3.1 A first C++ function

This section introduces C++ functions. 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 the context of floating point number systems in Section 2.5, powers (of the base $\beta$) are ubiquitous. Even if you have not solved Exercise 45, you can imagine that computing powers is a basic and frequently used operation in many contexts. In C++, functions are used to encapsulate such operations, making it easy to invoke them many times, with different parameters, and from different programs.

We have already seen quite a number of functions, since the main function of every C++ program is a special function (Section 2.1.3).

Program 13 shows how functions can be used in general. 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.

```cpp
#include <iostream>

// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
double pow (double b, int e) {
    double result = 1.0;
    if (e < 0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e; ++i) result *= b;
    return result;
}

int main() {
    std::cout << pow( 2.0, -2) << "\n"; // outputs 0.25
    std::cout << pow( 5.0, 1) << "\n"; // outputs 5
    std::cout << pow( 5.0, 4) << "\n"; // outputs 625
}
```
3.1 A FIRST C++ FUNCTION

Program 13: 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 c, b^c is obtained from 1 by c-fold multiplication with b. This is what the for loop does. The case of negative c can be handled by the formula b^c = (1/b)^-c: after inverting b and negating c in the for statement, we have an equivalent problem with a positive exponent. The latter only works if b ≠ 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. In case of the function pow, the precondition

```cpp
// PRE: e >= 0 && b != 0.0
```

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

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

tells us the function value, depending on the parameter. 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 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 parameters 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 parameters 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 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.

3.1.2 Function definitions

Lines 8-18 of Program 13 define a function called pow. The syntax of a function definition is as follows.

**T**<br>

```cpp
T \text{pow} ( \text{T1 parameter}_1, \text{T2 parameter}_2, \ldots, \text{TN parameter}_N )
```

This defines a function called pow, with return type T, and with formal parameters parameter_1, parameter_2, ..., parameter_N of type T1, ..., TN, and with a function body block.

Syntactically, T and T1, ..., TN are type names, as well as parameter_1, ..., parameter_N are identifiers (Section 2.1.8), and block is a block, a sequence of statements enclosed by curly braces (Section 2.4.3).

We can think of the formal parameters as placeholders for the actual parameters that are supplied during a function call.

Function definitions may appear inside blocks, other functions, or control statements. They may appear inside namespaces, though, or at global scope, like in a program 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 13, 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.

```cpp
frame (expression1, ..., expressionN)
```

Here, `expression1, ..., expressionN` must be expressions of types whose values can be converted to the formal parameter types `T1, ..., TN`. These expressions are the call parameters. For all types we know so far, the call parameters as well as the function call itself are values. The type of the function call is the function's return type `T`.

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

The evaluation of a function call terminates as soon as a return statement is reached, see Section 2.1.13. This return statement must be of the form

```cpp
return expression
```

where `expression` is an expression of a type whose value 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 parameter 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.

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):

```cpp
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";
```

The type `void` has empty value range, and there are no literals, variables, or formal function parameters 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 expression of type `void`, or return statements of the form

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

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

```cpp
int main () {
  double b = 2.0;
  int e = -2;
  std::cout << pow(b, e); // outputs 0.25
  std::cout << b; // outputs 2
  std::cout << e; // outputs -2
  return 0;
}
```

The values of the variables `b` and `e` 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`, 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),
their scope would exclude the body of pow, since that body is in the potential scopes of
redemptions of the names b and e (the formal parameters), see Section 2.4, 3.
Since the formal parameters of a function have local scope, they also have automatic
storage duration (Section 2.4, 3). This means that we get a "fresh" set of formal pa-
rameters everytime 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
parameters, unless they appear in a nested block. This makes sense since otherwise, it
would be possible to irrevocably hide the name of a formal parameter. Therefore, we
cannot write

\[
\text{int } f \text{ (int } i \text{)} \\
\{ \\
\quad \text{int } i = 5; \text{ // invalid; } i \text{ hides formal parameter} \\
\quad \text{return } i; \\
\} \\
\]
while the following is not recommended but legal,

\[
\text{int } f \text{ (int } i \text{)} \\
\{ \\
\quad \{ \\
\quad \quad \text{int } i = 5; \text{ // ok; } i \text{ is local to nested block} \\
\quad \} \\
\quad \text{return } i; \text{ // the formal parameter} \\
\} \\
\]
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 only once.

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

\[
\text{#include } \langle \text{iostream} \rangle \\
\text{int main()} \\
\{ \\
\quad \text{std::cout }<< f(1); \text{ // variable } f \text{ undeclared} \\
\quad \text{return } 0; \\
\} \\
\]

3.1.6 **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 a *modularization*,
a subdivision of the program into independent parts.

