Informatik I

An introduction to C++ with examples

Lecture Notes

Bernd Gärtner, Joachim Giesen, Michael Hoffmann
ETH Zürich

Winter semester, 2004/2005

Vorwort

Dieser erste Entwurf eines Vorlesungsskripts ist im Wintersemester 2004/2005 entstanden. Das Skript ergänzt zwei in diesem Semester an verschiedenen Departementen der ETH Zürich gelesene und vom Konzept her identische Vorlesungen. Das Skript wurde von den beiden Dozenten Bernd Gärtner und Joachim Giesen gemeinsam mit Michael Hoffmann geschrieben, der entscheidend an der Konzeption der Vorlesung beteiligt war. Wir haben uns entschieden, das Skript in englischer Sprache anzufertigen, weil uns dies natürlicher erscheint und auch eine bessere Wiederverwendbarkeit von Teilen des Skripts ermöglicht.

Das Konzept zur Vorlesung ist neu und befindet sich deshalb noch in der Testphase. Das Skript reflektiert bereits einige Ergebnisse dieser Testphase. So haben wir uns zum Beispiel entschieden, in Zukunft einige Anpassungen des Stoffs vorzunehmen. Material, auf das in Zukunft verzichtet wird (Sortieren, reguläre Ausdrücke) ist nicht durch das Skript abgedeckt. An einigen Stellen weicht auch die Reihenfolge der Themen von der Reihenfolge in der Vorlesung ab. Trotz des gemeinsamen Konzepts unterscheiden sich die beiden Vorlesungen an einigen Stellen, insbesondere in den Beispielen. Als Studierende werden Sie in diesem Skript deshalb sowohl Beispiele aus 'Ihrer' Vorlesung als auch welche aus der 'anderen' Vorlesung wiederfinden.

Ursprünglich war der Plan, dieses Skript erst zum nächsten Wintersemester und mit den vollen Erfahrungen aus diesem Semester zu schreiben. Wir haben allerdings früh realisiert, dass die Vorlesung ohne ein Skript nicht sehr gut dokumentiert ist, da es kein einzelnes Buch gibt, das den Stoff in ähnlichem Masse abdeckt. Wir haben uns deshalb entschieden, bereits semesterbegleitend am Skript zu arbeiten, so dass Sie zum Ende des Semesters eine erste Version in der Hand halten.

Wir möchten aber klar sagen, dass es sich hierbei um eine 'Beta-Version' handelt, die in vielen Bereichen noch Defizite und Inkonsistenzen aufweisen wird. Selbstverständlich haben wir versucht, darauf zu achten, dass der Stoff korrekt wiedergegeben ist, garantieren können wir das bei der knappen Zeit, die uns zur Verfügung stand aber nicht. Wir hoffen, dass das Skript für Ihre Prüfungsvorbereitung nützlich ist, sehen Sie es aber bitte nur als Ergänzung zum 'offiziellen' Vorlesungsstoff, der das Material auf der Webseite sowie die Tafelaufzeichnungen umfasst.

Inzwischen haben wir einige Fehler im Skript, die uns gemeldet worden sind, korrigiert (besonderer Dank gebührt Herrn Thomas Rast). Falls Sie weitere Unstimmigkeiten feststellen, melden Sie sich bitte bei uns, wir werden versuchen, sie schnellstmöglich zu beheben.

Zürich, 1. Juli 2005

Bernd Gärtner
Joachim Giesen
Michael Hoffmann

Contents

Chapter 1

Foundations

1.1 A first C++ Program

Let us begin with a small program which already demonstrates several aspects of C++ programming.

