Chapter 3

Functions

3.1 A first C++ function

Crackrib spent five years in Dartmoor for house-breaking. He bore an exemplary character, and devoted his spare time to study; his special interest being mathematics. He spent many months in copying out tables from a book of logarithms, and he was permitted by the authorities to take these copies away. Subsequently, Crackrib published a sensational "Diary" of Dartmoor life. The following is a specimen of the "first-hand material" on which his "Diary" was based:[1]

Logs	Nos.	5050		5051		5052		5053
0	703	2920	703	3714	703	4665	703	5523
1		3015		3819		4718		5595
2		3084		3920		4805		5664
3		3151		4015		4920		5720
4		3225		4112		4988		5828
5		3310		4204		5065		5935
6		3421		4313		5125		6019
7		3497		4435		5219		6123
8		3567		4523		5320		6201
9		3639		4608		5415		6277

Can you decipher the entry? (This cipher is based upon a episode which is stated to have actually occurred.)

Hubert Phillips, "Caliban" Problem No. 35,
The Week-End Problems Book (1933)

This section introduces C++ functions as a means to encapsulate and reuse functionality, and to subdivide a program into subtasks. You will learn how to add functions to your programs, and how to call them. We also explain how functions can efficiently be made available for many programs at the same time, through separate compilation and libraries.

In many numerical calculations, computing powers is a fundamental operation (see Section 2.5), and there are many other operations that occur frequently in applications. In C++, *functions* are used to encapsulate such frequently used operations, making it easy to invoke them many times, with different arguments, and from different programs, but *without* having to reprogram them every time.

[1] the entry in row "7" and column "5052", for example, has to be read as "the fractional part of $\log_{10}(50527)$ is .7035219 (up to a precision of seven digits)"

Even more importantly, functions are used to structure a program. In practice, large programs consist of many small functions, each of which serves a clearly defined subtask. This makes it a lot easier to read, understand, and maintain the program.

We have already seen quite a number of functions, since the main function of every C++ program is a special function (Section 2.1.4).

Program 17 emphasizes the encapsulation aspect and shows how functions can be used. It first defines a function for computing the value b^e for a given real number b and given integer e (possibly negative). It then calls this function for several values of b and e. The computations are performed over the floating point number type double.

```
1  // Prog: callpow.C
2  // Define and call a function for computing powers.
3
4  #include <iostream>
5
6  // PRE:  e >= 0 || b != 0.0
7  // POST: return value is b^e
8  double pow (double b, int e)
9  {
10   double result = 1.0;
11   if (e < 0) {
12     // b^e = (1/b)^(-e)
13     b = 1.0/b;
14     e = -e;
15   }
16   for (int i = 0; i < e; ++i) result *= b;
17   return result;
18 }
19
20 int main()
21 {
22   std::cout << pow( 2.0, -2) << "\n"; // outputs 0.25
23   std::cout << pow( 1.5,  2) << "\n"; // outputs 2.25
24   std::cout << pow( 5.0,  1) << "\n"; // outputs 5
25   std::cout << pow( 3.0,  4) << "\n"; // outputs 81
26   std::cout << pow(-2.0,  9) << "\n"; // outputs -512
27
28   return 0;
29 }
```

Program 17: *progs/callpow.C*

Before we explain the concepts necessary to understand this program in detail, let us get an overview of what is going on in the function pow. For nonnegative exponents e, b^e is obtained from the initial value of 1 by e-fold multiplication with b. This is what the

for-loop does. The case of negative e can be handled by the formula $b^e = (1/b)^{-e}$: after inverting b and negating e in the if-statement, we have an equivalent problem with a positive exponent. The latter only works if $b \neq 0$, and indeed, negative powers of 0 are mathematically undefined.

3.1.1 Pre- and postconditions

Even a very simple function should document its *precondition* and its *postcondition*, in the form of comments. The precondition specifies what has to hold when the function is called, and the postcondition describes value and effect of the function. This information allows us to understand the function without looking at the actual sourcecode; this in turn is a necessary for keeping track of larger programs. In case of the function pow, the precondition

```
// PRE:  e >= 0 || b != 0.0
```

tells us that e must be nonnegative, or (if e is negative) that $b \neq 0$ must hold. The postcondition

```
// POST: return value is b^e
```

tells us the function value, depending on the arguments. In this case, there is no effect.

The pre- and postconditions specify the function in a mathematical sense. At first sight, functions with values *and* effect [2] do not fit into the framework of mathematical functions which only have values. But using the concept of *program states* (Section 1.2.3), a C++ function can be considered as a mathematical function that maps program states (immediately *before* the function call) to program states (immediately *after* the function call).

Under this point of view, the precondition specifies the *domain* of the function, the set of program states in which the function may be called. In case of pow, these are all program states in which the arguments b and e are in a suitable relation. The postcondition describes the function itself by specifying how the (relevant part of the) program state gets transformed. In case of pow, the return value b^e will (temporarily) be put at some memory location.

To summarize, the postcondition tells us what happens when the precondition is satisfied. On the other hand, the postcondition gives *no guarantee whatsoever* for the case where the precondition is not satisfied. From a mathematical point of view, this is fine: a function is simply not defined for arguments outside its domain.

Arithmetic pre- and postconditions. The careful reader of Section 2.5 might have realized that both pre- and postcondition of the function pow cannot be correct. If e is too large, for example, the computation might overflow, but such e are not excluded by

[2] Formally, it is the function *call* that has the value and effect, but we suppress this subtlety. Even mathematicians talk about a function value when they mean the value resulting from an evaluation of the function with certain arguments.

the precondition. Even if there is no overflow, the value range of the type `double` may have a hole at b^e, meaning that this value cannot be returned by the function. The postcondition is therefore imprecise as well.

In the context of arithmetic operations over the fundamental C++ types, it is often tedious and even undesirable to write down precise pre- and postconditions; part of the problem is that fundamental types may behave differently on different platforms. Therefore, we often confine ourselves to pre- and postconditions that document our *mathematical* intention, but we have to keep in mind that in reality, the function might behave differently.

Assertions. So far, our preconditions are just comments like in

```
// PRE:  e >= 0 || b != 0.0
```

Therefore, if the function `pow` is called with arguments `b` and `e` that violate the precondition, this passes unnoticed. On the syntactical level, there is nothing we can do about it: the function call `pow (0.0, -1)`, for example, will compile. But we can make sure that this blunder is detected at runtime. A simple way to do this uses *assertions*. An assertion has the form

assert (*expr*)

where *expr* is a predicate, an expression of a type whose values can be converted to `bool`. No comma is allowed in *expr*, a consequence of the fact that `assert` is not a function but a *macro*. A macro is a piece of meta-code that the compiler replaces with actual C++ code prior to compilation.