The power function pow from Program 13 is certainly of general purpose. In order to
make it available to all our programs, we can simply put the function definition into a
separate source code file pow.c, say, in our working directory.

\[
1 \text{ // PRE: } e >= 0 \text{ // } b^e \\
2 \text{ // POST: return value is } b^e \\
3 \text{ double pow (double } b, \text{ int } e \text{)} \\
4 \{ \\
5 \quad \text{double result }= 1.0; \\
\]

3.1, A FIRST C++ FUNCTION

6 if (e < 0) {
7     return 1.0/b;
8     e = -e;
9 }
10 for (int i =0; i<e; ++i) result *= b;
11 return result;
12 }

Program 14: 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 #include <iostream>
4 #include "pow.C"
5 #include <cmath>
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    return 0;
15 }

Program 15: 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 15, 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 #include <iostream>
4 #include "pow.h"
5 #include <cmath>
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    return 0;
15 }

Program 16: 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.
3.1. A FIRST C++ FUNCTION

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 pro 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 pro and postconditions are of critical importance, since they may be the only available information 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.²

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, 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 it is 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 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 6 schematically shows how object code files, a library and finally an executable program are obtained from a number of sourcecode files.

```
#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
3.1. A FIRST C++ FUNCTION

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 undeclared names, see Section 2.1.2. 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 only pow).

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 17: prog/math.h

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

Program 18: prog/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     return 0;
15 }

Program 19: prog/callpow4.C

3.1.7 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 as in Program 7. You might have realized this much earlier, but when we are trying to find some proper divisor of a natural number n >= 2, it is sufficient to search in the range [2, sqrt(n)]. 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 integer, we even get a bound of |sqrt(n)|.

The program prime.C could therefore be written more efficiently as in Program 20, using the function std::sqrt from the library cmath, whose parameter and return types are double.
3.1, *A FIRST C++ FUNCTION*

```cpp
1 // Program: prime2.C
2 // Test if a given natural number is prime.
3 #include <iostream>
4 #include <cmath>
5 int main ()
6 {
7     // Input
8     unsigned int n;
9     std::cin >> n;
10
11     // Computation: test possible divisors d up to sqrt(n)
12     unsigned int bound = (unsigned int)(std::sqrt(n));
13     unsigned int d;
14     for (d = 2; d <= bound && n % d != 0; ++d);
15
16     // Output
17     if (d <= bound)
18         std::cout << "d is a divisor of n in \{2, ..., \sqrt(n)\};";
19     else
20         std::cout << "no proper divisor found";
21     std::cout << n << " is prime.\n";
22
23     return 0;
24 }
```

Program 20: `prog/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 \(\sqrt{n}\), 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` correctly works for squares. For example, `std::sqrt(121)` must return 11 (a little more would hurt), but not 10.99999999, 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.

3.1.8 Details

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 make subsequent code visible in 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 parameter names `param1, ..., paramN` 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 parameter 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

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

The only problem is that we need the formal parameter names to specify pre- and postconditions, without going to lengthy formulations involving "the first argument" and "the second argument". Therefore, we usually write the formal parameter 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`.

<table>
<thead>
<tr>
<th>name</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>std::abs</code></td>
<td>`</td>
</tr>
<tr>
<td><code>std::sin</code></td>
<td><code>\sin(x)</code></td>
</tr>
<tr>
<td><code>std::cos</code></td>
<td><code>\cos(x)</code></td>
</tr>
<tr>
<td><code>std::tan</code></td>
<td><code>\tan(x)</code></td>
</tr>
<tr>
<td><code>std::asin</code></td>
<td><code>\sin^{-1}(x)</code></td>
</tr>
<tr>
<td><code>std::acos</code></td>
<td><code>\cos^{-1}(x)</code></td>
</tr>
<tr>
<td><code>std::atan</code></td>
<td><code>\tan^{-1}(x)</code></td>
</tr>
<tr>
<td><code>std::exp</code></td>
<td><code>e^x</code></td>
</tr>
<tr>
<td><code>std::log</code></td>
<td><code>\ln x</code></td>
</tr>
<tr>
<td><code>std::log10</code></td>
<td><code>\log_{10} x</code></td>
</tr>
<tr>
<td><code>std::sqrt</code></td>
<td><code>\sqrt{x}</code></td>
</tr>
</tbody>
</table>
3.1.9 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) understand the function pow from Program 13;
4) 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) find pre- and postconditions for given functions;
(G2) find syntactical and semantical errors in function definitions, and in programs that contain function definitions;
(G3) evaluate function call expressions;
(G4) write functions and programs that use your own functions and standard library functions;
(G5) build a library on your platform, given that you know the necessary technical details.

3.1.10 Exercises

Exercise 46 Find pre- and postconditions for the following functions. (G1)(G3)

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;
   }

Exercise 47 What are the problems (if any) with the following functions? Fix them and find appropriate pre- and postconditions. (G1)(G2)

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 48 What is the output of the following program, depending on the input number a? Describe the output in mathematical terms, ignoring possible over and underflows. (G3)

#include <iostream>

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

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

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

int main()
   {  
     int a;


Exercise 50: Simplify the program from Exercise 45 by using the library function `std::pow`.

Exercise 51: Assume that on your platform, the library function `std::sqrt` is not very reliable. For 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 20 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.)

Exercise 52: 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 \([2, \ldots, 1000000]\) (two up to one million). A twin prime is a pair of numbers \((i, i+2)\) both of which are prime.

Exercise 53: The function pow in Program 19 needs \(\lceil n \rceil\) multiplications to compute \(b^n\). Change the function body such that less multiplications are performed. You may use the following fact. If \(c \geq 0\) and \(c\) has binary representation

\[
c = \sum_{i=0}^{\infty} b_i 2^i,
\]

then

\[
b^n = \prod_{i=0}^{\infty} (b^2)^{b_i}.
\]

Exercise 54: Build a library on your platform from the files `math.h` and `math.c` in Program 17 and Program 18. Use this library to generate an executable program from Program 19.

Exercise 55:

a) Implement the following function and test it. You may assume that the type double complies with the IEEE standard 754, see Section 2.3.6. The function \(\text{round}(x)\) is only required to work correctly, if the nearest integer is in the value range of the type int.

