one usually handles a stack of dinner plates. Adding something to the collection is called "pushing" it onto the stack. Removing something from the collection is called "popping" it off the stack. This terminology is inspired by the

[†]A function definition that contains a call to itself is said to be *recursive*.

device that is often used in cafeterias: it holds up a stack of plates by means of a spring so that the top plate remains at the level of the counter and the temptation never arises to remove any other plate than the top one.

What needs to be stored is not really a function call. All that matters is that we remember the left and right bounds of the segment that still needs sorting. We dedicate two arrays for this purpose: one (named `lft`) for the left bounds, and one (named `rht`) for the right bounds. We also dedicate a variable, `csr` (think "cursor") to indicate what part of the array is in use for the stack.

This way of implementing the stack is illustrated by the stack when it is at its highest in the above example:

```
                csr
                 |
        0  1  2  3  4  5  6  7
rht:    ?  ?  ?  ?  1  3  5 10
lft:    ?  ?  ?  ?  0  2  5  6
```

Because our starting point was how the leftmost call to `qsort` is replaced by two calls, we got a stack that grows from right to left. Most people are used to stacks that grow the other way around. That makes the example look like:

```
                   csr
                    |
        0  1  2  3  4  5  6  7
rht:    10 5  3  1  ?  ?  ?  ?
lft:    6  5  2  0  ?  ?  ?  ?
```

With this way of implementing the stack, pushing `p` onto the stack of left bounds is done by

```
lft[csr++] = p.
```

Popping the top item off the stack of left bounds and putting it into `p` is done by `p = lft[--csr]`.

From now on we consider the pair of stacks for the left and the right bounds as a single stack for array segments. So we will just talk about pushing onto and popping off "the" stack.

The reason for creating our own stack is the inefficiency of the recursive version. Our first opportunity for improvement is to use the fact that of the two parts resulting from partitioning only one needs to be stored for later treatment: we continue right away with the other half.

This leads to the code for `qSort1` in Program 16.4: *Non-recursive Quicksort*. The key is contained in the comment

```
/* ...
   to be sorted: a[p..q-1] and the segments on the stack
*/
```

Every time we go around the infinite loop, this is true. When it turns out that `a[p..q-1]` contains fewer than two elements, then there is nothing to be sorted in that segment. Accordingly, the top segment is moved from the stack to the variables `p` and `q`. But if there is also nothing on the stack, then nothing more needs to be sorted anywhere. The function can terminate.

```
 1 int partition(int a[], int p, int q);
 2 // partitions a[p..q]
 3 void qSort1(int a[], int p, int q) {
 4 // Purpose: sort a[p..q-1]
 5 // Preconditions: p < q and a[p..q-1] allocated.
 6 int max = 100, lft[max], rht[max], csr = 0;
 7 /* lft[0..csr-1] is the stack of left bounds
 8    rht[0..csr-1] is the stack of right bounds
 9    to be sorted: a[p..q-1] and the segments on the stack
10 */
11   while (1) {
12     if (p >= q-1) {
13       // one element or fewer, so a[p..q-1] already sorted
14       if (csr == 0) // and nothing on stack, so ...
15         return;
16       // something on stack; pop it into p and q
17       csr--; p = lft[csr]; q = rht[csr];
18     } else { // p < q-1: at least two elements
19       // partition a[p..q-1]
20       int m = partition(a, p, q);
21       // push left half
22       lft[csr] = p; rht[csr] = m; csr++;
23       // continue with sorting the right half
24       p = m+1;
25 } } }
```

Figure 16.4: *Non-recursive Quicksort*, a quicksort with explicit stack.

However, this still suffers from the main weakness of the recursive version: it is vulnerable to an unfavourable input that causes numbers of tiny segments to be pushed onto the stack. See Exercise 16.4.4.

16.4 Exercises

16.4.1 Partition

Here are some array contents.

```
a:   3 9 4 1 8 3 4 2 1

b:   3 1 4 1 8 3 4 2 9

c:   3 1 4 1 4 5 4 9 5

d:   1 3 4 1 4 5 4 9 5
```

Which could have been the result of partitioning? For those where your answer is "yes" state which element(s) must (could) have been the pivot.

16.4.2 French national flag

Given is an array containing codes for the colours in the French national flag (from left to right: blue, white, and red). The contents of this array cannot be assumed to be in any particular order. Write a program that permutes the contents of this array in the order as they occur in the French national flag[‡]. Full marks only if it takes the program no more than a single pass through the array.

16.4.3 Faster selection sort

Program 16.1 for selection sort spends most of its time scanning the array for finding a maximum element. A plausible way of speeding up the function is to find both a minimum and a maximum in a single scan. This is done in Program 16.5. Write the required function `minmax`.

16.4.4 Quicksort with stacks

Program 16.4 suffers from the same weakness as the recursive version: it is vulnerable to an unfavourable input that causes large numbers of tiny segments to be pushed onto the stack. In the most extreme situation the height of the stack can become equal to the number of elements in the array to be sorted.

This exercise is to remedy that: ensure that the height of the stack never exceeds $\log_2 n$, where n is the length of the array to be sorted. So the modest size of 100 for the arrays storing the stack is good for lengths up to 2^{100} of the arrays to be sorted.

16.4.5 Quicksort with structures

In Exercise 16.4.4 there are two weaknesses that can be remedied by means of structures. The first is that there are separate stacks of integers for the left bounds and for the right bounds of segments that remain to be sorted. We should take the

[‡]The problem is known as the problem of the *Dutch* national flag. However, in that flag the coloured bands are arranged vertically. As this book visualizes arrays horizontally, the nationality of the flag has been changed accordingly.

```
 1 #include <stdio.h>
 2 #include <stdlib.h> /* rand() srand() */
 3
 4 void swap(int a[], int i, int j);
 5 void randPerm(int a[], int n);
 6 void minmax(int a[], int p, int q);
 7 // Performs exchanges in array a that result in
 8 // a[p] being a minimum and a[q] a maximum element.
 9 void sort(int a[], int n) {
10   int p = 0, q = n-1;
11   for( ; p<q; p++,q--) minmax(a,p,q);
12 }
13 int main() {
14   int numErr = 0, i, n = 70000, a[n];
15   for(i = 0; i < n; i++) a[i] = i;
16   srand(4321); randPerm(a, n);
17   sort(a, n);
18   for(i = 0; i < n; i++) if (a[i] != i) numErr++;
19   printf("number of errors: %d\n", numErr);
20 }
```

Figure 16.5: A function for symmetric Selection Sort. Function `minmax` to be provided; see Exercise 16.4.3.

concept "segment" seriously and create structure for it that contains the left and right bounds as components. This will allow us to create a single stack of segments.

The second weakness is that the concept "stack" is left implicit in an array that contains the elements of the stack and an integer variable representing the stack's cursor. The concept of pushing or an element or popping the stack is left implicit by incrementing or decrementing the cursor variable at the correct moment.

The implied concepts are made explicit in the program in Figure 16.6. Use it to correct the weaknesses of the function `qSort1` of Exercise 16.4.4.

16.4.6 Sorting an array of strings

Adapt quicksort to sort an array of strings.

```
 1 typedef struct {int lft; int rht;} segment;
 2 segment mkSeg(int lft, int rht) {
 3   segment temp;
 4   temp.lft = lft; temp.rht = rht;
 5   return temp;
 6 }
 7 typedef struct {segment* st; int max; int ptr;} stack;
 8 /* st is array of segments of length max
 9    ptr is stack pointer */
10 stack mkStack(segment* st, int max) {
11   stack temp;
12   temp.st = st; temp.max = max; temp.ptr = 0;
13   return temp;
14 }
15 void push(segment* seg, stack* st) {
16   *((st -> st) + ((st -> ptr)++)) = *seg;
17 }
18 segment pop(stack *st) {
19   return *((st -> st) + --((st -> ptr)));
20 }
21 int empty(stack *st) {
22   return (st -> ptr) == 0;
23 }
```

Figure 16.6: A stack of segments.

Chapter 17

The power of squaring and halving

In Program 3.7, *Extreme Compounding* interest over a year, compounded by the second, was computed with 31,535,999 multiplications, being one less than the number of seconds in 365 days. I start by showing how to do this with fewer than a hundred multiplications. The technique is an example of one that has applications other than the computation of high powers.

17.1 Fast exponentiation

How can we speed up the computation of a^n where n is a large positive integer? We have

$$\underbrace{a * a * \cdots \cdots * a * a}_{31{,}536{,}000 \text{ factors}}$$

The key observation is that this expression does not specify in which order the multiplications are performed. Program 3.7 performs them from left to right, which is far from optimal. The value of

$$\underbrace{\underbrace{a \odot \cdots \cdots \odot a}_{i \text{ times}} \odot \underbrace{a \odot \cdots \cdots \odot a}_{j \text{ times}}}_{n=i+j \text{ times}} \tag{17.1}$$

depends only on n, where $\odot$ can be *any associative operation*. In Program 3.7 i was always made equal to 1.

Associativity gives us freedom to choose the order in which to activate the $\odot$ operation. Reverting to the special case where $\odot$ is multiplication, we consider how to decompose a^n. We can choose $a^n = a * a^{n-1}$, which is what Program 3.7 does. But we can also choose

$$a^n \quad = \quad 1 \quad \text{if } n = 0$$

$$\begin{aligned} &= (a * a)^{n/2} \quad \text{if } n \text{ is positive and even} \\ &= a * (a * a)^{(n-1)/2} \quad \text{if } n \text{ is odd} \end{aligned}$$

Translated to C this becomes

```
double exp(double a, int n) {
  if (n == 0) return 1.0;
  return n%2 == 0 ? exp(a*a, n/2) : a*exp(a*a,(n-1)/2);
}
```

This function is recursive. Not that there's anything wrong with recursion, but let's see if we can avoid it. Just out of curiosity.

One reason why recursion is elegant is that the value of the function remains implicit, a luxury we do not have in iterative functions. As the first step towards an iterative fast exponentiation function, let's make the value of the function explicit as the variable v.

$$\begin{aligned} v * a^n &= v * a * (a * a)^{(n-1)/2} \quad \text{if } n \text{ is odd} \\ &= v \quad \text{if } n = 0 \\ &= v * (a * a)^{n/2} \quad \text{if } n \text{ is positive and even} \end{aligned}$$

This suggests the three-parameter function `exp1`, which is auxiliary to `exp` with the usual interface.

```
void exp1(double* v, double a, int n) {
  if (n%2 == 1) { *v *= a; n--; }
  if (n == 0) return;
  a *= a; n /= 2;
  exp1(v, a, n);
}
```

The function is *pro forma* recursive, but not substantially so. This is because there is a single recursive call, which occurs just before the end of the function body, and it has the same actual parameters as the formal parameters. Under these conditions it can be replaced by a jump to the beginning of the body. This transformation is called *tail-call optimization*.

```
double exp1(double* v, double a, int n) {
L: if (n%2 == 1) { *v *= a; n--; }
  if (n == 0) return;
  a *= a; n /= 2;
  goto L;
}
```

Now is the time to remember that `goto` is almost everywhere forbidden, so we banish the offending four-letter word by writing:

```
double exp1(double* v, double a, int n) {
  while (1) {
    if (n%2 == 1) { *v *= a; n--; }
    if (n == 0) return;
    a *= a; n /= 2;
  }
}
double exp(double a, int n) {
  double v = 1.0;
  exp1(&v, a, n);
  return v;
}
```

Substituting the body of `exp1` for the call in `exp` gives an iterative fast exponentiation function; see Program 17.1: *Fast Exponentiation*.

```
double exp(double a, int n) {
  double v = 1.0;
  while (1) {
    if (n%2 == 1) { v *= a; n--; }
    if (n == 0) return v;
    a *= a; n /= 2;
  }
}
```

Figure 17.1: *Fast Exponentiation*, a function that efficiently computes a^n by repeated squaring of a accompanied by halving n.

This function performs at most about $2 \log n$ multiplications to compute a^n. It does this by combining the squaring of a with the halving of n whenever that is possible. Hence "the power of squaring and halving". We will now see that this translates to "the power of doubling and halving" for the *multiplication* of n and a.

17.2 Egyptian multiplication

The computer pioneers of the twentieth century felt smug about how computers led people to make exciting discoveries like fast exponentiation. Four thousand years ago the scribes of Egypt had found a similar algorithm. They avoided multiplication tables by observing that multiplication can be done by repeated addition and that this need not be onerous if one knows about the power of doubling and halving.