With assertions, `pow` can be written as follows.

```
// PRE:  e >= 0 || b != 0.0
// POST: return value is b^e
double pow (double b, int e)
{
  assert (e >= 0 || b != 0.0);
  double result = 1.0;
  // the remainder is as before
  ...
}
```

The purpose of an assertion is to check whether a certain predicate holds at a certain point. The precise semantics of an assertion is as follows. *expr* is evaluated, and if it returns *false*, execution of the program terminates immediately with an error message telling us that the respective assertion was violated. If *expr* returns *true*, execution continues normally. In our case, this means that the evaluation of the expression `pow (0.0,-1)` leads to a *runtime error*. This might not be a very polite way of telling the user that the arguments were illegal but the point will surely come across.

You can argue that it is costly to test the assertion in every function call, just to catch a few "bad" calls. However, it is possible to tell the compiler to ignore the `assert` macro, meaning that an empty piece of C++ code replaces it. The usual way to go is therefore as follows: during code development, put assertions everywhere you want to be sure that something really holds. When the code is stable (and no assertion violations seem to occur anymore), tell the compiler to remove the assertions. The machine language code is then as efficient as if you would never have written the assertions in the first place.

To use the `assert` macro, we have to include the header `cassert`.

3.1.2 Function definitions

Lines 8–18 of Program 17 define a function called `pow`. The syntax of a function definition is as follows.

T fname (*T1 pname1*, *T2 pname2*, ..., *Tk pnamek*)
    block

This defines a function called *fname*, with *return type T*, and with *formal arguments pname1*,..., *pnamek* of types *T1*,..., *Tk*, respectively, and with a *function body block*.

Syntactically, *T* and *T1*,..., *Tk* are type names, *fname* as well as *pname1*,..., *pnamek* are identifiers (Section 2.1.9), and *block* is a block, a sequence of statements enclosed by curly braces (Section 2.4.3).

We can think of the formal arguments as placeholders for the actual arguments that are supplied (or "passed") during a function call.

Function definitions must not appear inside blocks, other functions, or control statements. They may appear inside namespaces, though, or at global scope, like in `callpow.C`. A program may contain an arbitrary number of function definitions, appearing one after another without any delimiters between them. In fact, the program `callpow.C` consists of *two* function definitions, since the `main` function is a function as well.

3.1.3 Function calls

In Program 17, `pow(2.0,-2)` is one of five function calls. Formally, a function call is an expression. The syntax of a function call that matches the general function definition from above is as follows.

fname (*expr1*, ..., *exprk*)

Here, *expr1*,...,*exprk* must be expressions of types whose values can be converted to the formal argument types *T1*,...,*Tk*. These expressions are the *call arguments*. For all types that we know so far, the call arguments as well as the function call itself are rvalues. The type of the function call is the function's return type *T*.

When a function call is evaluated, the call arguments are evaluated first (in an order that is unspecified by the C++ standard). The resulting values are then used to initialize the formal arguments. Finally, the function body is executed; in this execution, the formal arguments behave like they were variables defined in the beginning of *block*, initialized with the values of the call arguments.

The evaluation of a function call terminates as soon as a return statement is reached, see Section 2.1.14. This return statement must be of the form

```
return expr;
```

where *expr* is an expression of a type whose values can be converted to the return type *T*. The resulting value is the value of the function call. The effect of the function call is determined by the joint effects of the call argument evaluations, and of executing *block*.

The function body may contain several return statements, but if no return statement is reached during the execution of *block*, value and effect of the function call are undefined (unless the return type is `void`, see Section 3.1.4 below).

For example, during the execution of *block* in `pow(2.0,-2)`, `b` and `e` initially have values 2 and -2. These values are changed in the `if`-statement to 0.5 and 2, before the subsequent loop sets `result` to 0.5 in its first and to 0.25 in its second and last iteration. This value is returned and becomes the value of the function call expression `pow(2.0,-2)`.

3.1.4 The type void

In C++, there is a fundamental type called `void`, used as return type for functions that only have an effect, but no value. Such functions are also called *void functions*.

As an example, consider the following program (note that the function `print_pair` requires no precondition, since it works for any combination of `int` values).

```
1  #include <iostream>
2
3  // POST: "(i, j)" has been written to standard output
4  void print_pair (int i, int j)
5  {
6    std::cout << "(" << i << ", " << j << ")\n";
7  }
8
9  int main()
10 {
11   print_pair(3,4); // outputs (3, 4)
12 }
```

The type `void` has empty value range, and there are no literals, variables, or formal function arguments of type `void`. There are expressions of type `void`, though, for example `print_pair(3,4)`.

A void function does not require a return statement, but it may contain return statements with *expr* of type `void`, or return statements of the form

```
return;
```

Evaluation of a void function call terminates when a return statement is reached, *or* when the execution of *block* is finished.

3.1.5 Functions and scope

The parenthesized part of a function definition contains the declarations of the formal arguments. For all of them, the declarative region is the function definition, so the formal arguments have local scope (Section 2.4.3). The potential scope of a formal argument declaration begins after the declaration and extends until the end of the function body. Therefore, the formal arguments are not visible outside the function definition. Within the body, the formal arguments behave like variables that are local to *block*.

In particular, changes made to the values of formal arguments (like in the function `pow`) are "lost" after the function call and have no effect on the values of the call arguments. This is not surprising, since the call arguments are rvalues, but to make the point clear, let us consider the following alternative `main` function in `callpow.C`.

```
1 int main() {
2   double b = 2.0;
3   int e = -2;
4   std::cout << pow(b,e); // outputs 0.25
5   std::cout << b;        // outputs 2
6   std::cout << e;        // outputs -2
7
8   return 0;
9 }
```

The values of the variables `b` and `e` defined in lines 2–3 stay the same throughout, since the function body of `pow` is not in the scope of their declarations, for two reasons. First, the definition of `pow` appears *before* the declarations of `b` and `e` in lines 2–3, so the body of `pow` cannot even be in the *potential* scope of these declarations. Second, even if we would move the declarations of the variables `b` and `e` to the beginning of the program (before the definition of `pow`, so that they have global scope), their scope would exclude the body of `pow`, since that body is in the potential scopes of redeclarations of the names `b` and `e` (the formal arguments), see Section 2.4.3.

But the general scope rules of Section 2.4.3 *do* allow function bodies to use names of global or namespace scope; the program on page 169 for example uses `std::cout` as such a name. Here is a contrived program that demonstrates how a program may modify a *global variable* (a variable whose declaration has global scope). While such constructions may be useful in certain cases, they usually make the program less readable, since the effect of a function call may then become very non-local.

```
1  #include<iostream>
2
3  int i = 0; // global variable
4
5  void f()
6  {
7    ++i;     // in the scope of declaration in line 3
8  }
9
10 int main()
11 {
12   f();
13   std::cout << i << "\n"; // outputs 1
14
15   return 0;
16 }
```

Since the formal arguments of a function have local scope, they also have automatic storage duration. This means that we get a "fresh" set of formal arguments every time the function is called, with memory assigned to them only until the respective function call terminates.