Program: add.C

```
// Program: add.C
// adds two input numbers of type int.

#include <iostream>

int main()
{
  int a;     // 1. addend
  int b;     // 2. addend

  // Input
  std::cout << "Input for the first addend : ";
  std::cin >> a;
  std::cout << "Input for the second addend: ";
  std::cin >> b;

  // Computation and output
  std::cout << "The sum of " << a
            << " and " << b << " is "
            << a+b << "." << std::endl;
  return 0;
}
```

If you compile this program on your computer and then run the executable file produced by the compiler you find the following line on the standard output, which typically is a window on your computer screen.

```
Input for first addend :
```

You can now enter an integer, e.g., 7, on the standard input which is usually the keyboard. After pressing ENTER the output on your screen should read as follows.

```
Input for first addend : 7
Input for second addend:
```

If you now enter a second integer, e.g., 5, and press ENTER afterwards, the output should read.

```
Input for first addend : 7
Input for second addend: 5
The sum of 7 and 5 is 12.
```

Thus the program asked for two numbers as input and outputs (as we would expect from a program called `add.C`) the sum of these two numbers.

Let us have a closer look at the different parts of the program and the role they play in providing the functionality we observed above.

1.1.1 Comments

The first two lines of the program

```
// Programm: add.C
// adds two input numbers of type int.
```

are comments. They do not provide any functionality. That is, the program would function exactly the same without these two lines. Nevertheless, comments are very important. Keep in mind that on the one hand we write programs for the compiler to transform them into executable programs that serve some functionality, but on the other hand we also write the programs for other people to read, e.g., in order to modify, correct or extend the program. Without comments it becomes in general a tedious if not to say impossible task for other people (or even oneself after a couple of weeks) to read and understand the program. Comments start with two slashes `//` and continue until the end of the line.

1.1.2 The `include` directive

More important to the functionality of the program is the line

```
#include <iostream>
```

In this line we state that we want to use the in- and output facilities as they are implemented in the standard library. In C++, many important facilities are part of the standard library rather than of the core language. Whenever we want to use such a facility we use the `include` directive to state that we want to use a corresponding library. The name of the library is always enclosed in the angle brackets (< and >). The `iostream` library is part of the standard library and supports sequential in- and output. From the `iostream` library we use the the input stream `std::cin` and the operator `>>` to read values for the variables `a` and `b` from the standard input, and we use the output stream `std::cout` and the operator << to write to the standard output. E.g., with the line

```
std::cout << "Input for first addend : ";
```

we send the string (finite sequence of characters enclosed in double quotes `"`) to standard output. To avoid confusion, notice that the letters `std` before the double colon `::` (scope operator) do not indicate that we refer to the standard output, but that the output stream `cout` is defined in the name space `std`. For now that means that `cout` is defined in the standard library. The same of course holds for `std::cin`. The last entity that we use from the `iostream` library is the stream manipulator `std::endl` that causes (besides other things) a line break in the output.

1.1.3 The `main` function

The next interesting and also very important line of our program is

```
int main()
```

In this line we declare (and define, i.e., give its implementation) the main function of our program. Every C++ program must contain a main function. It gets called when the program is executed. The main function must return an integer as its result. This is indicated by the three letters `int` before the letters `main`. The integer can be used by the calling instance, i.e., the operating system of our computer, to decide if the program has run successfully or not. The convention is that a zero value indicates success whereas any other value indicates failure. That is why we have as the second last line of the program the line

```
return 0;
```

If the program control reaches this line then everything was processed as planned and we can report success to the calling instance, i.e., we can return the integer value zero. It is also possible for the calling instance to pass parameters to the main function. These parameters are listed in between the brackets `(` and `)` after the letters `main`. Here, this parameter list is empty which means that our main function does not take any parameters from the calling instance.

1.1.4 Blocks

The body, i.e., the whole program logic that encodes the functionality, of the main function is enclosed in curly brackets. Everything enclosed in curly brackets is considered in C++ as one unit called a block, i.e., an opening curly bracket { and the next subsequent (when reading the program top down) closing curly bracket } define a block of our program.