Little is known about the mathematics of the ancient Egyptians. They wrote on papyrus; it is a marvel that anything written on this medium has survived since the time the Rhind papyrus was written, which is somewhere between 1788 and 1580

B.C. The document has been deciphered sufficiently to have retrieved the example of multiplying 23 and 27, for which the following table is shown.*

```
      \      1      27
      \      2      54
      \      4     108
             8     216
      \     16     432
  Total     23     621
```

The first two columns represent the binary expansion of 23: the second column contains the necessary powers of 2, while the first column contains \ for binary one and space for binary zero. The third column contains the necessary doublings starting at 27. The first column indicates which items in the third column should be added up to obtain 23 times 27.

Our awe for the Egyptians should not inhibit us from tweaking the Rhind a bit. I suspect the scribes had a little side computation to obtain the binary expansion of 23. This is not necessary when the multiplication is done in the following format.

```
              23      27
      27      11      54
      54       5     108
     108       2     216
               1     432
     432       0
Total  621
```

Every next line in the third column is double that of the previous line. Every next line in the second column is one half of the value in the previous line. If there is a remainder in the division by two, then the third column is copied to the first. The algorithm halts when zero is reached in the second column. The result is obtained by adding the first column.

The Egyptian multiplication method can be obtained by making the following changes in Program 17.1:

- change `*` to `+`,
- change `1.0` to `0.0`, and
- change `exp` to something more appropriate.

giving the following result:

*"The Rhind Papyrus", by James R. Newman; in: *The World of Mathematics*, James R. Newman, ed., Simon and Schuster, 1956.

```
double mul(double a, int n) {
  double v = 0.0;
  while (1) {
    if (n%2 == 1) { v += a; n--; }
    if (n == 0) return v;
    a += a; n /= 2;
  }
}
```

In the early days of digital computers multiplication was done by a similar algorithm in software. Since then it has migrated downward, with the hardware performing basically the same algorithm. In the same early days, this strategy was even more important for the computation of quotient and remainder, as it is more complex.

17.3 "Egyptian" quotient and remainder

Just as multiplication can be defined as repeated addition, division can be defined as repeated subtraction. Just as the number of additions for multiplication can be reduced by a combination of doubling and adding, division can be speeded up by a combination of doubling and subtracting[†].

We consider the problem of dividing non-negative integer a by positive integer b. Let us call the quotient and remainder resulting from this operation m and u, respectively. We also consider the quotient and remainder resulting from dividing a by $2b$, and call these n and v, respectively.

We have

$$\begin{aligned} a &= mb + u \\ a &= n(2b) + v \end{aligned}$$

hence

$$\begin{aligned} u &= v \quad \text{if } v < b \\ &= v - b \quad \text{if } v \geq b \\ m &= 2n \quad \text{if } v < b \\ &= 2n + 1 \quad \text{if } v \geq b \end{aligned}$$

These equalities suggest Program 17.2: *Fast Division*.

[†]We follow the method of *Elements of Programming* by Alexander Stepanov and Paul McJones. Addison-Wesley, 2009. The program we present is a dumbed-down version of the one of Stepanov and McJones. Their programs are valuable not only for their algorithms, but also for the mathematical analysis that identifies the most general types for which the algorithms are valid. The programs in the book are written generically; the program here translates the generic type to integer.

```
 1 #include<stdio.h>
 2 typedef unsigned nat; // natural number
 3 typedef struct{nat q; nat r;} pair;
 4
 5 pair mkQRP(nat q, nat r) { // "mkQRP": make pair
 6 // Purpose: return the struct with q and r as components.
 7 // Preconditions: None.
 8   pair qr; qr.q = q; qr.r = r;
 9   return qr;
10 }
11 pair quotRem(nat a, nat b) {
12 // Purpose: return the struct with quotient and remainder as
13 // components resulting from division of a by b.
14 // Preconditions: a >= 0 and b > 0.
15   if (a < b) { return mkQRP(0, a); }
16   if (a-b < b) { return mkQRP(1, a-b); }
17   pair qr = quotRem(a, b+b); // qr.q = n and qr.r = v
18   if (qr.r < b) return mkQRP(qr.q + qr.q, qr.r);
19   else          return mkQRP(qr.q + qr.q+1, qr.r-b);
20 }
21 int main() {
22   int a = 123456789, b = 123;
23   pair qr = quotRem(a, b);
24   printf("%d %d\n", qr.q, qr.r);
25   printf("%d %d\n", a/b, a%b);
26 }
```

Figure 17.2: *Fast Division*, a function that efficiently performs integer division by repeated subtraction.

17.4 Fractional powers

Just as we can raise a number to the millionth power by fewer than a hundred multiplications by iterating squaring, so we can compute the millionth root, i.e. $b^{1/q}$ for q equal to a million, by iterating the square root operation. We combine this idea with Fast Exponentiation and write an algorithm for $b^{p/q}$ for arbitrarily large, or small, natural numbers p and q, provided of course that $q \neq 0$.

The following equalities are useful.

$$\begin{aligned} vu^{p/q} &= vuu^{(p-q)/q} \quad \text{if } 0 < q < p \\ &= v(\sqrt{u})^{2p/q} \quad \text{if } 0 < p < q \\ &= vu \quad \text{if } p = q \\ &= v \quad \text{if } u = \sqrt{u} \end{aligned}$$

The translation to code is in Program 17.3: *Fractional Powers*.

```
 1 #include<stdio.h>
 2 #include<math.h>
 3
 4 double fractPow(double u, int p, int q) {
 5 // Purpose: return u^(p/q)
 6 // Preconditions: p >= 0 and q > 0.
 7    double v = 1.0;
 8    while(p >= 2*q) { p /= 2; u *= u; }
 9    while (p != q) {
10      if (p > q) { p -= q; v *= u; }
11      else if (p < q) {
12        double u1 = u; p *= 2; u = sqrt(u);
13        if (u1 == u) return v;
14      }
15    }
16    return v*u;
17 }
18 int main() {
19   double u = M_PI, p = 31416, q = 10000;
20   printf("%lf\n", fractPow(u, p, q));
21   printf("%lf\n", exp((p/q) * log(u)));
22 }
```

Figure 17.3: *Fractional Powers*, a function that iterates squaring and square-root extraction to compute fractional powers.

Part IV

Method

Chapter 18

Top-down programming

I begin with an example of top-down programming, then explain the concept in general terms, and continue with additional examples.

18.1 Square root by guess-and-improve

Here is a function, named `iterRt`, for computing the square root of a non-negative floating-point number:

```
typedef double flt;

flt ab(flt x) { return x>0 ? x : -x; }
flt iterRt(flt a) {
  flt gOld, gNew = 1.0;
  do {gOld = gNew; gNew = (gNew + a/gNew)/2.0;}
  while (ab(gNew-gOld) > 1.e-6);
  return gNew;
}
```

The idea behind it is that an initial guess g at the square root of a can be improved and that by improving the guess often enough one can get an as good approximation to the square root as one wants.

How can we find out whether g is any good? Of course the closer g^2 is to a the better g approximates $\sqrt{a}$. For the sake of simplicity we start with the same initial guess for any a. Mostly the initial guesses are way off. For example if we use 1 as guess for everything, then our initial guess for $\sqrt{1001}$ will not be close to the true value.

It is therefore important to have not only a way of checking a guess, but also a way for improving one. Comparing g^2 with a is not the only way of checking: we can also see g is equal to a/g, as they should be when g is equal to $\sqrt{a}$. This latter method of checking has the advantage of suggesting a way of improving the guess.

If g guesses low, then a/g will be high, and vice versa. *Therefore,* $(g + a/g)/2$*, the average of the two, is a better guess than either.*

With this idea one can figure out why `iterRt` in the above code comes up with the square root. But isn't there a *systematic* way of converting the idea to code? To answer this question, let us break down into steps what the idea is. The steps are typeset **bold** in the enumeration below.

To obtain the square root of a

1. **decide** on a guess g of what the answer is
2. given g and a, **improve** g as much as possible
   (a) if g is **close enough** to its previous version then return g; otherwise make a **one-step improvement** of g and improve the result of that as much as possible
   (b) g is close enough to its previous version if their **difference** is less than 10^{-6}
      i. the difference between x and y is the **absolute value** of $x - y$
         A. the absolute value of x is x if $x \geq 0$; otherwise it is $-x$
   (c) a one-step improvement of g is $(g + a/g)/2$

Each of the underlined steps is so simple that one can immediately write down a function that performs this step. In Program 18.1 we find **improve** as `improve`, **close enough** as `close`, **one-step improvement** as `imp1`, **difference** as `diff`, and **absolute value** as `ab`.

This completes an example of what I call *top-down programming*. In the next section I explain this in general terms.

18.2 Use of functions for top-down programming

Functions are the main building blocks of programs. Every well-defined task, however small, can have a function devoted to it. To keep code manageable, it helps to keep functions small. Even the most complex task can be programmed with small functions because their code can contain function calls. It is not uncommon for function f_0 to call f_1, which calls f_2, and so on until f_{10} finally actually does something instead of passing the buck yet another time.

The functions of a program form a hierarchy. At the bottom are functions that do not call any function. At the top is the function that performs the entire application (the challenge to the designer of the program is to keep even this function small). Because functions higher in the hierarchy rely on functions that are lower, a plausible approach is to avoid calls to functions that have not been implemented. In this way you are sure you are building on solid ground. This is the *bottom-up* approach.

The fatal flaw of bottom-up is that initially one doesn't know what the functions at the bottom of the hierarchy are. In the beginning we only know the specification.

```
 1 #include <stdio.h>
 2 #include <math.h>
 3
 4 typedef double flt;
 5
 6 flt ab(flt x) { return x>0 ? x : -x; }
 7 flt diff(flt x, flt y) { return ab(x-y); }
 8 int close(flt x, flt y) { return diff(x, y) < 1.e-6; }
 9 flt imp1(flt g, flt a) { return (g + a/g)/2; }
10 flt improve(flt g, flt a) {
11   flt gNew = imp1(g, a);
12   return close(gNew, g) ? gNew
13                         : improve(gNew, a);
14 }
15 flt rt(flt a) { return improve(1.0, a); }
16 int main() {
17   printf("%lf %lf\n", rt(0.5), sqrt(0.5));
18 }
```

Figure 18.1: *Square Root*, a numerical recipe broken down in small steps, with each step in its own function.

It gives a good idea of what the top level function should be; it doesn't give any hints about the functions at the bottom. Indeed, in the beginning the entire hierarchy still needs to be discovered: we have no choice but to work top-down.

In top-down programming we have a problem to be solved by a function f. We note that the solution would be easy if only we had a function g for this and a function h for that. We don't. We go ahead and code f anyway. This way of proceeding is an example of wishful thinking, and the fact the top-down programming can be made to work is proof that wishful thinking can be made to produce results.

In a program hierarchy we have a chain in which function f_0 depends of f_1, which in turns depends on f_2, and so on. Whether this chain has to end depends on whether we think of a chain of function calls or a chain of function definitions. The chain of calls has to end, otherwise the program would not terminate. The chain of definitions does not need to end because of conditionals (statements or expressions). In a definition the recursive call needs to be part of a conditional with a branch containing no call to itself. In this way the possibility exists that at one stage of the computation a call to the function gives rise to a call to itself and that it does not at a later stage.

Often f_1, the function called by f_0, is the same as f_0. That is the function calls itself. Such a definition is said to be *recursive*; the corresponding execution is said to be an example of *recursion*. If f_0 calls f_1 and f_1 calls f_0, then we have a case of *mutual recursion*.

18.3 Greatest common divisor

How did Program 8.7 come about? Someone told us a recipe and illustrated it with an example (Table 8.3). Then, knowing the effect of `if`, `else`, and `while`, we tried to reproduce the desired behaviour. It looks like the attempt was successful. How can we be sure? How could we have discovered the recipe by ourselves? The key is some mathematics.

Consider the following theorem.

Theorem 18.1 *If x and y are positive integers, and if $x > y$, then $gcd(x, y) = gcd(x - y, y)$.*

This is more than a theorem. We can read it as

> *one can replace the problem of finding $gcd(x, y)$ by the problem of finding $gcd(x - y, y)$.*

This makes sense, because the new problem is likely to be easier, if only because the greatest of the two numbers of which we have to find the GCD is smaller.

Wishful thinking can take us directly from Theorem 18.1 to a function in C. See Program 18.2: *Recursive Euclid*.

```
0 int gcd(int x, int y) {
1 // Purpose: return the greatest common denominator of x and y.
2 // Precondition: x and y are positive integers.
4   return (x == y) ? x : ((x<y) ? gcd(x, y-x) : gcd(x-y, y));
5 }
```

Figure 18.2: *Recursive Euclid*: compute the greatest common denominator by wishful thinking.