Names declared inside the function body must be distinct from the names of all formal arguments, unless they appear in a nested block. This makes sense since otherwise, it would be possible to irrevocably hide the name of a formal argument. Therefore, we cannot write

```
int f (int i)
{
  int i = 5; // invalid; i hides formal argument
  return i;
}
```

while the following is not recommended but legal.

```
int f (int i)
{
  {
    int i = 5; // ok; i is local to nested block
  }
  return i; // the formal argument
}
```

The latter function is the identity, since the scope of the declaration `int i = 5` is limited to the nested block.

Function declarations. A function itself also has a scope, and the function can only be called within its scope. The scope of a function is obtained by combining the scopes of

all its *declarations*. The part of the function definition before *block* is a declaration, but there may be function declarations that have no subsequent *block*. This is in contrast to variables where every declaration is at the same time a definition. A function may be declared several times, but it can be defined once only.

The following program, for example, does not compile, since the call of f in main is not in the scope of f.

```
#include<iostream>

int main()
{
  std::cout << f(1); // f undeclared
  return 0;
}

int f (int i) // scope of f begins here
{
  return i;
}
```

But we can put f into the scope of main by adding a declaration before main, and this yields a valid program.

```
#include<iostream>

int f (int i); // scope of f begins here

int main()
{
  std::cout << f(1); // ok, call is in scope of f
  return 0;
}

int f (int i)
{
  return i;
}
```

In the previous program, we could get rid of the extra declaration by simply defining f before main, but sometimes, separate function declarations are indeed necessary. Consider two functions f and g such that g is called in the function body of f, and f is called in the function body of g. We *have* to define one of the two functions first (f, say), but since we call g within the body of f, g must have a declaration *before* the definition of f.

3.1.6 Procedural programming

So far, we have been able to "live" without functions only since the programs that we have written are pretty simple. But even some of these simple ones would benefit from functions. Consider as an example the program `perfect.C` from Exercise 39. In this exercise, we have asked you to find the perfect numbers between 1 and n, for a given input number n. The solution so far uses one "big" *double loop* (loop within a loop) that in turn contains two `if` statements. Although in this case, the "big" loop is still small enough to be read without difficulties, it doesn't really reflect the logical structure of the solution. Once we get to triple or quadruple loops, the program becomes very hard to follow.

But what *is* the logical structure of the solution? For every number i between 1 and n, we have to **test whether** i **is perfect**; and to do the latter, we have to **compute the sum of all proper divisors of** i and check whether it is equal to i. Thus, we have two clearly defined subtasks that the program has to solve for every number i, and it is best to encapsulate these into functions. Program 18 shows how this is done. Note that the program is now almost self-explanatory: the postconditions can more or less directly be read off the function names.

```
1  // Program: perfect2.C
2  // Find all perfect numbers up to an input number n
3
4  #include <iostream>
5
6  // POST: return value is the sum of all divisors of i
7  //       that are smaller than i
8  unsigned int sum_of_proper_divisors (unsigned int i)
9  {
10   unsigned int sum = 0;
11   for (unsigned int d = 1; d < i; ++d)
12     if (i % d == 0) sum += d;
13   return sum;
14 }
15
16 // POST: return value is true if and only if i is a
17 //       perfect number
18 bool is_perfect (unsigned int i)
19 {
20   return sum_of_proper_divisors (i) == i;
21 }
22
23 int main()
24 {
25   // input
```

```
26    std::cout << "Find perfect numbers up to n =? ";
27    unsigned int n;
28    std::cin >> n;
29
30    // computation and output
31    std::cout << "The following numbers are perfect.\n";
32    for (unsigned int i = 1; i <= n ; ++i)
33      if (is_perfect (i)) std::cout << i << " ";
34    std::cout << "\n";
35
36    return 0;
37  }
```

Program 18: *progs/perfect2.C*

Admittedly, the program is longer than `perfect.C`, but it is more readable, and it has simpler control flow. In particular, the double loop has disappeared.

The larger a program gets, the more important is it to subdivide it into small subtasks, in order not to lose track of what is going on in the program on the whole; this is the *procedural programming* paradigm, and in C++, it is realized with functions.

The procedural programming paradigm is not so self-evident as it may seem today. The first programming language that became accessible to a general audience since the 1960's was BASIC (**B**eginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode).

In BASIC, there were no functions; in order to execute a code fragment responsible for a subtask, you had to use the `GOTO` statement (with a line number)—or `GOSUB` in many dialects—to jump to that code, and then jump back using another `GOTO` (`RETURN`, respectively). The result was often referred to as *spaghetti code*, due to the control flow meandering like a boiled spaghetti on a plate. Moreover, programmers often didn't think in terms of clearly defined subtasks, simply because the language did not support it. This usually lowered the code quality even further.

Despite this, BASIC was an extremely successful programming language. It reached the peak of its popularity in the late 1970's and early 1980's when the proud owners of the first home computers (among them the authors) created programs of fairly high complexity in BASIC. [3]

3.1.7 Arrays as function arguments

We have seen in Section 2.6.2 that an array cannot be initialized from another array, and this implies that arrays have to receive special attention in the context of functions. The usual first step in a function call evaluation (the call arguments are evaluated, and their values are used to initialize the formal arguments) can't be done with arrays.

[3] If you want to understand how we accomplished this, you should know that we were also passionate enough to type up several pages of program code published in computer magazines, and that we used to store programs as sequences of beeps on audio cassettes.

Given this, it might be surprising that formal arguments of array type are allowed. For example, we could declare a function

```
// PRE:  a[0],...,a[n-1] are elements of an array
// POST: a[i] is set to value, for 0 <= i < n
void fill_n (int a[], int n, int value);
```

to set all elements of an array to some fixed value. The compiler, however, internally *adjusts* this to the completely equivalent declaration

```
// PRE:  a[0],...,a[n-1] are elements of an array
// POST: a[i] is set to value, for 0 <= i < n
void fill_n (int* a, int n, int value);
```

The same adjustment would happen for the formal argument `int a[5]`, say, meaning that the array length is ignored. You could in fact (legally, but quite confusingly) have a formal argument `int a[5]`, and then use an array of length 10 as call argument.

The moral is that in reality, no formal arguments of array type exist, and in order to avoid confusion, it is better not to pretend otherwise.