```
// 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 definition.

c) Add a declaration of your function to the file `math.h` (Program 17 and a definition to `math.c` (Program 18). Build a library from these two files, and rewrite your test function from a) to call the library version of the function `round`.

---

3.1, A FIRST C++ FUNCTION

```cpp
std::cin >> a;
b(a);
return 0;
)
```

Exercise 49: Find three problems in the following program.

```
#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;
)
```

---

CHAPTER 3, FUNCTIONS

Exercise 52: 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 \([2, \ldots, 1000000]\) (two up to one million). A twin prime is a pair of numbers \((i, i+2)\) both of which are prime.

Exercise 53: The function pow in Program 19 needs \(\lceil n \rceil\) multiplications to compute \(b^n\). Change the function body such that less multiplications are performed. You may use the following fact. If \(c \geq 0\) and \(c\) has binary representation

\[
c = \sum_{i=0}^{\infty} b_i 2^i,
\]

then

\[
b^n = \prod_{i=0}^{\infty} (b^2)^{b_i}.
\]

Exercise 54: Build a library on your platform from the files `math.h` and `math.c` in Program 17 and Program 18. Use this library to generate an executable program from Program 19.

Exercise 55:

a) Implement the following function and test it. You may assume that the type double complies with the IEEE standard 754, see Section 2.3.6. The function \(\text{round}(x)\) is only required to work correctly, if the nearest integer is in the value range of the type int.

```
// 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 definition.

c) Add a declaration of your function to the file `math.h` (Program 17 and a definition to `math.c` (Program 18). Build a library from these two files, and rewrite your test function from a) to call the library version of the function `round`. 

---
3.2 Recursion

This section introduces recursive functions, functions that directly or indirectly call themselves. You will see that recursive functions are very natural in many situations, and that they lead to compact and readable code close to mathematical function definitions. We will also explain how recursive function calls are processed, and how recursion can (in principle) be replaced with iteration.

3.2.1 A warm-up

Many mathematical functions are naturally defined recursively, meaning that the function to be defined appears in its own definition. For example, the number n! can recursively be defined as follows,

\[ n! := \begin{cases} 
1, & \text{if } n \leq 1 \\
n \cdot (n-1)!, & \text{if } n > 1.
\end{cases} \]

In C++ we can also have recursive functions: a function may call itself. This is nothing exotic, since after all, a function call is just an expression that can in principle appear anywhere in the function's scope, and that scope includes the function body. Here is a recursive function for computing n!; in fact, this definition exactly matches the mathematical definition from above.

```c
// POST: return value is n!
unsigned int fac(unsigned int n)
{
    if (n <= 1) return 1;
    return n * fac(n-1); // n > 1
}
```

Here, the expression fac(n-1) is a recursive call of fac.

Infinite recursion. With recursive functions, we have the same issue as with loops (Section 2.4.2): it is easy to write down function calls whose evaluation does not terminate. Here is the shortest way of creating an infinite recursion: define the function