18.4 Printing numerals

In section 14.1 we were given a number N and a base b and solved the problem of printing the digits of the base-b representation of N.

Requirement Given a nonnegative integer N and an integer b greater than 1. Print the digits of the base-b representation of N.

Analysis If, for example, we want the digits of the binary representation of 12345, then we obtain the last digit as the remainder of the division of 12345 by 2 (base $b = 2$) and we solve the problem of obtaining the remaining digits by obtaining

all digits of the integer part of 12345/2. Phrased in this way, we see that we have reduced the original problem for 12345 to the same problem for 12345/2. This is progress, because the reduced problem applies to a smaller number. By repeating the reduction, we ultimately reduce to the trivial problem of obtaining the digits of the binary representation of 0.

Implementation Let's give the function to solve this problem as header

```
void num(int N, int b)
```

The formal parameter `N` is the number of which we want to display the digits. One is typically only interested in binary, octal, decimal, or hexadecimal digits. This choice is determined by `b`. But we can set `b` to any value that makes sense. The function does not return a value, as we have decided to obtain the result by printing its digits. As the problem reduction in this case is so simple, we can directly write it out in code:

```
void num(int N, int b) {
  if (N == 0) /* do nothing */ return;
  printf("%d", N%b); num(N/b, b);
}
```

In our earlier example, we implemented the requirement by code that is arguably simpler:

```
while (N != 0) {printf("%d", N%b); N = N/b;}
```

What have we gained by using a function?

If we are content with the fact that this solution prints the digits in reverse order, then the answer is that we have gained nothing. But suppose that we are more ambitious and that we want the digits in the right order. Then we find that the problem reduction approach with the use of functions is more flexible.

Let us recapitulate the problem reduction we have just used, but this time emphasizing the order in which the digits are produced:

> *To print the digits of N in reverse order, print the last digit, as the remainder of N upon division by b, and then print in reverse order the remaining digits as the digits of $\lfloor N/b \rfloor$*[*].

Because the digits are printed in reverse order, the last digit is printed first. We now see that it is just as easy to specify as follows:

> *To print the digits of N in normal order, print in normal order all digits except the last by printing all digits of $\lfloor N/b \rfloor$ and then printing the last digit as the remainder of N upon division by b.*

Both problem reductions can be directly transcribed to code. The latter gives:

[*]The meaning of $\lfloor x \rfloor$ (the *floor* of x) is the greatest integer not greater than x. Similarly, $\lceil x \rceil$ (the *ceiling* of x) means the least integer not less than x.

```
void num(int N, int b) {
  if (N == 0) /* do nothing */ return;
  num(N/b, b); printf("%d", N%b); // for normal order
}
```

See Program 18.3, with output

```
987654321
123456789
```

```
 1 #include <stdio.h>
 2
 3 void numRev(int N, int b) {
 4   if (N == 0) /* do nothing */ return;
 5   printf("%d", N%b); numRev(N/b, b);
 6 }
 7 void num(int N, int b) {
 8   if (N == 0) /* do nothing */ return;
 9   num(N/b, b); printf("%d", N%b);
10 }
11 int main() {
12   numRev(123456789,10); printf("\n");
13   num(123456789,10); printf("\n");
14 }
```

Figure 18.3: Printing the digits of a number.

Chapter 19

Verification-driven programming

In this chapter "search" means determining whether a given value occurs in an array. If the array is ordered, then the answer can be found quickly, even when the array is very large, by an algorithm called *binary search*. If the array is not known to be ordered, then the appropriate algorithm is *linear search*. Binary search has proved to be a surprisingly tricky problem. This is the reason why we use binary search to demonstrate a method of programming that not only produces code but also a proof that the code achieves its intended purpose.

The pioneer programmers learned to be careful about the code they submitted to be run on the computer. Most learned it the hard way by finding that the printout contained no information other than that there was an unspecified error in line 13 of their thousand-line program. They were lucky if they could have another try the same day.

Modern programming is different. After a modification aimed at correcting what you thought was "the" error you can run the new version instantly. This lessens the incentive to make sure that the correction is correct. It is not uncommon for a novice programmer to be caught in a seemingly endless loop of: Oops! ... ah, of course ... Oops!, and so on.

In this chapter I introduce what may be called "verification-driven programming". It is based on the technique of proving correctness by means of assertions. We can think of assertions as relations between values of variables, relations that have to hold whenever execution passes the point in the code where the assertion is placed. When an assertion is simple enough, it is a condition according to the definition of C. In that case the assertion can be checked as part of the computation by writing a statement of the form `assert(`*condition*`)`. Verification-driven programming does not rely on executability of assertions, but on the possibility of expressing assertions in English in comments in the code.

One needs to make sure that there are enough assertions to "cover" the code and to make sure that the assertion about the final state conforms to the specifi-

cation of the program. The method of assertions only ensures *partial* correctness, which means that the final assertion holds *if* execution terminates. *That* execution terminates also has to be proved somehow; a common error is that it doesn't ("the program goes into an infinite loop").

Verification-driven programming is more than the method of verification by assertions. Experience shows that when you write code first, it is too hard to find a set of assertions that prove that it does what you want it do. It is obvious that one should start with assertions, because the specification of the program is in that form rather than in the form of code.

But one can't really start with the assertions, as these refer to code. So there can't be assertions without code and it's too hard to find the assertions when the code is in place—a chicken-and-egg situation. Verification-driven programming solves the deadlock by alternating the writing of assertions and code: a bit of the one, then a bit of the other, and so on. This not only makes correctness proof by verification feasible, but helps to discover an algorithm that solves the problem.

19.1 Binary Search

Background In Section 1.3.1 we attempted an informal description of an algorithm for binary search. The first attempt was clearly inadequate. The second attempt, although quite involved, still left some loose ends. We abandoned the project, hoping for success with a better approach.

Part of the difficulty was that the task is more difficult than might appear at first sight. Bentley* describes his experience as a teacher:

> *... we are to determine whether the sorted array* `X[1..N]` *contains the element* `T`*. Precisely, we know that* $N \geq 0$ *and that* $X[1] \leq X[2] \leq \ldots \leq X[N]$*; when* $N = 0$ *the array is empty. The types of* T *and the elements of* X *are the same; the pseudo code should work equally well for integers, reals or strings. The answer is stored in the integer* P *(for position): when* P *is zero* T *is not in* `X[1..N]`*, otherwise* $1 \leq P \leq N$ *and* $T = X[P]$*.*
>
> ...
>
> *I've assigned this problem in courses at Bell Labs and IBM. Professional programmers had a couple of hours to convert the above description into a program in the language of their choice; a high-level pseudo code was fine. At the end of the specified time, almost all the programmers reported that they had correct code for the task. We would then take thirty minutes to examine their code, which the programmers did with test cases. In several classes and with over a hundred programmers, the results varied little: ninety percent of the programmers found bugs in their programs (and I wasn't convinced of the correctness of the code in which no bugs were found).*

* "Programming Pearls" by Jon Bentley, Addison-Wesley, 1986.

> *I was amazed: given ample time, only about ten percent of professional programmers were able to get this small program right. But they aren't the only ones to find this task difficult: in the history in Section 6.2.1 of his Sorting and Searching, Knuth points out that while the first binary search was published in 1946, the first binary search without bugs did not appear until 1962.*

The problem is basically the one discussed in Section 1.3.1. As we suggested there, we should be more ambitious and solve a more useful problem than the one posed by Bentley: in case the element searched for does not occur, the program should tell us where it should be inserted in case we wanted to do so. Don't take it for granted that the more ambitious version is harder to write.

Another part of the difficulty is that it is difficult to be precise in English; in C one can't help being precise. You now understand enough of C to read, if not appreciate, the solution in Program 13.2. To appreciate this code we need to be sure it is correct and how one might arrive at a correct solution to the problem.

In view of Bentley's experience quoted above it may seem a bold claim that this mysterious bit of code is free of errors. In the remainder of this section we systematically develop a verified version of Program 13.2 without assuming any knowledge of an algorithm.

Before using the logic of verification, we start with the mathematics of the problem.

Definition 19.1 *Let $a_{-1}, a_0, \ldots, a_{n-1}$ be a non-decreasing sequence of integers, except for a_{-1}, which equals $-\infty$. The insertion point i_x of x with respect to a is the greatest i such that $a_i \leq x$.*

Theorem 19.1 *Let the sequence a be as in Definition 19.1. Let $n > 2$ and let m be the result of integer division if $n-1$ by 2. It is the case that $i_x < m$ if $x < a[m]$ and that $i_x \geq m$ otherwise.*

This is not only a theorem. It can also be used as a rule for reducing the problem of finding the insertion of x with respect to $a_{-1}, a_0, \ldots, a_{n-1}$ to a smaller problem, where the sequence is reduced to either the left or the right half of a. And this reduction can be huge: binary search comes into its own when n is in the millions. Binary search is also what makes dictionaries and telephone directories feasible in book form.

We already know what the beginning of the function looks like:

```
00 int findOrd(int a[], int n, int x) {
01 // Purpose: return the insertion point IP for x w.r.t. a[0..n]
02 // Preconditions: n >= 0 and a[0..n-1] allocated.
```

The typical case for an insertion point i_x is to have $a_{i_x} \leq x < a_{i_x+1}$. There are two untypical cases: one is that $x < a_0$; in this case we use the imaginary $a_{-1} = -\infty$. The other is that $x \geq a_{n-1}$; in this case the insertion point is $n-1$. These observations write the next few lines.

```
03   if (x < a[0]) return -1; // 0 <= IP <= n-1
04   if (x >=  a[n-1]) return n-1; // 0 <= IP < n-1
```

Now we have the insertion point bounded on both sides by indexes of the array. The remainder of the algorithm brings these bounds closer to each other until the lower bound `lb` and the upper bound `ub` are next to each other.

```
05   int lb = 0, ub = n-1, m;
```

One can label every assertion. I have done this for the one in line `06`; labels for the other ones have been omitted, as they would remain unused.

```
06 U:  // lb <= IP < ub
07     if (lb+1 == ub) return lb;
08     // lb+1 < ub
```

Lines `07 -- 08`: if the bounds are next to each other, we are done, as the insertion point is `lb`. If the bounds are not next to each other, there is an index `m` between them that divides the gap between the bounds as well as possible into two equal halves.

```
09     m = lb + (ub-lb)/2;
10     // lb < m < ub and lb <= IP < ub
```

In line `09` the temptation exists to write simply `m = (lb+ub)/2`. This works most of the time, but fails when the sum gives overflow. This can happen even when neither bound gives overflow.

```
11   if (x < a[m]) ub = m; else lb = m;
12   // lb <= IP < ub
13   goto U;
14 }
```

In line `11` the comparison `x < a[m]` is used to determine in which of `a[lb..ub]` the insertion point lies. Line `12` is a stronger version of line `06`. Line `12` has been weakened back to become the same as line `06`. Hence line `13` transfers control to line `06`.

Termination is proved by observing that $ub - lb$ decreases every time around the loop while maintaining $ub > lb$.

Presentation of the algorithm We have derived an algorithm for binary search in such a way that it has also been proved correct. I claim the "derived" is justified, not because I can write a program that automates the process, but in the sense that at every step along the way it is clear what needs to be done. What more could one want?

There is more to be done. Almost everywhere where programming is done, *the* `goto` *statement is forbidden.* So a further step is needed, which is to rewrite the algorithm derived by systematic development to an equivalent and socially acceptable form. This is what you find in Program 13.2: *Binary Search.*

19.2 Linear search

It is required to determine whether `int x` occurs in `int a[n]`. This is too vague for a specification. There are several ways to make it sufficiently precise to proceed.

```
int find(int a[], int n, int x) {
// Purpose: return -1 if x does not occur in a[0..n-1] otherwise
// returns least i such that x does not occur in a[i..n-1].
// Preconditions: n >= 0 and a[0..n-1] allocated.
 ...
}
```

As one can see, perhaps not the most obvious way of making the general idea more precise. One can say that it shows the scars of struggles with work done to simplify the results of other approaches.

```
int find(int a[], int n, int x) {
// Purpose: return -1 if x does not occur in a[0..n-1] otherwise
// returns least i such that x does not occur in a[i..n-1].
// Preconditions: n >= 0 and a[0..n-1] allocated.
// n0 = n
}
```

`n0` is not a variable but a recording of the initial value of the formal parameter `n` to facilitate later assertions.

```
int find(int a[], int n, int x) {
// Purpose: return -1 if x does not occur in a[0..n-1] otherwise
// returns least i such that x does not occur in a[i..n-1].
// Preconditions: n >= 0 and a[0..n-1] allocated.
// n0 = n
U: // x does not occur in a[n..n0-1]
 ...
}
```

The first time execution passes the assertion labeled `U` there are no elements in `a[n..n0-1]`, so we can safely infer that `x` is not among them.