If we want to build a function that works with arrays, we therefore have to think about alternative ways of passing the array to the function. An obvious way is suggested by the declaration of `fill_n` above: we pass a pointer to the first element, along with the number of elements. A different possibility is to pass *two* pointers, one to the first element, and a past-the-end pointer. This also uniquely describes the array. In both variants, we may actually choose the call arguments in such a way that they describe a contiguous *subarray* of the original array. This generalization is possible since the array itself never appears as an argument.

At this point, it seems like a matter of taste which of the two variants is preferable; but if you think about how the function `fill_n` is naturally implemented in both variants, we see a difference. Here is a program that defines and uses the two variants (the second one is just called `fill` since there is no `n`) that naturally result, given the respective formal arguments.

```
1  // Program: fill.C
2  // define and use two functions to fill an array
3
4  #include<iostream>
5
6  // PRE:  a[0],...,a[n-1] are elements of an array
7  // POST: a[i] is set to value, for 0 <= i < n
8  void fill_n (int* a, int n, int value) {
9    // iteration by index
10   for (int i = 0; i < n; ++i)
11     a[i] = value;
12 }
13
```

```
14 // PRE:  [first, last) is a valid range
15 // POST: *p is set to value, for p in [first, last)
16 void fill (int* first, int* last, int value) {
17   // iteration by pointer
18   for (int* p = first; p != last; ++p)
19     *p = value;
20 }
21
22 int main()
23 {
24   int a[5];
25   fill_n (a, 5, 0); // a == {0, 0, 0, 0, 0}
26   fill (a, a+5, 1); // a == {1, 1, 1, 1, 1}
27   return 0;
28 }
```

Program 19: *progs/fill.C*

In the first variant, we iterate over all indices in the set $\{0, \ldots, n-1\}$, and we get the array elements by random access. In the second variant, we iterate over all addresses in the *range* [first, last), and we get the array elements by dereferencing. A valid range contains the addresses of a (possibly empty) set of consecutive array elements, where the halfopen interval notation [first, last) means that the range is given by the values of first, first+1, ...,last-1. In other words, last is a past-the-end pointer for the subarray described by the range.

As we have already argued in Section 2.6.5, the second variant implements the "natural" iteration over an array, and therefore seems preferable. But the *real* reason why it is indeed preferable lies somewhere else. In C++, there are techniques to make functions like fill or fill_n available not only for arrays, but for many other containers at the same time. In this general setting, the functions work with iterators.We may think of them as generalized pointers to container elements, but the operations that we can actually perform on these "pointers" depend on the container.

There are many natural containers that do not offer random access to their elements. For such containers, fill_n as above won't work, since the subscript operator is not available for their "pointers". The underlying operation of adding integers to such "pointers" is then not defined, either.

On the other hand, the way we have defined a container in Section 2.6.5, we *are* guaranteed that we can iterate over its elements. By convention, this is realized through "pointer" increment, using the operator ++. In fact, the operation ++p is available, even if p+1 is not; the latter is random access functionality for the special right-hand side operand 1.

Therefore, the function fill as above has the potential to work for all containers, since it only requires "pointer" functionality that is offered by all container iterators.

Mutating functions. There is a substantial difference between the function pow on the one hand, and the functions fill and fill_n on the other hand. A call to the function pow has no effect, since the computations only modify formal argument values; these values are "local" to the function call and "disappear" upon termination. With pow, it's the *value* of a function call that we are interested in.

Calls to the functions fill and fill_n, on the other hand, have effects: they modify the values of array elements, and these values are *not* local to the function call. When we write

```
int a[5];
fill (a, a+5, 0);
```

the effect of the expression fill (a, a+5, 0) is that all elements of a receive value 0. This is possible since there are formal arguments of pointer type. When the function call fill (a, a+5, 0) is evaluated, the formal argument first is initialized with the *address* of a's first element. In the function body, the value at this address is modified through the lvalue *p, and the same happens for the other four array elements in turn.

Formal arguments of pointer type are therefore a means of constructing functions with value-modifying effects. Such functions are called *mutating*.

3.1.8 Modularization

There are functions that are tailor-made for a specific program, and it would not make sense to use them in another program. But there are also general purpose functions that are useful in many programs. It is clearly undesirable to copy the corresponding function definition into any program that calls the function; what we need is *modularization*, a subdivision of the program into independent parts.

The power function pow from Program 17 is certainly general purpose. In order to make it available to all our programs, we can simply put the function definition into a separate sourcecode file pow.C, say, in our working directory.

```
1  #include <cassert>
2
3  // PRE:  e >= 0 || b != 0.0
4  // POST: return value is b^e
5  double pow (double b, int e)
6  {
7    assert (e >= 0 || b != 0.0);
8    double result = 1.0;
9    if (e < 0) {
10     // b^e = (1/b)^(-e)
11     b = 1.0/b;
12     e = -e;
13   }
14   for (int i=0; i<e; ++i) result *= b;
```

```
15    return result;
16  }
```

Program 20: *progs/pow.C*

Then we can include this file from our main program as follows.

```
1  // Prog: callpow2.C
2  // Call a function for computing powers.
3
4  #include <iostream>
5  #include "pow.C"
6
7  int main()
8  {
9    std::cout << pow( 2.0, -2) << "\n"; // outputs 0.25
10   std::cout << pow( 1.5,  2) << "\n"; // outputs 2.25
11   std::cout << pow( 5.0,  1) << "\n"; // outputs 5
12   std::cout << pow( 3.0,  4) << "\n"; // outputs 81
13   std::cout << pow(-2.0,  9) << "\n"; // outputs -512
14
15   return 0;
16 }
```

Program 21: *progs/callpow2.C*

An include directive of the form

```
#include "filename"
```

logically replaces the include directive by the contents of the specified file. Usually, *filename* is interpreted relative to the working directory.

Separate compilation and object code files. The code separation mechanism from the previous paragraph has one major drawback: the compiler does not "see" it. Before compilation, pow.C is logically copied back into the main file, so the compiler still has to translate the function definition into machine language *every time* it compiles a program that calls pow. This is a waste of time that can be avoided by *separate compilation*.

In our case, we would compile the file pow.C separately. We only have to tell the compiler that it should not generate an executable program (it can't, since there is no main function) but an *object code* file, called pow.o, say. This file contains the machine language instructions that correspond to the C++ statements in the function body of pow.

Header files. The separate compilation concept is more powerful than we have seen so far: surprisingly, even programs that call the function pow can be compiled separately,

without knowing about the source code file `pow.C` or the object code file `pow.o`. What the compiler needs to have, though, is a declaration of the function `pow`.