```c
void f() {
    f();
}
```

with no parameters and evaluate the expression f(). The reason for non-termination is clear: the evaluation of f() consists of an evaluation of f() which consists of an evaluation of f() which... you get the picture.

Like for loops, the function definition has to make sure that progress towards termination is made in every function call. For the function fac above, this is the case: each time fac is called recursively, the value of the call parameter becomes smaller, and when the value reaches 1, no more recursive calls are performed: we say that the recursion "bottoms out".

3.2.2 The call stack

Let's try to understand what exactly happens during the evaluation of fac(3), say. The formal parameter n is initialized with 3, and since this is greater than 1, the statement return n * fac(n-1); is executed next. This first evaluates the expression n + fac(n-1) and in particular the right operand fac(n-1). Since n-1 has value 2, the formal parameter n is therefore initialized with 2.

But wait: what is "the" formal parameter? Automatic storage duration implies that each function call has its own fresh instance of the formal parameter, and the lifetime of this instance is the respective function call. In evaluating f(n-1), we therefore get a new instance of the formal parameter n, on top of the previous instance from the call f(n) (that has not yet terminated). But which instance of do we use in the evaluation of f(n-1)? Quite naturally, it will be the new one, the one that "belongs" to the call f(n-1). This rule is in line with the general scope rules from Section 2.4.3: the relevant declaration is always the most recent one that is still visible.

The technical realization of this is very simple. Every time a function is called, the call parameter is evaluated, and the resulting value is put on the call stack which is simply a region in the computer's memory.\footnote{If the function has several parameters, several values are put on the call stack; to keep the description simple, we concentrate on the case of one parameter.}

Like a stack of papers on your desk, the call stack has the property that the object that came last is "on top". Upon termination of a function call, the top object is taken off the stack again. Whenever a function call accesses or changes its formal parameter, it does so by accessing or changing the corresponding object on top of the stack.

This has all the properties we want: every function call works with its own instance of the formal parameter; when it calls another function (or the function itself recursively), this instance becomes temporarily hidden, until the nested call has terminated. At that point, the instance reappears on top of the stack and allows the original function call to work with it again.

Table 4 shows how this looks like for f(3), assuming that the right operand of the multiplication operator is always evaluated first. Putting an object on the stack "pushes" it, and taking the top object of "pops" it.

Because of the call stack, infinite recursions do not only consume time but also memory. Unlike infinite loops, they usually lead to a program abortion as soon as the memory reserved for the call stack is full.
3.2, **RECURSION**

<table>
<thead>
<tr>
<th>call stack (bottom ⟷ top)</th>
<th>evaluation sequence</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>⋯</td>
<td>⋯</td>
<td>push 3</td>
</tr>
<tr>
<td>n; 3</td>
<td>n * fac(n-1)</td>
<td></td>
</tr>
<tr>
<td>n; 3</td>
<td>n * fac(2)</td>
<td>push 2</td>
</tr>
<tr>
<td>n; 3; 2</td>
<td>n * (n * fac(n-1))</td>
<td></td>
</tr>
<tr>
<td>n; 3; 2</td>
<td>n * (n * fac(1))</td>
<td>push 1</td>
</tr>
<tr>
<td>n; 3; 2; 1</td>
<td>n * (n + 1)</td>
<td>pop</td>
</tr>
<tr>
<td>n; 3; 2</td>
<td>n * (2 + 1)</td>
<td></td>
</tr>
<tr>
<td>n; 2</td>
<td>3 * 2</td>
<td>pop</td>
</tr>
<tr>
<td>n; 3</td>
<td>6</td>
<td>pop</td>
</tr>
</tbody>
</table>

Table 4: The call stack, and how it evolves during an evaluation of fac(3); the respective value of n to use is always the one on top

### 3.2.3 Basic practice

Let us consider two more simple recursive functions that are somewhat more interesting than fac. They show that recursive functions are particularly amenable to correctness proofs of their postconditions, and this makes them attractive. On the other hand, we also see that it is easy to write innocent-looking recursive functions that are very inefficient to evaluate.

**Greatest common divisor.** Consider the problem of finding the greatest common divisor \( \gcd(a, b) \) of two natural numbers \( a, b \). This is defined as the largest natural number that divides both \( a \) and \( b \) without remainder. In particular, \( \gcd(n, 0) = \gcd(0, n) = n \) for \( n > 0 \); let us also define \( \gcd(0, 0) = 0 \).

The **Euclidean algorithm** finds \( \gcd(a, b) \), based on the following

**Lemma 1** If \( b > 0 \), then

\[
\gcd(a, b) = \gcd(b, a \mod b).
\]

**Proof.** Let \( k \) be a divisor of \( b \). From

\[
a = (a \div b) b + a \mod b
\]

it follows that

\[
\frac{a}{k} = (a \div b) \cdot \frac{b}{k} + \frac{a \mod b}{k}.
\]

Since \( a \div b \) and \( b/k \) are integers, we get

\[
\frac{a \mod b}{k} \in \mathbb{N} \iff \frac{a}{k} \in \mathbb{N}.
\]

In words, if \( k \) is a divisor of \( b \), then \( k \) divides \( a \) if and only if \( k \) divides \( a \mod b \). This means, the divisors of \( a \) and \( b \) are exactly the divisors of \( b \) and \( a \mod b \). This proves that \( \gcd(a, b) \) and \( \gcd(b, a \mod b) \) are equal.

Here is the corresponding C++ function for computing the greatest common divisor of two unsigned int values, according to the Euclidean algorithm.