The advantage of `U` as an assertion is that it allows us to state what the algorithm should do: to make `n` as small as possible while keeping the assertion true. Whether `n` can be make smaller depends on whether `a[n-1] == x`. If this is true, then `n` is the value to be returned.

```
int find(int a[], int n, int x) {
// Purpose: return -1 if x does not occur in a[0..n-1] otherwise
// returns least i such that x does not occur in a[i..n-1].
// Preconditions: n >= 0 and a[0..n-1] allocated.
// n0 = n
```

```
U: // x does not occur in a[n..n0-1]
   if (a[--n] == x) return n;
 ...
}
```

Otherwise, we can decrement `n` and yet maintain the truth of `U`. If `n` is 0, then we have in `n` the least `i` of the requirement.

```
int find(int a[], int n, int x) {
// Purpose: return -1 if x does not occur in a[0..n-1] otherwise
// returns least i such that x does not occur in a[i..n-1].
// Preconditions: n >= 0 and a[0..n-1] allocated.
// n0 = n
U: // x does not occur in a[n..n0-1]
   if (a[--n] == x) return n;
   if (n == 0) return 0;
 ...
}
```

We now have truth of `U`. This is expressed by transferring control to a code location where `U` is asserted, which completes the function and its correctness proof.

```
int find(int a[], int n, int x) {
// Purpose: return -1 if x does not occur in a[0..n-1] otherwise
// returns least i such that x does not occur in a[i..n-1].
// Preconditions: n >= 0 and a[0..n-1] allocated.
// n0 = n
U: // x does not occur in a[n..n0-1] and n > 0
   if (a[--n] == x) return n;
   // x does not occur in a[n..n0-1]
   if (n == 0) return -1;
   // x does not occur in a[n..n0-1] and n > 0
   goto U;
}
```

which we rewrite to

```
int find(int a[], int n, int x) {
// Purpose: return -1 if x does not occur in a[0..n-1] otherwise
// returns least i such that x does not occur in a[i..n-1].
// Preconditions: n >= 0 and a[0..n-1] allocated.
// n0 = n
   while (1) { // x does not occur in a[n..n0-1] and n > 0
     if (a[--n] == x) return n;
     // x does not occur in a[n..n0-1]
     if (n == 0) return -1;
     // x does not occur in a[n..n0-1] and n > 0
} }
```

Compare this to Program 13.1: *Find*, where the algorithm was known before writing any code. That there is a difference should not be surprising: in this chapter we did not take an algorithm as starting point, but a specification.

Chapter 20

Stepwise refinement

In Chapter 18 we looked at examples of top-down programming where the higher-level task could be expressed in terms of a procedure call, even though the procedure had not yet been written. The idea of top-down is more widely applicable: even when the task to be tackled cannot yet be neatly wrapped up in a single procedure, why not use a phrase of English as the basis for further progress? This idea has been given the name *stepwise refinement* by Niklaus Wirth*.

Stepwise refinement is essential for a beginning programmer because it shows how to develop an algorithm from the beginning, starting with a blank screen. To illustrate the technique we use it for a program that solves Sudoku puzzles.

Learning to program puzzles can be fun, and is not entirely frivolous. The puzzles treated here are valuable as prototypes of the scheduling problems that are important in, for example, industry, transportation, medical services, and universities.

In this chapter I take the opportunity to introduce another problem-solving technique, which is to exploit a familiar *pattern*. Problems come in families of which the common trait is that their development by stepwise refinement starts in the same way. This common trait is what I call "pattern". Sudoku is the most widely known member of a family sharing a certain pattern. The other members of the family that I present in this chapter are Eight Queens and Knight's Tour.

20.1 A pattern

What is a systematic way to write a program that solves Sudoku puzzles? Such a method should capitalize on the fact that Sudoku shares important features with other puzzles so one starts with a pattern that concentrates on these features and abstracts away the differences among the different kind of puzzles. In this chapter I demonstrate such a pattern suggested by the similarities between Sudoku and the two other puzzles.

* "Program development by stepwise refinement" by Niklaus Wirth. *Communications of the ACM*, vol. 14, no. 4 (April 1971), pp. 221–227.

In each puzzle there is a configuration of items that can be changed subject to the rules of the puzzle.

- In Sudoku the configuration is a nine-by-nine grid of cells, each of which is either empty or filled with one of the digits 1 through 9. The grid is subdivided into three rows and three columns of three-by-three blocks. The digits are placed so that they conform to certain *exclusion rules*: the same digit occurs at most once in each row, column, and block. Initially most of the cells are empty. We make a move towards solving the puzzle by placing a digit in an empty cell so that that digit in that cell satisfies the exclusion rules. If you are smart at picking the next cell to be filled according to the exclusion rules, you only have to keep doing this to end up with a solved puzzle.

- In the eight-queens puzzle one is required to find a way of placing eight queens on an otherwise empty chessboard in such a way that no queen attacks any other queen.

- Given an empty chess board, the problem of the Knight's Tour is to find a sequence of knight's moves starting from the lower left corner in such a way that every square of the board is visited once and not more than once. The problem is defined for n-by-n boards from $n = 3$ upwards.

Eight Queens and Knight's Tour have in common with Sudoku that the puzzle is solved by a sequence of moves, each of which have to obey certain rules. In stepwise refinement we start with an abstract view of puzzles that fits all three of our examples. In this way we can transfer concepts, if not code, from one program to another.

We call a configuration a *state*. A change of state that respects the rules is a *move*. We call a state S' resulting from a move from state S a *successor* of S. To solve the puzzle is to find a sequence of moves that transforms the *start state* to a specified state, the *goal state*. For example, in Sudoku the goal state is characterized by the absence of empty cells. A sequence of moves that transforms the start state to the goal state is a *solution*.

A sequence of moves from the start state is called a partial solution, even though it may not be extendable to a solution. If the puzzle can be solved at all, then the partial solution that is the empty sequence of moves can be extended to a solution. We view a computation that leads to a solution as a sequence of extensions of partial solutions.

This abstraction suggests the following form of code for solving a puzzle.

```
void puzzle() {
// Purpose: print all solutions.
  /* create start state S */
  /* make S the empty partial solution */
  /* extend S */
}
```

The idea behind this pseudo-code is put some of the compilable code in place while the parts of the code yet to be written are described by comments. The entire code is a self-contained unit so that it can take the form of the parameterless function `puzzle()`. The prohibition in `C` against nested functions prevents us from encapsulating the parts of the body as functions.

```
void extend(S) {
// Purpose: print all solutions reachable from S.
  // goal test
    /* if S is a solution, then print S and return */
  // move generator
    /* for each successor S' of S, extend S' */
}
```

The challenge of puzzles arises from the fact that a move may lead to a state from which a solution is not reachable. In such a case it is necessary to *backtrack*; that is, to undo the effect of the last one or more moves and to select a different move, one that may lead to the solution.

In spite of its simplicity, the above algorithm provides this behaviour. After completion of the call `extend(S')` the local variables that ensure that all successors of `S` are generated have the values they had at the time this function was called.

The system of states and moves may be such that a sequence of moves may exist that leads from a state back to that state. The algorithm has no safeguard against moving around such a cycle ad infinitum.

As the Eight-Queens Problem is the simplest of the three, I treat it first.

20.2 The eight-queens problem

However states are defined, the solution state has all eight queens on the board. One could define all states as having this property. The initial state could then be a randomly selected distribution of the eight queens, or some arbitrarily defined state like the one where all queens are on the first row. Such a representation of the puzzle's states has the disadvantage that cycles of states are possible so that the algorithm rarely works.

One could also define states without the requirement that all eight queens are on the board. In that case we can define a move as adding a queen to a board that contains fewer than eight queens. The corresponding definition of the initial state is the one where there are no queens on the board. This representation has the property that, if a solution exists, then it can be reached by eight choices of a square for the additional queen. Such a definition has the advantage that no cycles are possible. Let us adopt this definition.

To minimize the amount of work the algorithm has to do, we minimize the number of states that need to be considered. Without loss of generality we can eliminate all states that have more than one queen in the same row. We can further restrict consideration to states that have k queens in the first k rows. In this way it

is still possible to reach all solutions from the initial state. Furthermore we restrict consideration to states with k queens in such a way that no two queens attack each other. With $k = 0$ this is not a restriction, so that the algorithm remains applicable. Successor of a k-queen state is defined as the result of adding the $(k+1)$-st queen in such a way that the new queen does not attack any of the k queens already placed.

These choices allow us to represent a state by an integer `k` in `0..8` and an array `q[0..7]` such that `q[i] == j` iff the queen in the `i`-th row is in column `j`. The state determines the columns for the first `k` queens; that is, the contents of `q[0..k-1]`. For the initial state we have `k == 0`, with no queen on the board.

```
void eightQueens() {
// Purpose: print all solutions.
  // create start state
  const int N = 8; int q[N], k;
  // make S the empty partial solution
    k=0;
  // extend S
    extend(q, N, k);
}

void extend(int q[], int N, int k) {
// Purpose: to print all solutions reachable
// from state specified in parameters.
// Preconditions: 0 <= k < N and q[0..N-1] allocated.
  // goal test
    if (k == N) {print(q, N); return;}
  // move generator
  for(int j=0; j<N; j++)
    if (!attack(q, k, j)) {
      q[k] = j; // successor of current state
      extend(q, N, k+1);
    }
}
```

The recursive definition of `extend` is essential in making this work. Suppose the `for` statement is halfway and has assigned 4 to `j`. The call `extend(q, N, k+1)` creates its own copy of `j` and typically makes yet another call with *its* own copy. Whether or not these calls result in a solution being printed, eventually `j` is restored with its value of 4, so that the `for` statement under consideration continues with the consequences of assigning 5 to `j`.

It remains to define under what conditions a queen in row `k` and column `j` attacks any of the queens in `q[0..k-1]`.

```
int absVal(int x); // returns absolute value of x
int attack(int queens[], int k, int j){
// Purpose: return 1 or 0 according to whether a queen
```

```
// in [k,j] attack any of those in rows [0..k-1].
// Preconditions: queens[0..k] allocated.
  for(int i=0; i<k; i++) {
  // Does a queen in [k,j] attack the queen row i and column j?
    if (queens[i] == j) /* same column */ return 1;
    if (absVal(queens[i]-j) == k-i)
      /* same diagonal */ return 1;
  }
  return 0;
}
```

Now that only `print` and `absVal` remain to be defined, we can consider the eight-queens problem to be solved.

```
int main() { eightQueens(); }
```

prints all 92 solutions.

20.3 Sudoku

We consider the original Sudoku puzzle, where a 9-by-9 grid of cells has to be filled with digits $1, \ldots, 9$ in such a way that the exclusion rules are satisfied: each digit occurs once in each row, once in each column, and once in each of the non-overlapping 3-by-3 blocks into which the the grid can be decomposed. In the initial state part of the solution is given; the other cells are empty. Each move consists of placing a digit in an empty cell in such a way that the exclusion rules are satisfied.

We represent the state by an array of digits in which an empty cell is indicated by its containing the digit 0.

In the eight-queens problem our representation was such that there was a unique initial state. Sudoku is different in that there are as many initial states as there are Sudoku puzzles.