1.1.5 Types and variables

The remaining lines that we have not described so far deal with the integer variables `a` and `b`. A variable is a part of the computer's memory that has a name and a value. In the lines

```
int a;    // 1. addend
int b;    // 2. addend
```

we define two variables of type `int`. The first variable has the name `a` and the second has the name `b`. The definition of a variable is also a declaration. Valid names in C++ are all finite sequences of symbols that can be composed of letters and digits and the underscore _ symbol. A name must always begin with a letter.[1] Upper and lower case letters are distinguished. Certain names reserved by C++ are not allowed as names, e.g., "main" or "int".

Inside the block where the name of the variable is defined we can always refer to the associated memory (that gets reserved in the definition) by the name of the variable. Outside this block the name is not visible and we can not use the name to refer to the associated memory. From the type information the compiler deduces how much memory to allocate for the variable and how to interpret the sequence of bits (zeros and ones) that is stored in the allocated memory. Variables of type `int` are intended to store integer values. We will later see how much memory gets allocated for a variable of type `int` and how an integer stored in allocated memory is encoded by a sequence of bits. To summarize, a variable has a name, type, associated memory and value. The value of the variable can be derived from memories content and the type information.

Usually types come with associated operations defined on objects of this type. One operation defined on `int` variables is the binary operator + that we use in the lines

```
std::cout << "The sum of " << a
          << " and " << b << " is "
          << a+b << "." << std::endl;
```

to add the integers stored associated with variables named `a` and `b`, or shorter to add the variables `a` and `b`.

[1] or an underscore

1.1.6 Scope and name lookup

Every block (piece of code enclosed by curly brackets) defines its own (local) scope. Names (like variable names) declared in this scope are only visible in this scope and all its sub-scopes. That is, they are local to this scope and consequently often referred to as local variables. Have a look at the following code fragment (this is not a complete program, which can be compiled).

```
// global scope
{ // local scope
  int a; // only a is visible in this scope
  { // another local scope
    int b; // a and b are visible in this scope
  }
}
```

To find the declaration associated to a name the compiler first looks in the scope where the name appears. In its scope, a name is visible from its point of declaration in a top-down manner, but not before. If it does not find a declaration in this scope it continues its search in the next enclosing super-scope. This continues until a scope is found that contains an appropriate declaration or the topmost scope, the global scope, is reached. This process is called name lookup. The following code fragment exemplifies the name lookup.

```
{
  int a = 7;
  int b = a;
  {
     std::cout << a << std::endl; // output is 7
     int a = 5;
     std::cout << a << std::endl; // output is 5
     std::cout << b << std::endl; // output is 7
  }
  std::cout << a << std::endl; // output is 7
}
```

One says that the second definition of the `int` variable `a` in the sub-scope hides the first definition of the variable `a`.

In this code fragment we have also used *initialization*. In the lines

```
int a = 7;
int b = a;
```

we are not only defining the `int` variables `a` and `b`, but we are also initializing them (`a` with the value seven and `b` with the value of `a`) using the binary assignment operator =.

1.1.7 Statements

It remains to discuss one small but important detail that we have not described so far. Actually this detail is only a letter, namely, the letter `;`. This letter is always the last letter of a statement, e.g., the declaration of a variable (declaration statement). A lone `;` is also a statement, namely, the null statement, which has no effect. We also consider a block (of statements) as a statement, i.e., a sequence of statements enclosed in curly brackets. A block is also referred to as a compound statement.

1.2 Control flow

We have not said so but the implicit assumption when discussing our first program `add.C` was that the lines of the program are processed one after the other top-down. This is the linear (top-down) control flow. To write more complex programs we have to deviate from the top-down control flow at times. These deviations are controlled by so called (flow) control statements.

Motivated by the following episode our second program should test if a given positive integer is a prime number or not, i.e., if it has more then two divisors.