```cpp
// POST: return value is the greatest common divisor of a and b
unsigned int gcd(unsigned int a, unsigned int b)
{
    if (b == 0) return a;
    return gcd(b, a % b); // b != 0
}
```

The Euclidean algorithm is very fast. We can easily call it for any unsigned int values on our platform, without noticing any delay in the evaluation.

**Correctness and termination.** For recursive functions, it is often very easy to prove that the postcondition is correct, by using the underlying mathematical definition directly (like \( n! \) for fac), or by using some facts that follow from the mathematical definition (like Lemma 1 for gcd).

The correctness proof must involve a termination proof, so let’s start with this: any call to \( \gcd \) terminates, since the value \( b \) of the second parameter is bounded from below by 0 and gets smaller in every recursive call (we have \( a \mod b < b \)).

Given this, the correctness of the postcondition follows from Lemma 1 by induction on \( b \). For \( b = 0 \), this is clear. For \( b > 0 \), we inductively assume that the postcondition is correct for all calls to \( \gcd \) where the second parameter has value \( b' < b \). Since \( b' = a \mod b \) satisfies \( b' < b \), we may assume that the call \( \gcd(b, a \mod b) \) correctly returns \( \gcd(b, a \mod b) \). But by the lemma, \( \gcd(b, a \mod b) = \gcd(a, b) \), so the statement return \( \gcd(b, a \mod b) \) correctly returns \( \gcd(a, b) \).

**Fibonacci numbers.** The sequence 0, 1, 1, 2, 3, 5, 8, 13, 21, … of Fibonacci numbers is one of the most famous sequences in mathematics. Formally, the sequence is defined as follows.

\[
F_0 := 0,
F_1 := 1,
F_n := F_{n-1} + F_{n-2}, \quad n > 1.
\]
This means, every element of the sequence is the sum of the two previous ones. From this definition, we can immediately write down a recursive C++ function for computing Fibonacci numbers, getting termination and correctness for free.

```cpp
// POST: return value is the n-th Fibonacci number F_m
unsigned int fib (unsigned int n)
{
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib(n-1) + fib(n-2); // n > 1
}
```

If you write a program to compute the Fibonacci number \( F_n \) using this function, you will notice that somewhere between \( n = 30 \) and \( n = 50 \), the program becomes very slow. You even notice how much slower it becomes when you increase \( n \) by just 1.

The reason is that the mathematical definition of \( F_n \) does not lead to an efficient algorithm, since all values \( F_i, \) \( i \leq n - 1 \), are repeatedly computed, some of them extremely often. You can for example check that the call to \( \text{fib}(50) \) computes \( F_{48} \) already twice (once directly in \( \text{fib}(n-2) \)), and once indirectly from \( \text{fib}(n-1) \). \( F_{47} \) is computed three times, \( F_{46} \) five times, and \( F_{45} \) eight times (do you see a pattern?).

### 3.2.4 Recursion versus iteration

From a strictly functional point of view, recursion is superfluous, since it can be simulated through iteration (and a call stack explicitly maintained by the program; we don't know yet how to implement a stack, but trust us that it can be done). We don't have the means to prove this here, but we want to show it for the recursive functions that we have seen in the previous section.

The function gcd is very easy to write iteratively, since it is tail-end recursive. This means that there is only one recursive call, and that one appears at the very end of the function body. Tail-end recursion can be replaced by a simple loop that iteratively updates the formal parameters until the termination condition is satisfied. In the case of gcd, this update corresponds to the transformation \((a, b) \rightarrow (b, a \mod b)\).

```cpp
// PRE: (a, b) \neq (0, 0)
// POST: return value is the greatest common divisor of a and b
unsigned int gcd2 (unsigned int a, unsigned int b)
{
    while (b != 0) {
        unsigned int a_prev = a;
        a = b;
        b = a_prev % b;
    }
    return a;
}
```

Again, this non-recursive version \( \text{fib2} \) is substantially longer and more difficult to understand than \( \text{fib} \), but this time there is a benefit: \( \text{fib2} \) is much faster, since it computes every number \( F_i, 1 \leq i \leq n \) exactly once. While we would grow old in waiting for the call \( \text{fib}(50) \) to terminate, \( \text{fib2}(50) \) gives us the answer in no time. Unfortunately, this answer may be incorrect, since \( F_{50} \) could exceed the value range of the type unsigned int.

In this case we would prefer \( \text{fib2} \) over \( \text{fib} \), simply since \( \text{fib} \) is too inefficient for practical use. The more complicated function definition of \( \text{fib2} \) is a moderate price to pay for the speedup that we get.

### 3.2.5 Primitive recursion

Roughly speaking, a mathematical function is primitive recursive if it can be written as a C++ function \( f \) in such a way that \( f \) neither directly nor indirectly calls itself with call parameters depending on \( f \). For example,