For every initial state there are zero or more ways of filling all empty cells while conforming to the exclusion rules. A Sudoku puzzle has an initial state that can be completed in exactly one way.

```
 1 void sudoku() {
 2 // Purpose: to print all solutions to the Sudoku puzzle
 3 // specified in the start state.
 4   // create start state
 5     const int n = 3; // blocksize
 6     int s[] = {
 7       0,0,0, 7,0,0, 2,1,0,
 8       0,0,0, 0,5,9, 0,4,3,
 9       0,0,0, 0,0,8, 9,0,0,
10
11       8,0,2, 0,0,0, 0,0,0,
```

```
12       6,5,0, 0,1,0, 0,2,4,
13       0,0,0, 0,0,0, 5,0,7,
14
15       0,0,7, 2,0,0, 0,0,0,
16       9,1,0, 5,8,0, 0,0,0,
17       0,8,4, 0,0,6, 0,0,0
18                };
19     int k;
20     // make (s, n, k) partial solution
21     k = 0;
22     { // extend partial solution
23       extend(s,n,k);
24 } }
```

"Solving" the puzzle is a special case of extending a partial solution to a full solution.

```
 1 void extend(int s[], int n, int k) {
 2 // Purpose: print all solutions that reachable from current state.
 3 // Precondition s[0..n*n*n*n-1] allocated and 0 <= k <= n*n*n*n.
 4   int n2 = n*n, n4 = n2*n2;
 5   // the n2 by n2 Sudoku grid is stored in s[0..n4-1]
 6   // goal test
 7     if (k == n4) { print(s, n); return; }
 8   // k < n4
 9   // move generator
10     // s[k] is cell in row k/n2 and column k%n2
11     if (s[k] != 0) // cell occupied; content checked OK already
12       extend(s, n, k+1);
13     else { // cell empty
14       for(int x = n2; x>0; x--) {
15         if (ok(s, n, x, k)) { // x ok in cell s[k]?
16           s[k] = x;
17           extend(s, n, k+1);
18           s[k] = 0; // restore to empty
19 } } } }
```

Function `ok` determines whether a proposed value `x` is compatible with the existing cell contents.

```
 1 int ok(int s[], int n, int x, int k) {
 2 // Purpose: return 0 or 1 according to
 3 // whether x in s[k] satisfies exclusion rules.
 4 // Preconditions: s[0..n*n*n*n-1] is allocated,
 5 // 1 <= x <= 9, and 0 <= k < n*n*n*n.
 6   int n2 = n*n, n4 = n2*n2;
 7 // the n2 by n2 Sudoku grid is stored in s[0..n4-1]
```

```
 8   int i = k/n2, j = k%n2;
 9   // s[k] is cell in row i and column j
10   for(int p = 0; p < n2; ++p)
11      if ((x == s[i*n2+p]) || (x == s[p*n2+j]))
12        return 0; // conflict in row or column
13   for(int p = n*(i/n); p < n*(i/n) + n; ++p)
14     for(int q = n*(j/n); q < n*(j/n) + n; ++q)
15     // [p][q] is in same block as [i][j]
16       if (x == s[p*n2 + q]) // conflict in block
17         return 0;
18   return 1; // no conflict in row, column, or block
19 }
```

To print all solutions:

```
int main() { sudoku(); }
```

On my laptop there is no noticeable delay between starting this program and the printing of the solution for any of the Sudoku's that I have tried. These include ones labeled as "hard" or "diabolical". Yet you can see from the code that the solution is obtained by brute force. This seemed necessary to avoid making the code more complex. A human Sudoku solver takes the opposite approach. She spends a lot of time finding a cell for which only one value is possible. In this way backtracking is avoided, or at least minimized.

20.4 Knight's Tour

A chessboard is an array of squares with eight rows and eight columns. Let's number columns by $x = 0, \ldots, 7$ and rows by $y = 0, \ldots, 7$. A knight is a piece that can move from a square (x, y) to $(x \pm 1, y \pm 2)$ or to $(x \pm 2, y \pm 1)$ in so far as these squares are on the board.

Of course we should not take for granted that the simplistic brute-force search will also work for the Knight's Tour. Fortunately, the problem scales down as well as up: it makes sense for any size of board from three-by-three upwards. Below we consider four rows and columns.

```
00 01 02 03       16 03 12 16
04 05 06 07       11 06 09 02
08 09 10 11       08 01 04 13
12 13 14 15       05 10 07 00
```

The two-digit numerals in the array on the left are an enumeration of the squares of the four-by-four board. Rows are numbered from the top down; columns from left to right. The enumeration has the property that the n-th square in the enumeration is in row $n/4$ and in column $n\%4$. This is chosen for ease of printing.

The two-digit numerals in the array on the right represent a sequence of moves of a knight using the enumeration on the left to identify the squares visited. The

sequence of squares visited is denoted 00, 01, 02, 03, and so on. As shown, the first square visited is in the lower right, the square with coordinates $x = 3$ and $y = 3$, which is the square enumerated as 15 in the array on the left. Note that the knight has no square to go to from step 13. Thus the sequence of moves shown in the right array cannot be extended to a solution. The two unvisited squares are marked 16, one more than the highest possible move number on a four-by-four board.

We represent the path of the thirteen-step partial solution by the array `p[0..15]` in such a way that step `k` is in `p[k]` according to the above enumeration.

```
index 00  01  02  03  04  05  06  07  08  09  10  11  12  13  14  15
    p 15  09  07  01  10  12  05  14  08  06  13  04  02  11  16  16
```

The chosen representation of squares and of partial solutions suggests the following definition.

```
 1 void knightsTour(int n) {
 2 // Purpose: print solutions of the Knight's tour puzzle
 3 // specified in this function.
 4 // Preconditions: None.
 5   // create start state
 6   const int n2 = n*n;
 7   int p[n2]; for(int i=0; i<n2; i++) p[i] = n2;
 8   // make S the empty partial solution
 9   // p[0] = n2-1; // knight starts in lower right corner
10   p[0] = 0; // knight starts in upper left corner
11   int k = 1;
12   // extend S
13   extend(p,n,k);
14 }
```

The function `extend` takes as parameters an array `p[0..n*n-1]` and an integer `k` such that `p[0..k-1]` contains a partial solution. The goal of the function is to extend this partial solution, if possible, to a complete solution, and to print it. The length of a complete solution is `n*n`. This fact provides a convenient test for completeness of a partial solution. This is used in the following definition of `extend`.

```
 1 void extend(int p[], int n, int k) {
 2 // Purpose: to print all solutions reachable from
 3 // partial solution contained in p[0..k-1].
 4 // Preconditions: p[0..n*n-1] allocated and k <= n*n.
 5   // goal test
 6   if (k == n*n) { print(p,n); return; }
 7   // move generator
 8   // k < n*n
 9   int sq = p[k-1];
10   // find successor of sq
11   for (int mvNum = 0; mvNum < 8; ) {
```

```
12     int succ; // for successor to sq
13     succ = sq;
14     next(&succ, &mvNum, n);
15     if (find(p, k, succ) == -1) {
16       p[k] = succ;
17       extend(p, n, k+1);
18       p[k] = n*n; // Reset to original value.
19 } } }
```

The function `next` handles the intricacies of the knight's move, including making sure that it does not jump off the board.

```
 1 int next(int* sq, int* i, int n) {
 2 // Purpose: if there is a successor
 3 // of *sq with move number *i or greater, but less than 8,
 4 // write that successor in *sq and its move number in *i.
 5 // Preconditions: *sq is the current square and all moves from
 6 // that square with move number less than *i have been tried.
 7 // n is the number of rows and columns of the board.
 8   static int dxdy[8][2] = // 8 candidates for knight's moves
 9     { {-2,-1}, {-2,+1}, {-1,-2}, {-1,+2},
10       {+1,-2}, {+1,+2}, {+2,-1}, {+2,+1}
11     };
12   int x = *sq/n, y = *sq%n;
13   while (*i < 8) {
14     if (0 <= x+dxdy[*i][0] && x+dxdy[*i][0] < n &&
15         0 <= y+dxdy[*i][1] && y+dxdy[*i][1] < n
16        ) // valid successor found
17       { *sq = (x+dxdy[*i][0])*n + y+dxdy[*i][1];
18         (*i)++; return 1;
19       } else (*i)++;
20   } // *i == 8
21   return 0;
22 }
```

The function `find` is defined as follows.

```
1 int find(int a[], int n, int x) {
2 // Purpose: return i such that a[i] == x, if it exists;
3 // return -1 otherwise.
4   for (int i = 0; i < n; i++) if (a[i] == x) return i;
5   return -1;
6 }
```

Finally, the indispensible function `print`.

```
1 void print(int p[], int n) {
2 // Purpose: print contents of p[] as n-by-n board.
```

```
 3 // Precondition: n > 0 is number of rows and columns.
 4 // p[0..(n*n)-1] is allocated.
 5 // Limit number of solutions to be printed:
 6 static int COUNT = 10; assert(COUNT-- > 0);
 7   int n2 = n*n;
 8   for(int i=0; i < n2; i++) {
 9     if (i%n == 0) printf("\n");
10     printf("%02d ", find(p, n2, i));
11   }
12   printf("\n");
13 }
```

This program shows that no solution exists for $n = 4$. It quickly finds the first solutions for n equal to 5, 6, and 7. With $n = 8$ it depends on where you start. From the lower right corner I have not found it worthwhile to run it long enough to get any solution. From the upper left corner it takes a minute or so before it starts printing solutions.

Scheduling From an abstract point of view puzzles are similar to the kind of schedules that need to be made when running a hospital, an airline, or the examinations in a university. In practice schedules are produced by a combination of sophisticated techniques from operations research and massive amounts of computing. Yet in practice human input can be essential, as illustrated by the career of Henry and Holly Stephenson who have produced schedules for Major League Baseball in preference to competitors using state-of-the-art pure computer methods. The Stephensons used homebrew software of which the output was further optimized by hand[†].

[†]The Stephensons got the contract every year from 1980 to 2004. For the beginning of this period, see "Popes, Blizzards And Walleyed Pike" by Albert Kim. *Sports Illustrated*, April 8, 1991.

Appendices

Appendix A

Table of operators

Even in a complex expression, its structure can be recovered from associativity and precedence only, without any help from parentheses. For example, how much is

```
u/w*-u+v/u+v-x- -x/v
```
(A.1)

when the integer variables `u`, `v`, `w`, and `x` have values 1, −2, 3, and −4 respectively?

Of the operators in this expression, unary minus has the highest precedence. The given expression therefore has the same value as

```
u/w*(-u)+v/u+v-x- (-x)/v
```

We replace the parenthesized sub-expressions by their values

```
u/w*(-1)+v/u+v-x- 4/v
```

As there are now operators with two levels of precedence, this expression has the same value as

```
(u/w*(-1))+(v/u)+v-x- (4/v)
```

We use left-associativity within the parenthesized sub-expression

```
((u/w)*(-1))+(v/u)+v-x- (4/v)
```

Further applications of associativity or precedence yield, successively,

```
(0*(-1))+(-2)+v-x-(-2)
0+(-2)+v-x-(-2)
(-2)+v-x-(-2)
(-4)-x-(-2)
0-(-2)
2
```

The specific algorithm for recovering the structure from associativity and precedence depends on the compilation method used by the implementation. Such an algorithm will evaluate any legal sequence of operands and operators that you throw at it.

Should the programmer know such an algorithm? I think not. I think a programmer should never write an expression like (A.1) where one needs an algorithm to know what it means. The meaning of any expression, however complex, can be made obvious by using enough parentheses. Don't forget that parentheses are really cheap for the compiler to process.

Tokens	Operator	Class	Prec.	Associates
a`[`*k*`]`	subscripting	postfix	16	left-to-right
f`(...)`	function call	postfix	16	left-to-right
`.`	direct selection	postfix	16	left-to-right
`->`	indirect selection	postfix	16	left-to-right
`++ --`	increment decrement	postfix	16	left-to-right
`(`*type name*`){`*init*`}`	compound literal	postfix	16	left-to-right
`++ --`	increment decrement	prefix	15	right-to-left
`sizeof`	size	unary	15	right-to-left
`~`	bitwise not	unary	15	right-to-left
`!`	logical not	unary	15	right-to-left
`- +`	arithmetic negation, plus	unary	15	right-to-left
`&`	address of	unary	15	right-to-left
`*`	indirection	unary	15	right-to-left
`(`*type name*`)`	cast	unary	14	right-to-left
`* / %`	multiplicative	binary	13	left-to-right
`+ -`	additive	binary	12	left-to-right
`<< >>`	left and right shift	binary	11	left-to-right
`< > <= >=`	relational	binary	10	left-to-right
`== !=`	equality inequality	binary	9	left-to-right
`&`	bitwise and	binary	8	left-to-right
`^`	bitwise xor	binary	7	left-to-right
`|`	bitwise or	binary	6	left-to-right
`&&`	logical and	binary	5	left-to-right
`||`	logical or	binary	4	left-to-right
`?  :`	conditional	ternary	3	right-to-left
`= += -= *=`	assignment	binary	2	right-to-left
`/= %= <<= >>=`	assignment	binary	2	right-to-left
`&= ^= %= <<= >>=`	assignment	binary	2	right-to-left
`,`	sequential evaluation	binary	1	left-to-right

Table A.1:
The operators of C and their properties. The column heading "Prec." stands for the precedence of the operator. From *C: a Reference Manual* by S.P. Harbison and G.L. Steele, 5th edition, Prentice-Hall, 2002.