This function declaration is best put into a separate file as well. In our case, this file `pow.h`, say, is very short; it contains just the lines

```
// PRE:  e >= 0 || b != 0.0
// POST: return value is b^e
double pow (double b, int e);
```

Since this is the "header" of the function `pow`, the file `pow.h` is called a *header file*. In the calling Program 21, we simply replace the inclusion of `pow.C` by the inclusion of `pow.h`, resulting in the following program.

```
1  // Prog: callpow3.C
2  // Call a function for computing powers.
3
4  #include <iostream>
5  #include "pow.h"
6
7  int main()
8  {
9    std::cout << pow( 2.0, -2) << "\n"; // outputs 0.25
10   std::cout << pow( 1.5,  2) << "\n"; // outputs 2.25
11   std::cout << pow( 5.0,  1) << "\n"; // outputs 5
12   std::cout << pow( 3.0,  4) << "\n"; // outputs 81
13   std::cout << pow(-2.0,  9) << "\n"; // outputs -512
14
15   return 0;
16 }
```

Program 22: *progs/callpow3.C*

From this program, the compiler can then generate an object code file `callpow3.o`. Instead of the machine language instructions for executing the body of `pow`, this object code contains a *placeholder* for the location under which these instructions are to be found in the executable program. It is important to understand that `callpow3.o` cannot be an executable program yet: it *does* contain machine language code for `main`, but *not* for another function that it needs, namely `pow`.

The linker. Only when an executable program is built from `callpow3.o`, the object code file `pow.o` comes into play. Given all object files that are involved, the *linker* builds the executable program by gluing together machine language code for function calls (in `callpow3.o`) with machine language code for the corresponding function bodies (in `pow.o`). Technically, this is done by putting all object files together into a single executable file, and by filling placeholders for function body locations with the actual locations in the executable.

Separate compilation is very useful. It allows to change the definition of a function without having to recompile a single program that calls it. As long as the function declaration remains unchanged, it is only the linker that has to work in the end; and the linker is usually very fast. It follows that separate compilation also makes sense for functions that are specific to one program only.

Separate compilation reflects the "customer" view of the calling program: as long as a function does what its pre- and postcondition promise in the header file, it is not important to know *how* it does this. On the other hand, if the function definition is hidden from the calling program, clean pre- and postconditions are of critical importance, since they may be the only information available about the function's behavior.

Availability of sourcecode. If you have carefully gone through what we have done so far, you realize that we could in principle delete the sourcecode file `pow.C` after having generated `pow.o`, since later, the function definition is not needed anymore. When you buy commercial software, you are often faced with the absence of sourcecode files, since the vendor does not want customers to modify the sourcecode instead of buying updates, or to discover how much money they have paid for lousy software design. [4]

In academic software, availability of sourcecode goes without saying. In order to evaluate or reproduce the contribution of such software to the respective area of research, it is necessary to have sourcecode. Even in commercial contexts, *open source* software is advancing. The most prominent software that comes with all sourcecode files is the operating system *Linux*. Open source software can very efficiently be adapted and improved if many people contribute. But such a contribution is possible only when the sourcecode is available.

Libraries. The function `pow` will not be the only mathematical function that we want to use in our programs. To make the addition of new functions easy, we can put the definition of `pow` (and similar functions that we may add later) into a single sourcecode file `math.C`, say, and the corresponding declarations into a single header file `math.h`. The object code file `math.o` then contains machine language code for all our mathematical functions.

Although not strictly necessary, it is good practice to include `math.h` in the beginning of `math.C`. This ensures consistency between function declarations and function definitions and puts the code in `math.C` into the scope of all functions declared in `math.h`, see Section 3.1.5. In all function bodies in `math.C`, we can therefore call the other functions, without having to think about whether these functions have already been declared.

In general, several object code files may be needed to generate an executable program, and it would be cumbersome to tell the linker about all of them. Instead, object code files that logically belong together can be *archived* into a *library*. Only the name of this library must then be given to the linker in order to have all library functions available for

[4] To be fair, we want to remark that there are also more honest reasons for not giving away sourcecode.

the executable program. In our case, we so far have only one object file `math.o` resulting from `math.C`, but we can still build a library file `libmath.a`, say, from it.

Figure 17 schematically shows how object code files, a library and finally an executable program are obtained from a number of sourcecode files.

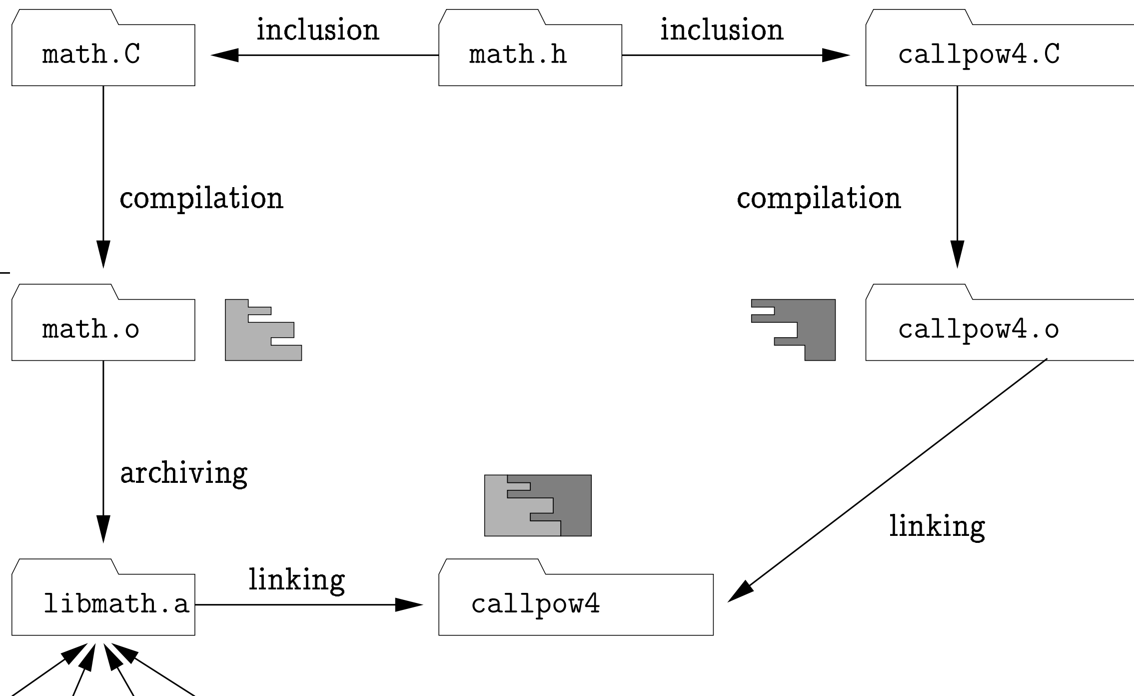

Figure 17: *Building object code files, libraries and executable programs.*

Centralization and namespaces It is clear that we do not want to keep header files and libraries of general interest in our working directory, since we (and others) may have many working directories. Header files and libraries should be at some central place.