```cpp
unsigned int f (unsigned int n)
{
    if (n == 0) return 1;
    return f(f(n-1) - 1);
}
```

is not allowed, since \( f \) recursively calls itself with a parameter depending on \( f \). This does not mean that the underlying mathematical function is not primitive recursive, it just...
3.2 Recursion

means that we have taken the wrong C++ function. Indeed, the above f implements the mathematical function satisfying \( f(n) = 1 \) for all \( n \), and this function is obviously

primitive recursive.

In the early 20th century, it was believed that the functions whose values can in

principle be computed by a machine are exactly the primitive recursive ones. Indeed,

the function values one computes in practice (including \( \text{gcd}(a,b) \) and \( F_n \)) come from

primitive recursive functions.

It later turned out that there are computable functions that are not primitive recursive.

A simple and well-known example is the binary Ackermann function \( A(m, n) \),

defined by

\[
A(m, n) = \begin{cases} 
  n + 1, & \text{if } m = 0 \\
  A(m-1, 1), & \text{if } m > 0, n = 0 \\
  A(m-1, A(m, n-1)), & \text{if } m > 0, n > 0.
\end{cases}
\]

The fact that this function is not primitive recursive requires a proof (that we don’t
give here). As already noted above, it is necessary but not sufficient that this definition

definitely uses \( A \) with a parameter that depends on \( A \).

It may not be immediately clear that the corresponding C++ function

```cpp
// POST: return value is the Ackermann function value A(m, n)
unsigned int A(unsigned int m, unsigned int n) {
  if (m == 0) return n + 1;
  if (n == 0) return A(m-1, 1);
  return A(m-1, A(m, n-1));
}
```

always terminates, but Exercise 57 asks you to show this. Table 5 lists some Ackermann

function values. For \( m \leq 3 \), \( A(m, n) \) looks quite moderate, but starting from \( m = 4 \),

the values get extremely large. You can still compute \( A(4, 1) \), although this takes surprisingly

long already. You might be able to compute \( A(4, 2) \); after all, \( 2^{65536} - 3 \) has “only” around

20,000 decimal digits. But the call to \( A(4, 3) \) will not terminate within any observable

period.

It can be shown that \( A(n, n) \) grows faster than any primitive recursive function

in \( n \) (and this is a proof that \( A \) cannot be primitive recursive). Recursion is a powerful

but also dangerous tool, since it is easy to encode (too) complicated computations with

very few lines of code.

3.2.6 Lindenmayer systems

So far it seems that the benefit of using recursive functions is that we may get (mildly)
simpler formulations, compared to equivalent iterative versions. In this section we want
to present an application in which recursion is predominant and much more difficult to

avoid (an iterative version would indeed require an explicit stack). As a bonus, this

application lets us draw beautiful pictures.

<table>
<thead>
<tr>
<th>n</th>
<th>( 0 )</th>
<th>( 1 )</th>
<th>( 2 )</th>
<th>( 3 )</th>
<th>( \ldots )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( 1 )</td>
<td>( 2 )</td>
<td>( 3 )</td>
<td>( \ldots )</td>
<td>( n+1 )</td>
</tr>
<tr>
<td>1</td>
<td>( 2 )</td>
<td>( 3 )</td>
<td>( 4 )</td>
<td>( \ldots )</td>
<td>( n+2 )</td>
</tr>
<tr>
<td>2</td>
<td>( 3 )</td>
<td>( 5 )</td>
<td>( 7 )</td>
<td>( \ldots )</td>
<td>( 2n+3 )</td>
</tr>
<tr>
<td>3</td>
<td>( 5 )</td>
<td>( 13 )</td>
<td>( 29 )</td>
<td>( 61 )</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>4</td>
<td>( 13 )</td>
<td>( 65533 )</td>
<td>( 2^{65536} - 3 )</td>
<td>( 2^{65534} - 3 )</td>
<td>( \ldots )</td>
</tr>
</tbody>
</table>

Table 5: Some values of Ackermann's function

Let us first fix an alphabet \( \Sigma \) which is simply a finite set of symbols, for example
\( \Sigma = \{F, +, -\} \). Let \( \Sigma^* \) denote the set of all words that we can form from symbols in \( \Sigma \).
For example, \( F + F \in \Sigma^* \).

Next, we fix a function \( P : \Sigma \rightarrow \Sigma^* \). P maps every symbol to a word, and these are
the productions. We might for example have the productions

\[
\sigma \mapsto P(\sigma) \\
P \mapsto F + F + \\
+ \mapsto + \\
- \mapsto -
\]

Finally, we fix an initial word \( s \in \Sigma^* \), for example \( s = F \).

The triple \( \mathcal{L} = (\Sigma, P, s) \) is called a Lindenmayer system. Such a system generates
an infinite sequence of words \( s = w_0, w_1, \ldots \) as follows. To get the next word \( w_{i+1} \) from
the previous word \( w_i \), we simply substitute all symbols in \( w_i \) by their productions.
In our example, this yields

\[
w_0 = F, \\
w_1 = F + F +, \\
w_2 = F + F + F + F +, \\
w_3 = F + F + F + F + F + F + F + F + + \\
\vdots
\]

The next step is to "draw" these words, and this gives the pictures we were talking
about.

Turtle graphics. Imagine a turtle sitting at some point \( p \) on a large piece of paper, with
its head pointing in some direction, see Figure 7 (left). The turtle can understand the
Figure 7: The turtle before and after processing the command sequence \( F + F + \)

commands \( F, +, \) and \(-\). \( F \) means "move one step forward", \( + \) means "turn counterclockwise by an angle of 90 degrees", and \(-\) means "turn clockwise by an angle of 90 degrees". The turtle can process any sequence of such commands, by executing them one after another. We are interested in the resulting path taken by the turtle on the piece of paper. The path generated by the command sequence \( F + F + \) is, for example, shown in Figure 7 (right), along with the position and orientation of the turtle after processing the command sequence.

The turtle can therefore graphically interpret any word generated by a Lindenmayer system over the alphabet \([F, +, -, R]\).

**Recursively drawing Lindenmayer systems.** Let \( P^i : \Sigma^* \rightarrow \Sigma^* \) be the function that maps any word \( w \in \Sigma^* \) to the word \( P^i (w) \) obtained by the i-fold substitution of all symbols in \( w \) according to their productions. With this notation we have \( w_0 = P^0 (s) \).

For \( \sigma \in \Sigma \), let \( w^\sigma_i \) denote the word \( P^i (\sigma) \). Suppose that \( P (\sigma) = \sigma_1 \cdots \sigma_n \). Then we get

\[
w^\sigma_i = P^{i-1} (P (\sigma)) = P^{i-1} (\sigma_1 \cdots \sigma_n) = P^{i-1} (\sigma_1) \cdots P^{i-1} (\sigma_n) = w_{i-1}^{\sigma_1} \cdots w_{i-1}^{\sigma_n}.
\]

This means that the drawing of \( w^\sigma_i \) is obtained by simply concatenating the drawings for \( w_{i-1}^{\sigma_1}, \cdots, w_{i-1}^{\sigma_n} \), and this is where the recursion comes in. To get \( w_i \), we argue in the same way that we can simply concatenate the drawings of all \( w^\sigma_i \), for \( \sigma \) running through the symbols of \( s \).

Program 21 shows how this works for our running example with productions \( F \rightarrow F + F, + \rightarrow +, - \rightarrow - \) and initial word \( F \). Since \( P^i (+) = +, P^i (-) = - \) for all \( i \), we do not need to substitute \(+\) and \(-\) and get

\[
w_1 = w^F_1 = w_0^F + w_0^F +.
\]

(3.1)

The program assumes the existence of a library turtle with predefined turtle commands forward, left (counterclockwise rotation with some angle) and right (clockwise rotations with some angle) in namespace ifm.

In the documentation of the program, we have omitted the "trivial" productions \( + \rightarrow +, - \rightarrow - \), and in specifying a Lindenmayer system, we can do so as well: we will usually only list productions for symbols that are not mapped to themselves.
As \( n \) gets larger, the picture does not seem to change much it rotates, and some more details develop, but apart from that the impression is the same. Assume you could draw the picture for \( n = \infty \). Then equation (3.1) would give

\[
W_\infty = W_\infty + W_\infty + .
\]

This is a self-similarity: the drawing of \( w_\infty \) consists of two rotated drawings of itself. We have a fractal!

Additional features. We can extend the definition of a Lindenmayer system to include a rotation angle \( \alpha \) that may be different from 90 degrees. This is shown in Program 22 that draws a snowflake for input \( n = 5 \).