Appendix B

Command-line parameters

So far we have relied on input to programs to be supplied by functions that are called from within bodies of functions. This is not the only possibility.

To execute a program, it has to be activated by the operating system. The result of compilation and linking is that the operating system recognizes a new command, which is the program. The program is then started by typing the name of this command on the command line. A good opportunity to transmit a small amount of information to the program is to type it after the name of the command, in the same way it is done with other commands to the operating system.

Suppose we want to use the computer to compute the sum of a short sequence of numbers. So far we would have started the program and then the code would prompt the user to type the numbers. It is also possible to write the program so that the numbers come right after the program command. An interaction with the program would look like this:

```
%add .123 .234 .345 .456 .567 .678
2.403000
```

where `%` is the operating systems prompt. We say that in such a situation the input has been provided by *command-line parameters*.

How do we get code to read command-line parameters? So far we have always defined the function `main` to have no parameters. When the program is intended to take command-line parameters, function `main` is given two parameters. The first has type `int`; the second has as type array of pointers to characters. The first parameter is given as value one more than the number of intended command-line parameters. (That number can be zero.) The second parameter is given as value an array containing the command line parameters, if any. Conventionally, the parameters for `main` are given the names `argc`, for "parameter count", and `argv`, for "parameter vector".

The elements of the array `argv` are strings. More precisely, each element is a pointer to the first character of the string. `argv[1]` points to the first character of the first command-line parameter (if there is one), `argv[argc-1]` points to the

first character of the last parameter, (remember that `argc` is one more than the number of parameters). `argv[argc]`, the last element of `argv`, is the null pointer. This leaves `argv[0]` unaccounted for: it points to the first character of the string that names the command by which the operating system invokes the program.

```
1 #include <stdio.h>
2 #include <stdlib.h> // for atof
3
4 int main(int argc, char *argv[]) {
5   double sum = 0;
6   for(int i=1; i < argc; ++i) sum += atof(argv[i]);
7   printf("%f\n", sum);
8 }
```

Figure B.1: *Add Parameters*, a program that adds numbers specified on the command line.

Program B.1: *Add Parameters* does the job.

Notice that the command-line parameters are made available as *strings*. To be interpreted as decimal fractions, which is what we want here, we pass them to the function `atof` ("a-to-f", for "alphanumeric to float") from the library `stdlib`, which determines their equivalent floating-point value.

A command line consisting of a command followed by parameters is just one example of the common practice of representing text as an array (in this case `argv`) of strings. Any text can be represented as array of strings representing the words or the lines in the text.

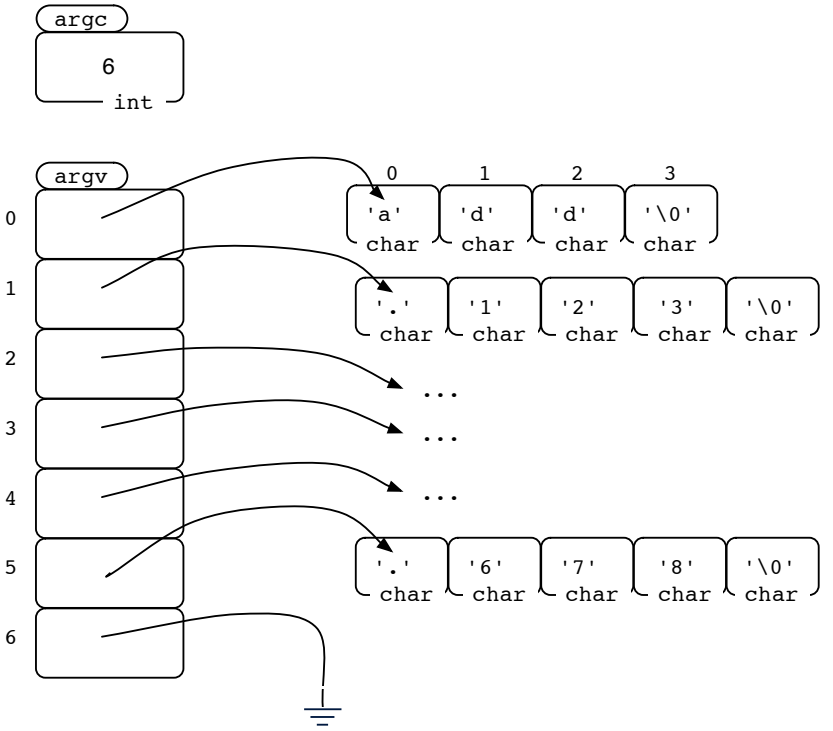


Figure B.2: Data structure for command line `add .123 .234 .345 .456 .567 .678`.

Appendix C

The C standard library

I may have given the impression that one always starts a program from scratch. This is misleading. For example, every program at least needs some input or output. Even Program 2.1 needed both. Input and output do not happen by magic: to be able to connect the I/O statements in a program to the computer and to the peripheral hardware is a sizeable and challenging programming task. Once this is done, it should be possible to let many programs benefit by building on this effort.

Re-use of code I/O is an example of code that is needed by many programmers. We include in the present survey facilities other than I/O that are often needed and are included in the C standard library.

In general, *library* is understood as *a collection of program components that can be re-used*. "Program components" suggests primarily functions. But other things can also benefit from re-use. For example, it may require quite a bit of research to discover that on your system the greatest finite double-length floating-point number is `1.7976931348623157E+308`. This, or whatever it is, is conveniently available according to the C standard library under the standard name `DBL_MAX`. Accordingly, we tend to speak of library "facilities", rather than specifically restrict ourselves to functions. Over time, the number of such facilities, major or minor, connected or not, runs in the hundreds, representing a considerable quantity of code, often hundreds of times longer than the program, which may only use `printf`.

Of course we don't want to load and link to all of the standard library's executable when we only need a single library facility. This is the reason we include only the library containing the required item.

Each library comes with a header file, which contains just enough so that, by including it, we can separately compile our own program and then link with the required pre-compiled libraries. Thus, `DBL_MAX` is in a library accessed by `#include <float.h>`. For `printf` it is `stdio.h`.

C.1 Overview of the C standard Library

Standard C comprises both a language standard and a set of libraries. Below is a list of the headers. As you can surmise, the size and degree of focus varies among these libraries. There are large and well-focused libraries, such as the ones concerned with time, input and output, and localization. There are also facilities that are not so easily classified, and one does not want to leave these floating around on their own in hundreds of little bits. These unclassifiable ones are herded together in grab bags such as `stddef.h` and `stdlib.h`.

Here are the fifteen libraries identified by their header files.

`assert.h` Enables the use of executable assertions.

`ctype.h` Tells whether a character is a digit, a letter, ... Includes functions such as `isdigit`, `isalpha`, `isspace`, ...

`errno.h` Standardizes the reporting and handling of errors.

`float.h` Floating-point number systems may differ in the precision they provide, how large the exponent can be, among other things. Such characteristics can be found here.

`limits.h` C programs should be able to run on computers with different architectures. These can be described by such numbers as the minimum and maximum values of `char`, `int`, `short`, `long`, and so on. This makes it necessary for C code to find out what the relevant data are on the current machine.

`locale.h` Facilities that support internationalization are collected in this library. The best introduction to the topic is given by the following remark*

> *If one lives in the United States, it's easy to forget that English is not the only language, ASCII is not the only character set, $ is not the only currency symbol, dates can be written with the day first, times can be based on a 24-hour clock, and so on.*

If one lives in Los Angeles or in Canada it is *not* easy to forget that there are languages other than English. Locale-dependent conventions are characterized and made accessible under this header.

`math.h` Lots of stuff here: trigonometric functions, logarithms, powers, and some odds and ends.

* "The Practice of Programming" by Brian W. Kernighan and Rob Pike. The quote is a good excuse for bringing this delightful book to your attention, very much recommended as a companion to the one you are reading now.

`setjmp.h` In C `goto` statements are restricted to act within a function. Sometimes it is desired to jump far away. A set of functions to make this possible is collected under this header.

`signal.h` A *signal* is an unforeseen event that occurs during execution of a program. For example: division by zero, or the user pressing `ctl-c`. When code needs to be written for the proper handling of such events, this header provides a standardized set of functions.

`stdarg.h` In this book we have only defined functions with a fixed number of parameters. However, we have *used* a function with a variable number of parameters (`printf`). Such functions can be defined in C, but need the help of some functions found in this header.

`stddef.h` A miscellaneous collection.

`stdio.h` I/O facilities.

`stdlib.h` Another miscellaneous collection. One often needs this library for random permutation of an array (see Section C.5, the *Fisher-Yates Shuffle*) and for random numbers generally.

`string.h` String handling: we often need to determine the length of a string, compare or concatenate two strings, copy a string. That, and more, can be found here.

`time.h` Suppose you want to start with now and then go back 2000 months, or advance 2 billion seconds. Standard functions can be found here that make this easy, as well as many less bizarre operations.

C.2 Formatted I/O

The function `printf` is unusual in that it allows a variable number of parameters. The first parameter is a string. It is this string that is output, subject to such modifications as will now be described. The string contains zero or more codes, *formatting codes*, that are substituted by the same number of parameters following the first paramter of `printf`. It is the string, with the substitutions in place, that is printed. Each code determines the format in which the corresponding parameter is printed.

The function `scanf` is the input counterpart to `printf`. Its first parameter is the string containing the formatting codes. The part of the string that is not a code has to be present in the input. No action is taken as a result of that part of the string. Each code determines the format expected for the corresponding part of the input. That part of the input is scanned according to the format specified. The resulting

value becomes the value of the actual parameter. To be any use, that parameter has to be a pointer, like any output parameter. `scanf` returns the number of items that were input as a result of the call.

Program C.1: *Formatted Input and Output*, illustrates that `scanf` and `printf` are to some extent each other's mirror images. With input `09/08/07`, the output

```
0 #include <stdio.h>
1
2 int main() {
3   int dd, mm, yy;
4   /* input date in format dd/mm/yy */
5   int count = scanf("%d/%d/%d", &dd, &mm, &yy);
6   printf("number of items input: %d\n", count);
7   printf("%02d-%02d-%02d\n", mm, dd, yy);
8   /* output date in format mm-dd-yy */
9 }
```

Figure C.1: *Formatted Input and Output*, an example of `printf` and `scanf`. Note that the slashes specified in the pattern of the `scanf` call have to be present in the input string.

`08-09-07` results. There are three occurrences of the same code `%02d` in

```
printf("%02d-%02d-%02d\n", mm, dd, yy);
```

Each specifies that the corresponding integer value is to be printed in a field of width 2 padded if necessary with zeros.

Program C.2: *Long Print String*, contains an example with a more elaborate print string, where the conversion codes `%d`, `%o`, `%x`, and `%f` are used.

The output could be the following:

```
as an unsigned in decimal: 90
as a character: Z
as an unsigned in octal: 132
same in hex: 5a
negated, as an integer: -90
negated, as unsigned: 4294967206
as a flpt num in default format: 0.014925
as a flpt num in exponential format: 1.492537e-02
as a flpt num in wide field: |        0.01492537313432835792|
same, left justified: |0.01492537313432835792        |
```

As a first approximation, the output is the string that is the first parameter of `printf`. As far as the C compiler is concerned, this first parameter is one long

```
00 #include <stdio.h>
01
02 int main() {
03   int x;
04   scanf("%d", &x);
05   printf("\
06 as an unsigned in decimal: %u\nas a character: %c\n\
07 as an unsigned in octal: %o\n\
08 same in hex: %x\n\
09 negated, as an integer: %d\n\
10 negated, as unsigned: %u\n", x, x, x, x,-x, -x);
11    double y = 1.0/67.0;
12   printf("\
13 as a flpt num in default format: %f\n\
14 as a flpt num in exponential format: %e\n\
15 as a flpt num in wide field: |%30.20f|\n\
16 same, left justified: |%-30.20f|\n", y, y, y, y);
17 }
```

Figure C.2: *Long Print String*, an example with an elaborate print string. Note the possibility of a string being distributed over several lines by means of a backslash followed immediately by a newline. This has to do with the preprocessor; see Appendix D.

line. However, to allow more convenient layout, when entering the source text one can interrupt a line by inserting the backslash character followed by pressing the "enter" key.

The above examples contain but a small sample of the codes available for specifying formats for `scanf` and `printf`. They constitute a "little language", as the people at Bell Labs liked to call this system of codes. The best known of the several other little languages they developed is the one for regular expressions.