We can make our programs independent from the location of header files by writing

```
#include <filename>
```

but in this case, we have to tell the compiler (when we start it) where to search for files to be included. This is exactly the way that headers like `iostream` from the standard library are included; their locations are known to the compiler, so we don't have to provide any information here. Similarly, we can tell the linker where the libraries we need are to be found. Again, for the various libraries of the standard library, the compiler knows this.

We want to remark that *filename* is not necessarily the name of a physical file; the mapping of *filename* to actual files is implementation defined.

Finally, it is good practice to put all functions of a library into a namespace, in order to avoid clashes with user-declared names, see Section 2.1.3. Let us use the namespace `ifm` here.

Here are the header and implementation files `math.h` and `math.C` that result from these guidelines for our intended library of mathematical functions (that currently contains `pow` only).

```
1  // math.h
2  // A small library of mathematical functions.
3
4  namespace ifm {
5    // PRE:  e >= 0 || b != 0.0
6    // POST: return value is b^e
7    double pow (double b, int e);
8  }
```

Program 23: *progs/math.h*

```
1  // math.C
2  // A small library of mathematical functions.
3
4  #include <cassert>
5  #include <IFM/math.h>
6
7  namespace ifm {
8
9    double pow (double b, int e)
10   {
11     assert (e >= 0 || b != 0.0);
12     // PRE:  e >= 0 || b != 0.0
13     // POST: return value is b^e
14     double result = 1.0;
15     if (e < 0) {
16       // b^e = (1/b)^(-e)
17       b = 1.0/b;
18       e = -e;
19     }
20     for (int i=0; i<e; ++i) result *= b;
21     return result;
22   }
23
24 }
```

Program 24: *progs/math.C*

Finally, the program `callpow4.C` calls our library function `ifm::pow`. It includes the header file `math.h` from a central directory `IFM`.

```
1  // Prog: callpow4.C
2  // Call library function for computing powers.
3
4  #include <iostream>
5  #include <IFM/math.h>
6
7  int main()
8  {
9    std::cout << ifm::pow( 2.0, -2) << "\n"; // outputs 0.25
10   std::cout << ifm::pow( 1.5,  2) << "\n"; // outputs 2.25
11   std::cout << ifm::pow( 5.0,  1) << "\n"; // outputs 5
12   std::cout << ifm::pow( 3.0,  4) << "\n"; // outputs 81
13   std::cout << ifm::pow(-2.0,  9) << "\n"; // outputs -512
14
15   return 0;
16 }
```

Program 25: *progs/callpow4.C*

3.1.9 Using library functions

You can imagine that we were not the first to put a function like `pow` into a library. Indeed, the standard library contains a function `std::pow` that is even more general than ours: it can compute b^e for *real* exponents e. Accordingly, the arguments of `std::pow` and its return value are of type `double`. In order to use this function, we have to include the header `cmath`. This header contains declarations for a variety of other numerical functions.

Using functions from the standard library can help us to get shorter, better, or more efficient code, without having to write a single new line by ourselves. For example, computing *square roots* can speed up our primality test in Program 7. You might have realized this much earlier, but when we are looking for some proper divisor of a natural number $n \geq 2$, it is sufficient to search in the range $\{2, \ldots, \lfloor\sqrt{n}\rfloor\}$. Indeed, if n can be written as a product $n = dd'$, then the smaller of d and d' must be bounded by $\sqrt{n}$; since the divisors are integral, we even get a bound of $\lfloor\sqrt{n}\rfloor$, $\sqrt{n}$ rounded down.

The primality test could therefore be written more efficiently as in Program 26, using the function `std::sqrt` from the library `cmath`, whose argument and return types are `double`.

```
1  // Program: prime2.C
2  // Test if a given natural number is prime.
3
```

```
 4 #include <iostream>
 5 #include <cmath>
 6
 7 int main ()
 8 {
 9   // Input
10   unsigned int n;
11   std::cout << "Test if n>1 is prime for n =? ";
12   std::cin >> n;
13
14   // Computation: test possible divisors d up to sqrt(n)
15   unsigned int bound = (unsigned int)(std::sqrt(n));
16   unsigned int d;
17   for (d = 2; d <= bound && n % d != 0; ++d);
18
19   // Output
20   if (d <= bound)
21     // d is a divisor of n in {2,...,[sqrt(n)]}
22     std::cout << n << " = " << d << " * " << n / d << ".\n";
23   else
24     // no proper divisor found
25     std::cout << n << " is prime.\n";
26
27   return 0;
28 }
```

Program 26: *progs/prime2.C*

The program is correct: if `d <= bound` still holds after the loop, we have left the loop because the *other* condition `n % d != 0` has failed. This means that we have found a divisor. If `d > bound` holds after the loop, we have tried all possible divisors smaller or equal to `bound` (whose value is $\lfloor\sqrt{n}\rfloor$, since the explicit conversion rounds down, see Section 2.5.3), so we certainly have not missed any divisor. But we have to be a little careful here: our arguments assume that `std::sqrt` works correctly for squares. For example, `std::sqrt(121)` must return 11 (a little more wouldn't hurt), but *not* 10.99998, say. In that latter case, `(unsigned int)(std::sqrt(121))` would have value 10, and by making this our bound, we miss the divisor 11 of 121, erroneously concluding that `121` is prime.

It is generally not safe to rely on some precise semantics of library functions, even if your platform implements floating point arithmetic according to the IEEE standard 754 (see Section 2.5.6). The square root function is special in the sense that the IEEE standard still guarantees the result of `std::sqrt` to be the floating point number closest to the real square root; consequently, our above implementation of the primality test is safe. But similar guarantees do *not* necessarily hold for other library functions.

Also in our second prime number application, *Eratosthenes's Sieve*, we'd better call

a standard library function in order to initialize our list of crossed out numbers, instead of doing it ourselves with a loop. For this, we would replace the two lines

```
  for (unsigned int i = 0; i < n; ++i)
    crossed_out[i] = false;
```

of Program 14 with the single line

```
std::fill (crossed_out, crossed_out + n, false);
```

The pre- and postconditions of this standard library function exactly match the ones of our own `fill` function from Page 175. The benefit here is not the saving of one line of code; this saving does not even exist, since we additionally have to `#include <algorithm>` in the beginning of the program.

The benefit is that we eliminate possible sources of error (even a trivial loop has the potential of being wrong), and that we simplify the control flow (see also Section 2.4.8).