```
// Program: snowflake.C
// Draw turtle graphics for the Lindenmayer system with
// production \( P \rightarrow \text{F} \cdot \text{F}^{*} \cdot \text{F}^{*} \cdot \text{F} \), initial word \( \text{F} \cdot \text{F}^{*} \cdot \text{F}^{*} \cdot \text{F} \) and
// rotation angle 60 degrees.
#include <iostream>
#include <IFM/turtle>
#include <iostream>

// POST: the word \( w_n \cdot \text{F} \) is drawn
void f (unsigned int i) {
  if (i == 0)
    ifm::forward(); // F
  else {
    f(i-1); // w_{i-1} \cdot \text{F}
    ifm::right(60); // -
    f(i-1); // w_{i-1} \cdot \text{F}
    ifm::left(120); // ++
  }
}

int main () {
  std::cout << "Number of iterations = ? ";
  unsigned int n;
  std::cin >> n;
  // draw \( w_n \cdot \text{F} \)
  f(n);
  // \( w_n \cdot \text{F} \)
  ifm::left(120);
  // ++
  f(n);
  // \( w_n \cdot \text{F} \)
  ifm::left(120);
  // ++
  f(n);
  // \( w_n \cdot \text{F} \)
  return 0;
}
```
```cpp
// Prog: dragon.C
// Draw turtle graphics for the Lindenmayer system with
// productions X -> X+XF+, Y -> -FX-Y, initial word X
// and rotation angle 90 degrees
#include <iostream>
#include <IFM/turtle>

void y(unsigned int i); // necessary: x and y call each other

void x(unsigned int i) {
    if (i > 0) {
        x(i-1); // w_{i-1}^X
        y(i-1); // w_{i-1}^Y
        y(i-1); // F
        x(i-1); // w_{i-1}^X
        y(i-1); // w_{i-1}^Y
    }
}