In 1644 the monk Mersenne conjectured that all numbers of the form $2^n - 1$ are prime numbers for

$$n = 2, 3, 5, 7, 13, 17, 19, 31, 67, 127, 257,$$

and are not prime numbers for the remaining $1 < n < 257$. It took more than 200 years to show that Mersenne was wrong. Frank Cole showed in 1903 that $2^{67} - 1$ is not a prime number. We want to confirm Cole's result by devising the following program that reads an integer n from standard input and tests for all integers between 2 and $n - 1$ if they divide n. If one of these numbers divides n then n cannot be a prime number. That is, we need a means to enumerate all potential divisors, although we do not know the number to be tested at compile time, i.e., at the time when we compile the program. Even if we would know the number to be tested at compile time, a program with linear control flow needs at least one line of code for every potential divisor to be tested. Such a program to confirm Cole's result would have at least 2^{67} lines of code most of which are highly repetitive. Thus it is wise to deviate at times from the linear control flow.

Our program for primality testing reads as follows.

Program: prime.C

```
// Program: prime.C
// Test if a given integer is a prime number.

#include <iostream>

int main ()
{
  // Input
```

```
  int n; // integer to be tested
  std::cout << "Integer larger than one? ";
  std::cin >> n;

  // Test
  int d = 1; // potential divisor
  do {
    ++d;
  } while (n%d != 0);

  // Output
  if (d == n)
    std::cout << n << " is a prime number." << std::endl;
  else
    std::cout << n << " is not a prime number and " << d
              << " is the smallest non-trivial divisor of "
      << n << "." << std::endl;
  return 0;
}
```

1.2.1 The `do while` loop

The first control statement used in `prime.C` is the `do while` loop.

```
do {
  ++d;
} while (n%d != 0);
```

The statements (here we have only the single statement `++d`) in the block enclosed by `do` and `while` are executed as long as the condition in the parentheses after the letters `while`, i.e., in our case `n%d != 0` is fulfilled. That is, as long as the expression `n%d` is different from zero the `int` variable `d` gets increased by one. More specifically, the unary operator `++` increases the value of an `int` variable by one. The binary operator `!=` tests if the value of its left and right argument are not the same. Finally, the binary operator `%` is the modulo operator on the type `int`, i.e., it gives the rest of the integer division of its left argument (in our case `n`) by its right argument (here `d`). Hence these three lines of code say, increase `d` in steps of one as long as the division of `n` by `d` has a non-zero remainder, i.e., as long as `d` does not divide `n`.

The general form of a `do while` loop is as follows,

```
do {
  statement_1
      ...
  statement_n
} while (condition);
```

1.2.2 The `if else` statement

The second control statement used in `prime.C` is the `if else` statement,

```
if (d == n)
  std::cout << n << " is a prime number." << std::endl;
else
  std::cout << n << " is not a prime number and " << d
            << " is the smallest prime divisor of " << n
            << "." << std::endl;
```

The purpose of the `if else` structure is the conditional execution of statements. In our example that means under the condition that the values of the variables `d` and `n` coincide, the statement

```
std::cout << n << " is prime number." << std::endl;
```

is executed. Otherwise, i.e., in the case that `d` and `n` have different values, the statement

```
std::cout << n << " is not a prime number and " << d
          << " is the smallest non-trivial divisor of " << n
          << "." << std::endl;
```

will be executed. Notice that the binary operator == tests if the value of its left and right argument are the same. In this sense it is dual to the operator !=. Be careful not to confuse the assignment operator = with the operator ==.

The general form of an `if else` statement reads as follows,

```
if (condition)
  statement_1
else
  statement_2
```

Remember that statements can also be blocks of statements, i.e., a sequence of statements enclosed in curly brackets.