3.1.10 Details

Default arguments. Some functions have the property that there are "natural" values for one or more of their formal arguments. For example, when filling an array of underlying type `int`, the value 0 is such a natural value. In such a case, it is possible to specify this value as a *default argument*; this allows the caller of the function to omit the corresponding call argument and let the compiler insert the default value instead. In case of the function `fill` from Program 19, this would look as follows.

```
// PRE:  [first, last) is a valid range
// POST: *p is set to value, for p in [first, last)
void fill (int* first, int* last, int value = 0) {
  // iteration by pointer
  for (int* p = first; p != last; ++p)
    *p = value;
}
```

This function can now be called with either two or three arguments, as follows.

```
int a[5];
fill (a, a+5);     // means: fill (a, a+5, 0)
fill (a, a+5, 1);
```

In general, there can be default values for any number of formal arguments, but these arguments must be at consecutive positions $i, i+1, \ldots, k$ among the k arguments, for some i. The function can then be called with any number of call arguments between $i-1$ and k, and the compiler automatically inserts the default values for the missing call arguments.

A function may have a separate declaration that specifies default arguments, like in the following declaration of `fill`.

```
// PRE:  [first, last) is a valid range
// POST: *p is set to value, for p in [first, last)
void fill (int* first, int* last, int value = 0);
```

In this case, the actual definition must not repeat the default arguments (the actual rules are a bit more complicated, but this is the upshot).

Function declarations and definitions. A function may have several declarations, even with the same declarative regions (the latter is not allowed for variables, see Section 2.4.3). The purpose of a function declaration is to put subsequent code into the function's scope, and there may be several places where this is necessary.

On the other hand, every function can have only one definition, and this is the one all its declarations refer to.

Function signatures. In function declarations, the formal argument names *pname1*,..., *pnamek* can be omitted.

This makes sense since these names are only needed in the function definition. The important information, namely domain and range of the function, are already specified by the argument types and the return type. All these types together form the *signature* of the function.

In `math.h`, we could therefore equivalently write the declaration

```
double pow (double, int);
```

The only problem is that we need the formal argument names to specify pre- and post-conditions, without going to lengthy formulations involving "the first argument" and "the second argument". Therefore, we usually write the formal argument names even in function declarations.

Mathematical functions. Many of the mathematical functions that are available on scientific pocket calculators are also available from the math library `cmath`. The following table lists some of them. All are available for the three floating point number types `float`, `double` and `long double`.

name	function
`std::abs`	$\lvert x\rvert$
`std::sin`	$\sin(x)$
`std::cos`	$\cos(x)$
`std::tan`	$\tan(x)$
`std::asin`	$\sin^{-1}(x)$
`std::acos`	$\cos^{-1}(x)$
`std::atan`	$\tan^{-1}(x)$
`std::exp`	e^x
`std::log`	$\ln x$
`std::log10`	$\log_{10} x$
`std::sqrt`	$\sqrt{x}$

3.1.11 Goals

Dispositional. At this point, you should ...

1) be able to explain the purpose of functions in C++;
2) understand the syntax and semantics of C++ function definitions and declarations;
3) know what the term "procedural programming" means;
4) understand the function `pow` from Program 17 and the functions `fill_n` and `fill` from Program 19;
5) know that formal arguments of pointer type can be used to write array-processing functions, and mutating functions;
6) know why it makes sense to compile function definitions separately, and to put functions into libraries.

Operational. In particular, you should be able to ...

(G1) give two reasons why it is desirable to subdivide programs into functions;

(G2) find pre- and postconditions for given functions, where the preconditions should be as *weak* as possible, and the postconditions should be as *strong* as possible;

(G3) find syntactical and semantical errors in function definitions, and in programs that contain function definitions;

(G4) evaluate given function call expressions;

(G5) write (mutating) functions for given tasks, and write programs for given tasks that use functions;

(G6) subdivide a given task into small subtasks, and write a program for the given task that uses functions to realize the subtasks;

(G7) build a library on your platform, given that you are told the necessary technical details.

3.1.12 Exercises

Exercise 68 *Find pre- and postconditions for the following functions.* (G2)(G4)

a)
```
int f (double i, double j, double k)
{
  if (i > j)
    if (i > k)
      return i;
    else
      return k;
  else
    if (j > k)
      return j;
    else
      return k;
}
```

b)
```
double g (int i, int j)
{
  double r = 0.0;
  for (int k = i; k <= j; ++k)
    r += 1.0 / k;
  return r;
}
```

Exercise 69 *What are the problems (if any) with the following functions? Fix them and find appropriate pre- and postconditions.* (G2)(G3)

a)
```
bool is_even (int i)
{
  if (i % 2 == 0) return true;
}
```

b)
```
double inverse (double x)
{
  double result;
  if (x != 0.0)
    result = 1.0 / x;
  return result;
}
```

Exercise 70 *What is the output of the following program, depending on the input number* `i`*? Describe the output in mathematical terms, ignoring possible over- and underflows.* (G4)

```
#include<iostream>

int f (int i)
{
  return i * i;
}

int g (int i)
{
  return i * f(i) * f(f(i));
}

void h (int i)
{
  std::cout << g(i) << "\n";
}

int main()
{
  int i;
  std::cin >> i;
  h(i);

  return 0;
}
```

Exercise 71 *Find three problems in the following program.* (G3)(G4)

```
#include<iostream>

double f (double x)
{
  return g(2.0 * x);
}

bool g (double x)
{
  return x % 2.0 == 0;
}

void h ()
{
  std::cout << result;
}

int main()
```

```
{
  double result = f(3.0);
  h();

  return 0;
}
```

Exercise 72 *Simplify the program from Exercise 52 by using the library function* `std::pow`*.* (G5)

Exercise 73 *Assume that on your platform, the library function* `std::sqrt` *is not very reliable. For* x *a value of type* `double` *(*$x \geq 0$*), we let* $s(x)$ *be the value returned by* `std::sqrt(`*expr*`)`*, if expr has value* x*, and we assume that we only know that for some positive value* $\varepsilon \leq 1/2$*, the relative error satisfies*

$$\frac{|s(x) - \sqrt{x}|}{\sqrt{x}} \leq \varepsilon, \quad \forall x.$$

How can you change Program 26 such that it correctly works under this relative error bound? You may assume that the floating point number system used on your platform is binary, and that all values of type `unsigned int` *are exactly representable in this system. (This is a theory exercise.)* (G5)

Exercise 74

a) *Write a function*

```
// POST: return value is true if and only if n is prime
bool is_prime (unsigned int n);
```

and use this function in a program to count the number of twin primes *in the range* $\{2, \ldots, 10000000\}$ *(two up to ten millions). A twin prime is a pair of numbers* $(i, i+2)$ *both of which are prime.*

b) *Is the approach of a) the best (most efficient) one to this problem? If you can think of a better approach, you are free to implement it instead of the one outlined in a).*

(G5)