int main() {
    std::cout << "Number of iterations =? ";
    unsigned int n;
    std::cin >> n;
    draw w_n = w_n^X x(n);
    return 0;
}
```

Program 23: progs/dragon.C

Finally, one can add symbols with graphical interpretation. Commonly used symbols are \( f \) (jump one step forward, this doesn’t leave a trace), \( l \) (remember current position) and \( r \) (jump back to last remembered position). It is also typical to add new symbols with the same interpretation as \( f \), say.

### 3.2.7 Details

Lindenmayer systems. Lindenmayer systems are named after the Danish biologist Aristid Lindenmayer (1925–1986) who proposed them in 1968 to model the growth of plants. Lindenmayer systems (with generalizations to 3-dimensional space) have found many applications in computer graphics.

### 3.2.8 Goals

Dispositional. At this point, you should ...

1) understand the concept of recursion, and why it makes sense to define a function through itself

2) understand the semantics of recursive function calls and be aware that they do not always terminate

3) appreciate the power of recursion in drawing Lindenmayer systems.

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

(G1) find pre- and postconditions for given recursive functions;

(G2) prove or disprove termination and correctness of recursive function calls;

(G3) translate recursive mathematical function definitions into C++ function definitions;

(G4) rewrite a given recursive function in iterative form;

(G5) recognize inefficient recursive functions and improve their performance;

(G6) write recursive functions for given tasks.

### 3.2.9 Exercises

Exercise 56 Find pre- and postconditions for the following recursive functions, (G1)

a) bool f(int n) {
    if (n == 0) return false;
    return f(n-1);
}
b) void g (unsigned int n) 
{
    if (n == 0) { 
        std::cout << "*"; 
        return; 
    }
    g(n-1); 
    g(n-1); 
}

c) unsigned int h (unsigned int n, unsigned int b) 
{
    if (n == 1) return 0; 
    n = n / b; 
    return f(f(n));
}

Exercise 57 Prove or disprove for any of the following recursive functions that it terminates for all possible parameters.

a) unsigned int f (unsigned int n) 
{
    unsigned int t = 0; 
    while (t < n) return n; 
    return f(f(n));
}

b) unsigned int A (unsigned int m, unsigned int n) 
{
    // POST: return value is the Ackermann function value A(m,n) 
    if (m == 0) return n + 1; 
    return A(n - 1, A(m - 1, n)); 
}

c) unsigned int f (unsigned int n, unsigned int m) 
{
    unsigned int t = 0; 
    while (t < n) return n; 
    return 1 + (n + m) / 2; 
}

Exercise 58

a) Write a C++ function that computes binomial coefficients \( \binom{n}{k} \), \( n, k \in \mathbb{N} \). These may be defined in various equivalent ways. For example,

\[
\binom{n}{k} = \frac{n!}{k!(n-k)!}
\]

Exercise 59 Suppose you want to crack somebody's secret code, consisting of \( d \) digits between 1 and 9. You have somehow found out that exactly \( k \) of these digits are 1's.

a) Write a program that generates all possible codes. The program should contain a function that solves the problem for given parameters \( d \) and \( k \).

b) Adapt the program so that it also outputs the number of possible codes.

For example, if \( d = 2 \) and \( k = 1 \), the output may look like this:

```plaintext
12 13 14 15 16 17 18 19 21 31 41 51 61 71 81 91
```

There were 16 possible codes.

Exercise 60 Rewrite the following recursive function in iterative form and test with a program whether your iterative version is correct. What can you say about the runtimes of both variants for values of \( n \) up to 100, say?

```c
unsigned int f (unsigned int n) 
{
    if (n <= 2) return 1; 
    return f(n-1) + 2 * f(n-3); 
}
```

Exercise 61 Write a program dec2bin2.c that inputs a natural number \( n \) and outputs the binary digits of \( n \) (not in reverse order as in Exercise 52, but in normal order). For example, for \( n = 2 \) the output is 10 and for \( n = 11 \) the output is 1011.

Exercise 62 Write programs that produce turtle graphics drawings for the following Lindenmayer systems \( (\Sigma, P, s) \).
3.2 PROBLEM 153

a) \( \Sigma = \{ F, +, - \} \), \( s = F + F + F + F \) and \( P \) given by
\[
F \mapsto FF + F + F + F + F - F.
\]

b) \( \Sigma = \{ X, Y, +, - \} \), \( s = Y \), and \( P \) given by
\[
X \mapsto Y + X + Y \\
Y \mapsto X - Y - X.
\]

For the drawing, use rotation angle \( \alpha = 60 \) degrees and interpret both \( X \) and \( Y \) as "move one step forward".

c) Like b), but with the productions
\[
X \mapsto X + Y + Y - X - -XX - Y + \\
Y \mapsto -X + YY + +Y + X - -X - Y.
\]