The details are complicated: I need to consult reference material every other time I use formatted I/O. See Section 4.6.2.

C.3 Internal I/O

The functions `num` and `numRev` in Program 18.3 output the digits of a given integer. They combine the algorithm for determining the digits with a decision about how to output them. The code for the algorithm is more widely usable if it is isolated from concern with output. Strings are useful for effecting such a separation of concerns: the algorithm can be written so that the digits are placed in a string. Another function can then hide the decision about how to output the resulting string. We

can think of strings as a vehicle for *internal* output: output to memory, which is where the strings reside. Strings can be used similarly as internal alternative to input from external sources.

Program C.3, *Internal Output*, is an example of code where strings are used for internal output. The program outputs the digits of an integer. Unlike Program 18.3, here the code for the conversion of the integer to digits is separated from the code that outputs the digits. The function `main` creates a string of length 100. This

```
00 #include <stdio.h>
01
02 void reverse(char s[], int n);
03 // reverses s[0..n]
04 void int2str(int n, int b, char s[]) {
05 // write b-base digits of n in reverse order
06   int k = 0;
07   while (n>0) { s[k++] = '0' + n%b; n /= b; }
08   reverse(s, k-1);
09   s[k] = '\0';
10 }
11 int main() {
12   char s[100];
13   int2str(54321, 8, s);
14   printf("%s\n", s);
15 }
```

Figure C.3: *Internal Output*, printing the digits of a number via a string.

is a somewhat arbitrary number: big enough to hold any anticipated numeral, yet not wastefully large. Remember that 32 bits is already large for a variable of type `int`; this corresponds to a numeral of about ten decimal digits. The call `int2str(12345, 10, s)` ("integer to string") places the digits of the numeral for the given integer to the base ten in the string `s`. The natural way to produce the digits is in reverse order. Rather than to strain and try to get them to come out in the right order, we let it happen and then reverse the resulting string.

There is a much easier way to write `int2str` if we don't need the flexibility of an arbitrary numeral base. Suppose we want decimal numerals. Consider the statement

```
printf("%d ", n = 12345);
```

The compiler encounters the string `"12345"` in the source code and converts to the internal integer representation, which is, in the case of 32-bit integers:

```
00000000000000000011000000111001
```

When executing this statement, the internal representation is converted back to the string `"12345"`. So the library function `printf` already has the capability of `int2str` for the decimal case. But in `printf` the algorithm for producing the digits is tied up with output. Fortunately the same library has a function `sprintf`, which is the internal-output version of `printf`. We can use it to implement `int2str` for the decimal case:

```
void int2str(int n, char s[]) {
  sprintf(s, "%d", n);
}
```

Similarly, `scanf` has an internal-input version called `sscanf`.

C.4 File I/O

So far all our input has been taken from "Standard Input"; all our output went to "Standard Output". What are these mysterious entities? Concretely, they are *files*, which are, from the point of view of C, sequences of characters. In C programs, these files are referred to as `stdin` and `stdout`, respectively.

Strictly speaking, we have been doing "file I/O" ever since our first program. We didn't notice because `printf` and `scanf` imply by default that the files to be written to or read from are `stdout` or `stdin`, respectively. The more general versions are `fprintf` and `fscanf`, also made available from the standard library by the line `#include <stdio>`. `fprintf` and `fscanf` are called the same way as `printf` and `scanf`, except that there is an additional first parameter specifying the file. Thus `printf(...)` is equivalent to `fprintf(stdout, ...)`, while `scanf(...)` is equivalent to `fscanf(stdin, ...)`.

In addition to these files, there is also `stderr`, to which we can write error messages.

There are many situations where a C program needs to be able to read input from, or to write output to, files other than `stdin`, `stdout`, and `stderr`. This is what is usually understood as "file I/O". The general process for input is to create a *file pointer* and to cause it to point to the file that contains the intended input. This happens as a result of "opening" the file for reading. The function that makes this happen is `fopen`, which is provided by the library `stdio`. This function takes as parameters two strings: the first is the name of the file as known to the operating system; the second is `"r"`, which stands for "read". The result returned by `fopen` is a file pointer. The program accesses this file via that pointer. `stdin`, `stdout`, and `stderr` are file pointers. They are standard in the sense that they are available without having to be created by a call to `fopen`.

For output the file specified by the program need not exist. If it does not, then the operating system creates, if possible, a file with the specified name. If the file does exist, then opening the file for output implies a choice. This choice is whether to replace the existing file content with the output written by the program, or whether to append this output to the existing file contents. These two options are realised by calling `fopen` with `"w"` or `"a"` as second parameter, respectively.

For "file pointer" to make sense, there has to be a suitable type for the pointer variable to point to. This type is `FILE`, and it is defined by the `stdio` library. File pointers are often called "file *handles*". As these handles are usually a scarce resource, it is wise to free them for other use as soon as possible. This is done by the library function `fclose`.

Program C.4: *File Concatenation*, shows file I/O in action. Supposing this program is compiled and loaded to become the command `cat`, then, e.g., the command line

```
cat temp0 temp1 temp2 < temp3
```

will cause `temp0` to contain the result of concatenating `temp1`, `temp2`, and `temp3`. This command line may need additional explanation. For the `cat` command, the command-line arguments are `temp0`, `temp1`, and `temp2`, up to the `<` symbol. After placing the remaining command-line arguments into `temp0`, Program C.4 continues by adding standard input to `temp0`. In this example, it is `temp3` that acts as standard input.

```
 1 #include<stdio.h>
 2
 3 void fcopy(FILE* in, FILE* out) {
 4   int ch;
 5   while ((ch = getc(in)) != EOF) putc(ch, out);
 6 }
 7 int main(int argc, char* argv[]) {
 8   FILE *in, *out;
 9   out = fopen(argv[1], "w");
10   if (out == NULL) {
11     fprintf(stderr, "%s: can't open %s\n", argv[0], argv[1]);
12     return 1;  // abnormal return from function main
13   }
14   for (int i=2; i<argc; ++i) {
15     in = fopen(argv[i], "r");
16     if (in == NULL) {
17       fprintf(stderr, "%s: can't open %s\n", argv[0], argv[i]);
18       return 1;  // abnormal return from function main
19     }
20     fcopy(in, out); fclose(in);
21   }
22   fcopy(stdin, out); fclose(out);
23 }
```

Figure C.4: *File Concatenation*, A program that creates a file specified in the first command-line parameter and places in it the result of concatenating the files, if any, following the first parameter, followed by the standard input.

Lots of things can go wrong when a C program tries to open a file for reading or appending. In the example, the putative filename retrieved from the command line may not exist, or at least not in the directory where the runtime system looks for a file of that name. When the C program tries to open a file for writing, then the named file may not exist. In that case, the runtime system will ask the operating system to create a file with the specified name. This may not be possible for any of several reasons.

In cases like this, the function `fopen` returns the null pointer. If the program would try to follow a null pointer, then the operating system has to kill its execution in self-defense. To prevent this, Program C.4 prints a diagnostic message and terminates itself. The message is written to the standard file `stderr` (using `fprintf`), rather than to `stdout` (using `printf`). This is done in case standard output is redirected somewhere else than the screen. In such a situation the message would not be noticed and might mess up something that does not expect such output. The advantage of `stderr` is that even when standard output is redirected, text written to this file still appears on the screen.

C.5 The Fisher-Yates shuffle

To test an array program it can be useful to be able to randomly permute the contents of an array of length n. This means that the contents of the array should become any of its $n!$ permutations *with equal probability*. Fisher and Yates described[†] a pencil-and-paper method to do this. Function `randPerm` in Program C.5 is adapted from the one published[‡] by D. Knuth.

Here's how to randomly permute an array. With a random number generator select `i` from the range `0..n-1`. Exchange `a[n-1]` and `a[i]`. Next, randomly select `i` from the range `0..n-2`. Exchange `a[n-2]` and `a[i]`. And so on, until we end by selecting `i` from the range `0..1` and exchanging `a[1]` and `a[i]`. Note that at every step there is a chance that nothing changes because an element is exchanged with itself. This is as it should be. See Program C.5.

[†]Example 12, *Statistical Tables*, by R. A. Fisher and F. Yates, London, 1938

[‡]Algorithm 3.4.2P in *The Art of Computer Programming*, vol.2, by Donald E. Knuth, Addison-Wesley, 1969.

```
 1 #include<stdio.h>
 2 #include<stdlib.h>
 3
 4 void swap(int a[], int i, int j);
 5 // swaps a[i] and a[j]
 6 void randPerm(int a[], int n) {
 7 // Purpose: randomly permute a[0..n-1]
 8 // Preconditions: n >= 0 and a[0..n-1] allocated
 9 // a0[0..n-1] == a[0..n-1]
10   int j, i = n-1;
11   while (i>0) {
12     j = (rand()/(double)RAND_MAX)*(i+1);
13     // 0 <= j <= i with equal probability
14     swap(a, i, j);
15     // a[i..n-1] is random selection a0[0..n-1]
16     --i;
17   }
18 }
```

Figure C.5: *Random permutation*, a function that replaces the contents of an array by any of its permutations with equal probability by means of the Fisher-Yates shuffle.

Appendix D

The preprocessor

All our programs are written in a language loosely indicated as "C". A first crack in this fiction appeared when we started to scrutinize directives. We found that the line

```
#include <stdio.h>
```

is an instruction to replace that line by a header file. Such an instruction, called a "directive", is not part of the C language.

If C is the language that the compiler understands, then the directives (which have disappeared by the time the source code gets to the compiler) belong to another language, the preprocessor language. This language is confined to lines that start with `#` (possibly preceded by whitespace).

So far the only non-C lines we encountered were those that started with `#include`. In addition to file inclusion, the preprocessor has other uses. The more common of these are the removal of comments and the processing of "macros". Other uses include splicing interrupted lines, merging juxtaposed strings, as well as replacing trigraphs by their equivalents. These uses of the preprocessor will be briefly touched upon in the remainder of this appendix.

D.1 Macros

A line that starts with `#define` is a *macro* definition. After `#define` comes a name, typically followed by text. The preprocessor replaces the name by the text. If, untypically, that text is missing, then any occurrence of that name is replaced by nothing.

So far I have described macro definition without parameters. Even with parameters, a macro is no more than a simple-minded version of a function. Accordingly, a macro definition is best introduced by a function definition that is simplistic in that its body consists of a return statement only. For example the function

```
int absdiff(int a, int b) { return a>b ? a-b : b-a; }
```

is such a definition. An expression such as `absdiff(i, j)` is replaced by the expression in the return statement where the formal parameters get as values the values of the actual parameters.

A macro definition is a simple version of such a simplistic function definition. Like a function definition, a macro definition consists of a header followed by a body. The macro header is like a function header except that types are omitted. Because of its limited scope, the `return` keyword is superfluous and is omitted. In the following macro definition I have tried to get as close as possible to its function counterpart:

```
#define ABSDIFF(a, b) (a)>(b) ? (a)-(b) : (b)-(a)
```

Simplistic though macro definitions may be, there seems to be no end of subtleties that can arise in their use. All I try to do in this section is to give some idea of the possibilities provided by macros by the example program below. As the subject is complex, it is best to turn to a professional reference* for more information.

```
00 #include <stdio.h>
01
02 #define PI 3.1415926535
03 #define MULT(x, y) x*y    // maybe not what you want
04 #define ABSDIFF(a, b) (a)>(b) ? (a)-(b) : (b)-(a)
05
06 int absdiff(int a, int b) { return a>b ? a-b : b-a; }
07
08 int main() {
09   int i = 3, j = 4;
10
11 //                                          output
12   printf("%g\n", PI);                      //3.14159
13   printf("%d\n", MULT(2+3, 4));            //14
14   printf("%d\n", absdiff(i, j));           //1
15   printf("%d\n", ABSDIFF(i, j));           //1
16   printf("%d\n", absdiff(i, absdiff(j, i))); //2
17   printf("%d\n", ABSDIFF(i, ABSDIFF(j, i))); //2
18   int p = i, q = j;
19   printf("%d\n", absdiff(++i, --j));       //1
20   printf("%d\n", ABSDIFF(++p, --q));       //3
21 }
```

Figure D.1: *Macro Demo*, a program showing some possibilities and pitfalls of macros.

Consider Program D.1: *Macro Demo*.