Exercise 75 *The function* `pow` *in Program 17 needs* $|e|$ *multiplications to compute* b^e*. Change the function body such that less multiplications are performed. You may use the following fact. If* $e \geq 0$ *and* e *has binary representation*

$$e = \sum_{i=0}^{\infty} b_i 2^i,$$

then

$$b^e = \prod_{i=0}^{\infty} \left(b^{2^i}\right)^{b_i}.$$

(G5)

Exercise 76 *Write a program* swap.C *that defines and calls a function for interchanging the values of two* int *objects. The program should have the following structure.*

```
#include<iostream>

// your function definition goes here

int main() {
  // input
  std::cout << "i =? ";
  int i; std::cin >> i;

  std::cout << "j =? ";
  int j; std::cin >> j;

  // your function call goes here

  // output
  std::cout << "Values after swapping: i = " << i
            << ", j = " << j << ".\n";

  return 0;
}
```

Here is an example run of the completed program:

```
i =? 5
j =? 8
Values after swapping: i = 8, j = 5.
```

(G5)

Exercise 77 *Modify the program* sort_array.C *from Exercise 60 in such way that the resulting program* sort_array2.C *defines and calls a function*

```
// PRE: [first, last) is a valid range
// POST: the elements *p, p in [first, last) are
//       in ascending order
void sort (int* first, int* last);
```

to perform the sorting of the array into ascending order. It may be tempting (but not allowed for obvious reasons) to use std::sort *or similar standard library functions in the body of the function* sort *that is to be defined. It* is *allowed, though,*

to compare the efficiency of your `sort` *function with that of* `std::sort` *(which has the same pre- and postconditions and can be used after* `include<algorithm>`*).*

For this exercise, it is desirable (but not strictly necessary) to use pointer increment (`++p`*) as the only operation on pointers (apart from initialization and assignment, of course). If you suceed in doing so, your sorting function has the potential of working for containers that do not offer random access (see also Section 3.1.7).* (G5)

Exercise 78 *A* perpetual calendar *can be used to determine the weekday (Monday, ..., Sunday) of any given date. You may for example know that the Berlin wall came down on November 9, 1989, but what was the weekday? (It was a Thursday.) Or what is the weekday of the 1000th anniversary of the Swiss confederation, to be celebrated on August 1, 2291? (Luckily, this big party will be on a Saturday.)*

a) The task is to write a program that outputs the weekday (Monday, ..., Sunday) of a given input date.

   Identify a set of subtasks to which you can reduce this task. Such a set is not unique, of course, but all inividual subtasks should be small (so small that they could be realized with few lines of code). It is of course possible for a subtask in your set to reduce to other subtasks. (Without giving away anything, one subtaks that you certainly need is to determine whether a given year is a leap year).

b) Write a program `perpetual_calendar.C` *that reads a date from the input and outputs the corresponding weekday. The range of dates that the program can process should start no later than January 1, 1900 (Monday). The program should check whether the input is a legal date, and if not, reject it. An example run of the program might look like this.*

```
day =? 13
month =? 11
year =? 2007
Tuesday
```

   To structure your program, implement the subtasks from a) as functions, and put the program together from these functions.

(G5)(G6)

Exercise 79 *Build a library on your platform from the files* `math.h` *and* `math.C` *in Program 23 and Program 24. Use this library to generate an executable program from Program 25.* (G5)(G7)

Exercise 80

a) *Implement the following function and test it. You may assume that the type* `double` *complies with the IEEE standard 754, see Section 2.5.6. The function is only required to work correctly, if the nearest integer is in the value range of the type* `int`. (G5)

```
// POST: return value is the integer nearest to x
int round (double x);
```

b) *The postcondition of the function does not say what happens if there are two nearest integers. Specify the behavior of your implementation in the postcondition of your function.* (G2)

c) Add a declaration of your function to the file `math.h` (Program 23) and a definition to `math.C` (Program 24). Build a library from these two files, and rewrite your test function from a) to call the library version of the function `round`. (G7)

3.1.13 Challenges

Exercise 81 *A Sudoku puzzle is posed on a grid of 9×9 cells, subdivided into 9 square boxes of 3×3 cells each. Some grid cells are already filled by numbers between* 1 *and* 9*; the goal is to fill the remaining cells by numbers between* 1 *and* 9 *in such a way that within each row, column, and box of the completed grid, every number occurs exactly once. Here is an example of a Sudoku puzzle:*

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

In solving the puzzle, one may try to deduce from the already filled numbers that exactly one *number is a candidate for a suitable empty cell. Then this number is filled into the cell, and the deduction process is repeated. There are two situations where such a deduction for the cell in row* r */ column* c *and number* n *is particularly easy and follows the Sherlock Holmes approach (*How often have I said to you that when you have eliminated the impossible, whatever remains, however improbable, must be the truth?*).*

1. *All numbers distinct from* n *already appear somewhere in the same row, column, or 3x3 box. This necessarily means that the cell has to be filled with* n*, since we have eliminated all other* numbers *as impossible.*

2. *All other cells in the same row, or in the same column, or in the same 3x3 box are already known not to contain* n. *Again, the cell has to be filled by* n *then, since we have eliminated all other* cells *for the number* n *within the row, column, or box.*

Write a program sudoku.C *that takes as input a Sudoku puzzle in form of a sequence of* 81 *numbers between* 0 *and* 9 *(the grid numbers given row by row, where* 0 *indicates an empty cell). The numbers might be separated by whitespaces, so that the Sudoku puzzle from above could conveniently be encoded like this in a file:*

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

4 0 0  8 0 0  0 0 0
0 2 0  0 0 0  0 1 0
0 0 0  0 0 9  0 0 5

0 0 4  0 0 7  3 0 0
7 3 0  0 2 0  0 6 0
0 6 5  0 0 4  0 0 0
```

The program should now try to solve the puzzle by using only the two Sherlock-Holmes-type deductions from above. The output should be a (partially) completed grid that is either the solution to the puzzle, or the unique (why?) partial solution in which no Sherlock-Holmes-type deductions apply anymore (again, empty cells should be indicated by the digit 0*).*

In the above example, the output of a correct program will be the solution:

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

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

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

For reading the input from a file, it can be convenient to redirect the standard input to the file containing the puzzle data. For checking whether any Sherlock-Holmes-type deductions apply, it can be useful to maintain (and update) for any triple (r, c, n) *the information whether* n *is still a possible candidate for the cell in row r / column c.*

You will discover that most Sudoku puzzles that typically appear in newspapers can be solved by your program and are therefore easy, even if they are labeled as medium *or* hard*.*

Hint: *It is advisable not to optimize for efficiency here, since this will only lead to more complicated and error-prone code. Given the very small problem size, such optimizations won't have a noticeable effect anyway.*