1.2.3 The `if` statement

Sometimes the `else` part of the `if else` statement is not necessary. In such a case we can use just the `if` part and omit the `else` part. That is, the general form of the `if` statement reads (where the statement again can be a block),

```
if (condition)
  statement
```

In this form the `if` statement is equivalent to the following `if else` statement,

```
if (condition)
  statement
else
  ;
```

Let us return to the program `prime.C`. Can we say that this program is correct, i.e., can we trust its output? We can try to reason about the program. Let us collect what we know;

(1) Before the body of the `do while` loop of `prime.C` was entered for the first time we have `d == 1` and `n >= 2` (assuming the user of the program followed the instruction to enter an integer larger than one).

(2) Always after `d` was incremented we have `d > 1` and and `d <= n` and `n == (n/d)*d + n%d`. The second inequality must be valid since `n` is a divisor of itself. Thus if `d` has the same value as `n` then the condition of the `do while` is no longer satisfied and `d` is not incremented any further. The last equality is a property of the modulo operator `%`, the binary multiplication operator `*` and the binary integer division operator `/` on the type `int` that always holds for positive `n` and `d`. The value of the variable `d` has to be positive, because it is positive before we enter the `do while` loop and gets only incremented within the `do while` loop. Assuming the user followed the instructions also `n` has to be positive.

(3) After the `do while` loop was left we have `d > 1`, since `d` got incremented at least once, and `d <= n` (as in (2)) and `n%d == 0`, because we left the `do while` loop.

From this we can deduce that it is always valid that `d >= 1` and `d <= n` and `n == (n/d)*d + n%d`. Thus these statements are invariant under processing the `do while` loop. That is, why one refers to such statements as loop invariants. How does this help now to make a statement about the correctness of `prime.C`? After the `do while` was left we know that `d` has to be a divisor of `n`, because it holds that `n%d == 0`. Since it also holds `d > 1` and `d <= n` there are only two possibilities: either `d == n` and `n` is a prime number or `d` is a non-trivial divisor of `n` and `n` cannot be a prime number. These two cases are handled accordingly by the `if else` statement.

1.2.4 The `while` loop

The body of a `do while` loop is executed at least once before the condition of the loop gets evaluated. In the alternative `while` loop this is not the case. There the condition gets evaluated before the body of the loop is executed which can have the result that the body never gets executed. In an alternative implementation of the program `prime.C` we replace the actual primality test by the following code fragment that uses a `while` loop.

```
// Test
int d = 2; // potential divisor
while (n%d != 0)
  ++d;
```

Notice that this code fragment is equivalent to the original one with respect to functional correctness, but it is not equivalent in terms of processed statements. Actually, we should have used this form of the loop right from the beginning.

The general form of a `while` loop reads as follows.

```
while (condition)
  statement
```

where the statement can also be a block.

1.2.5 The `for` loop

An alternative to the `while` loop as we have used it above is a `for` loop. The `for` loop is typically used in case that one iterates over some domain and applies the same operation to all elements in this domain. That is exactly what we do in our primality testing program. Thus we can replace the test in the program `prime.C` by the following code fragment, which involves a `for` loop,

```
// Test
int d; // potential divisor
for (d = 2; n%d != 0; ++d);
```

The general form of a `for` loop reads as follows.

```
for (init-statement condition; expression)
  statement
```

The init-statement must be an expression statement. Remember that an expression statement always ends with a semicolon. The init-statement is typically used to declare and initialize the loop control variable(s). In our example the single loop control variable is `d`. We only initialize `d` in the init-statement, because if we would also define it there it would not be visible outside the scope defined by the `for` loop, but we need `d` also outside the loop. The expression, in our example `++d`, is an expression statement without the semicolon at the end.

A `while` loop equivalent to the general `for` loop reads as follows,

```
{
  init-statement
  while (condition) {
    statement
    expression;
  }
}
```

The equivalent `while` loop makes apparent why we could not define the variable `d` in the init-statement of the `for` loop. The init-statement and the `while` loop are contained in their own scope, i.e., they are enclosed by curly brackets. The variable name `d` is not visible outside this scope. Since we want to use it outside we also have to define it outside.