*I tend to use "C, a Reference Manual" by Samuel P. Harbison III and Guy L. Steele, Jr.

line 12 As the macro `PI` has no parameter, its name is merely replaced by what comes after the name. This is a very common use. As the early versions of C had no `const` qualifier, this was the only way to avoid the plague of "magic numbers". C has now a better way, but many programmers seem reluctant to adopt it.

line 13 It could be that the intention of

```
#define mult(x, y) x*y
```

is to get whatever replaces `x` multiplied by whatever replaces `y`. In this call `x*y`, the body of the macro, results in the expression `2+3*4`, the evaluation of which does not give the result of multiplying `2+3` by `4`. This is why one would see, in a case like this, the macro definition written as

```
#define mult(x, y) (x)*(y)
```

Always put parentheses around a macro's formal parameters: you never know what text is going to replace them.

lines 14 and 15 `ABSDIFF` is a typical example of a macro, where the motivation is to replace a function call by more efficient code. During execution, the function call `absdiff(i,j)` is processed by copying the values of `i` and `j` to the formal parameters `a` and `b` and then executing the body.

The macro is more efficient during execution because the macro is processed before compile time. The macro call `ABSDIFF(i, j)` is replaced by the expression

```
(i)>(j) ? (i)-(j) : (j)-(i)
```

which is the body of the macro. When execution reaches this expression, it is ready for evaluation.

When execution reaches the function call `absdiff(...)`, the actual parameters have to be evaluated, the values have to be assigned to the formal parameters, and control has to be transferred to the function definition. Only then can evaluation start of the expression that is ready right away in the case of a macro call. The macro version is therefore faster. To improve efficiency many facilities of the C standard library, are implemented as macros, though their activations look like function calls.

lines 16 and 17 Again the macro is successful in mimicking the function: the macro call gives the right answer. It is worth reflecting on how the same answers are arrived at, as it has happened in a different way.

To execute the outer call to the function `absdiff`, the expressions for the actual parameters have to be evaluated. One of these involves itself a call to this function. This inner call is completed first, and then the outer call can proceed.

The macro processor replaces in the macro body

```
(a)>(b) ? (a)-(b) : (b)-(a)
```

the name `a` by `i` and the name `b` by `ABSDIFF(j, i)` which results in

```
(i)>(ABSDIFF(j, i)) ? (i)-(ABSDIFF(j, i)) : (ABSDIFF(j, i))-(i)
```

This result contains itself macro calls. The macro processor continues replacing macro calls by bodies until no macro calls are left. Accordingly, it makes in this case another pass over the macro body with the result

```
(i)>((j)>(i) ? (j)-(i) : (i)-(j)
     ) ?  (i)-((j)>(i) ?  (j)-(i) : (i)-(j)
               ) : ((j)>(i) ? (j)-(i) : (i)-(j)
                   )-(i)
```

which gives 2 for `i` equal to 3 and `j` equal to 4.

lines 19 and 20 The preceding examples may have suggested that `ABSDIFF` is successful in emulating the function definition. This is not the case. For example in call 6 the increment and the decrement happen once, when evaluating the actual parameters.

On the other hand, the macro call replaces

```
(a)>(b) ? (a)-(b) : (b)-(a)
```

by

```
(++p)>(--q) ? (++p)-(--q) : (--q)-(++p)
```

In the evaluation of this expression the increment and the decrement happen twice: once during evaluation of the condition and once in evaluating the expression whose value is to become the value of the entire conditional.

D.2 Phases of preprocessing

The various activities of the pre-processor are dependent on the order in which they are executed. This order is the following.

1. Trigraph sequences are replaced by their equivalents. For example, the keyboard may have forced code to have been entered as
   `??=include <stdio.h>`
   The preprocessor changes this to
   `#include <stdio.h>`

2. There are various ways in which we can prevent lines of code from becoming too long. If all else fails, we can interrupt a line anywhere by a newline provided it is preceded immediately by a backslash. In this stage, the preprocessor removes every occurrence of a backslash followed by a newline (the compiler doesn't mind long lines). Such backslashes were used, for example, in Program C.2.

3. Comments are removed.

4. `#include` directives are obeyed. The resulting header files typically contain many macros.

5. Macros are expanded.

6. Escape sequences such as occur in `'\t'` or in `"Hello!\n"` are replaced by their equivalents. It may happen that strings are juxtaposed, as in `"Hello, th""ere!"`. Such occurrences are replaced by their concatenation. The result is that in this case it would be as if `"Hello, there!"` had been written.

The result is now ready for the compiler.

Appendix E

Difficult declarations

So far just about the most complex declaration was `char* argv[]`, which we had no difficulty recognizing as a declaration of `argv`, saying that it is an array of pointers to characters. But you may well find yourself puzzled when confronted with something* like

```
int*(*(*(*x)())[2])()
```

You show it to the kind lady in the next cubicle who throws a glance at it and says: "`x` is a pointer to a function returning a pointer to a two-element array of pointers to functions returning a pointer to an integer", which may not help at all.

"Why?"

"Oh—dunno. That's what it looks like and that's what it is."

What you need is the *Beginner's Introduction to Difficult Declarations*, which follows.

In the first place, however complicated a mess of asterisks, parentheses, brackets, primitive types, or `typedef` names you find, there is always a *name* in there that is not a `typedef` name, and that name is what is being declared by the declaration. In this case that name is `x`.

After you've found the name, you work outwards from there. The asterisk in front of `x` says that `x` is a pointer to something. The parentheses to the right of `(*x)` indicate that that something is a function. The asterisk to the left of `(*x)()` indicates that that function returns a pointer. The rest of the declaration specifies what type of thing that pointer points at.

Working outwards, we find as successive constituents of the declaration:

*Example from S.P. Harbison III and Guy L. Steele, Jr: "C, a Reference Manual", page 102.

```
                                x
                               *x
                              (*x)()
                             *(*x)()
                            (*(*x)())[2]
                           *(*(*x)())[2]
                          (*(*(*x)())[2])()
                 int* (*(*(*x)())[2])()
```

We now add the meanings of these constituents, starting at the top, with the simplest. Every time we add something simple to something we already know, so it's a safe and painless process.

```
           x             x is ...
          *x             a pointer to ...
         (*x)()          a function returning ...
        *(*x)()          a pointer to ...
       (*(*x)())[2]      a two-element array of ...
      *(*(*x)())[2]      pointers to ...
     (*(*(*x)())[2])()   a function returning ...
int* (*(*(*x)())[2])()   a pointer to an integer
```

Granted that `int* (*(*(*x)())[2])()` isn't as horrible as it looks at first sight, one should not write such complex declarations. Still, we should be able to handle them because other programmers may not have read this book.

How to avoid excessive complexity in declarations? In the first place cast a skeptical look on the design that seems to call for such a complicated data structure. If it is truly justified, then introduce the complicated declaration in stages. For example, you might use `typedef` to introduce an auxiliary type, say, `fpi` that indicates a pointer to a function returning a pointer to an integer:

```
typedef int*(*fpi)();
```

and an auxiliary type, say, `arp` that indicates a pointer to an two-element array of `fpi`:

```
typedef fpi (*arp)[2];
```

With these auxiliary types in place, the declaration of `x` becomes simply:

```
arp (*x)();
```

Let us now test this declaration to see whether our reasoning has been correct. We get a really simple function that is of type `*fpi`:

```
int* fn0(int* ip) {return ip;}
```

It just returns a copy of its actual parameter. We also define an array `arr` that can be pointed to by something of type `arp`:

```
fpi arr[] = {&fn0, &fn0};
```

Finally, a function `fn1` that can be pointed to by `x`:

```
arp fn1(arp arrPtr) {return arrPtr;}
```

See Program E.1: *Slaying the Monster*, where it all comes together.

The fact that we have tamed the difficult declaration does not imply that using the `x` obtained with the tamed declaration is easy to use.

```
00 #include<stdio.h>
01
02 typedef int*(*fpi)();
03 typedef fpi (*arp)[2];
04 arp (*x)();
05 int* fn0(int* ip) {return ip;} // &fn0 is of type fpi
06 arp fn1(arp arrPtr) {return arrPtr;}
07 // &fn1 and x have the same type
08 fpi arr[] = {&fn0, &fn0};
09 // &arr is of type arp
10 int i = 12345;
11
12 int main() {
13   x = &fn1;
14   printf("%d\n"
15          , *(((*(*x)(&arr))[0])(&i))
16          );
17 }
```

Figure E.1: *Slaying the Monster*, introducing a complicated type in stages.

```
*(((*(*x)(&arr))[0])(&i))
 123 4  45    53   2    1
```

Here is the expression in line 15 in Program E.1 annotated with a single digit under each parenthesis. The digits are chosen to show which parenthesis goes with which. In the table below you see what each subexpression denotes.

```
4..4: function returing pointer to arp
5..5: arp
4..45..5: arp
3..3: array[2] of fpi
2..2: fpi
1..1: pointer to int
```

Appendix F

Glossary

problem reduction

address Number used to identify the location of a byte in memory.

actual parameter Item in the parenthesized list of expressions in a function call. Has to match the corresponding formal parameter in the function's definition.

constant Variable of which the value cannot be changed. It can only be given a value by an initialized declaration. The declaration has to be qualified by the keyword `const`.

control Mechanisms that determine the order in which statements are executed.

declaration Makes an identifier and its type known to the compiler. Usually the declaration of a variable is the only one and also serves as the definition of the variable. The declaration of a function may be its unique definition (can be referred to as the *defining declaration*), but can also consist of only the function prototype and then it is referred to as a *referencing declaration*. Any number of referencing declarations may exist for the same function.

definition Declaration in which space for a variable is allocated. This may include initialization. Definition of a function also associates an identifier with an allocated area of memory. This area is initialized to contain the code resulting from compilation of the function's source code, and cannot be changed.

defining declaration Declaration in which a function is defined, as opposed to *referencing declaration*.

dependence Relation between definitions of functions A and B. A depends of B when the definition of A contains a call to B.

dereference Operation on a pointer that yields the object pointed at. It is denoted by the prefix operator `*`.

initialize To ensure that a variable has the intended value before the first time an expression containing the variable is evaluated. It is a common error to overlook this need. To avoid this error it is advisable to initialize variables in combination with their definition wherever possible.

formal parameter Any of the identifiers in a non-empty *parameter list* in the declaration of a function.

identifier Sequence of letters, digits, or underscores that does not begin with a digit.

literal Atomic expression that is not an identifier. It denotes the same value in all possible C programs. Compare constants and variables, which are identifiers that are associated with a value via a declaration and/or assignments.

name Attribute of a variable. It is an identifier that gives access to other attributes of the variable, such as content and address. Functions are another kind of object named by an identifier. Non-object entities that have names include enumeration constants, labels, components of structures and unions.

numeral Representation of a number. Distinct numerals can represent the same number, for example CLXXIX and, in C notation, `179`, `0263`, and `0xb3` are equivalent numerals.

parameter list List occurring between the name of the function and its body in a function declaration.

pointer Variable that has an address as value.

problem reduction Step in the problem-solving procedure that replaces a problem to be solved by zero or more easier problems. The aim is solve difficult problems in a number of easily executed problem-reduction steps. In the context of programming problem reduction translates to defining a function by zero or more function calls. Related to *recursion*, *stepwise refinement*, *top-down programming*, and *wishful thinking*.

pseudo code Text in the format of a program where the elements are succinct informal equivalents in English of their counterparts expressed in a programming language.

recursion Use or definition of a function f that depends on itself or depends on a function that depends on f. Can be regarded as an application of *problem reduction* when the function depended on represents an easier problem to be solved.

reference Used as a verb this refers to the relation between a name and the object named by it. In the context of C the use of the word as a noun is not helpful. For example, a pointer is a pointer, not a reference. But see *dereference*. Also, the use of a name before the introduction of that name by a declaration is called "forward reference". It is forbidden in C, but for functions its effect can be obtained by writing a function prototype as a *referencing declaration*, in advance of the *defining declaration*.

referencing declaration Declaration of a function that is not the *defining declaration*. It takes the form of function heading followed by semicolon.

scope Part of the program in which a declaration is in force. It may happen that inside the scope of a declaration of a variable a declaration occurs of a variable of the same name. In the inner scope that name refers to the variable created by the declaration in the inner scope.

stepwise refinement A form of *problem reduction* in which problem reduction steps are stated in *pseudo code*.

straight-line code Code in which every statement is executed exactly once, and executed in the order written in the source code.

visibility Inside the scope of a variable's declaration it may happen that the identifier of the declaration refers to another variable; see *scope*. It is said that the outer declaration is then not visible.

wishful thinking Can be of interest to problem solving when it takes the form: "This problem would be solvable if only I had solutions to this, this, and that other problem." In a programming context, it translates to problem-solving procedures known as *top-down programming* and *stepwise refinement*.

Index