1.3 Integers

We have already seen several operations defined on the type `int`, which can be used to represent integers. Here we want to use some of these operators to convert a temperature given in degrees Celsius into degrees Fahrenheit. The conversion is defined by the following formula,

$$\text{Fahrenheit} = \frac{9 \cdot \text{Celsius}}{5} + 32.$$

We use this formula in the following program.

Program: fahrenheit.C

```
// Program: fahrenheit.C
// converts temperatures from Celsius to Fahrenheit.

#include <iostream>

int main()
{
  // Input
  std::cout << "Temperature in degrees Celsius (integer) ? ";
  int celsius;
  std::cin >> celsius;

  // Computation and output
  std::cout << celsius << " degrees Celsius are approximately"
            << 9*celsius/5 + 32 << " degrees Fahrenheit."
            << std::endl;
  return 0;
}
```

1.3.1 Associativities and precedences

Applying the rules of arithmetic it looks like we could replace the line

```
std::cout << celsius << " degrees Celsius are approximately"
            << 9*celsius/5 + 32 << " degrees Fahrenheit."
            << std::endl;
```

in the program `fahrenheit.C` by the following line.

```
std::cout << celsius << " degrees Celsius are approximately"
            << 9/5*celsius + 32 << " degrees Fahrenheit."
            << std::endl;
```

If we try out out both versions after compilation, we find that in the first version we get on input 15 degrees Celsius the output

```
15 degrees Celsius are approximately 59 degrees Fahrenheit.
```

whereas the modified version outputs

```
15 degrees Celsius are approximately 47 degrees Fahrenheit.
```

That is, the first version is much more accurate (actually, it is accurate). The reason for this behavior is twofold. Firstly, the binary division operator `/` on the type `int` implements the integer division, i.e., the result is always an integer. The remainder of the division can be obtained with the binary modulo operator `%`. It always holds `a == a/b + a%b` for `int` variables `a` and `b` that have non-negative values. Secondly, the expressions `9*celsius/5 + 32` and `9/5*celsius + 32` get evaluated from left to right, since the C++ operators `/` and `*` are left associative. Plugging in the value 15 for the variable `celsius` we get the following evaluation sequence for the two expressions

$$\begin{array}{rcl} 9 * 15 / 5 + 32 & \longrightarrow & 135 / 5 + 32 \\ & \longrightarrow & 27 + 32 \\ & \longrightarrow & 59 \end{array}$$

and in the modified expression, respectively,

$$\begin{array}{rcl} 9 / 5 * 15 + 32 & \longrightarrow & 1 * 15 + 32 \\ & \longrightarrow & 15 + 32 \\ & \longrightarrow & 47. \end{array}$$

In the second evaluation sequence one sees the effect of the integer division: $9/5$ evaluates to 1. During the evaluation we have assumed (as we are used to when writing down arithmetic expressions) that the operators `*` and `/` bind stronger than the operator `+`. This is true indeed. The operators `*` and `/` have a higher precedence than the operator `+`, i.e., they bind stronger.

Notice that associativities and precedences do not completely specify the evaluation order of the operators in an arithmetic expression. For example the following two evaluation sequences are both compatible with the precedences and associativities,

$$\begin{aligned} 9 * 15 + 5 * 32 &\longrightarrow 135 + 5 * 32 \\ &\longrightarrow 135 + 160 \\ &\longrightarrow 295 \end{aligned}$$

and

$$\begin{aligned} 9 * 15 + 5 * 32 &\longrightarrow 9 * 15 + 160 \\ &\longrightarrow 135 + 160 \\ &\longrightarrow 295. \end{aligned}$$

Precedences can be overruled using parentheses as in the following expression,

$$\begin{aligned} 9 * (15 + 5) * 32 &\longrightarrow 9 * (20) * 32 \\ &\longrightarrow 9 * 20 * 32 \\ &\longrightarrow 180 * 32 \\ &\longrightarrow 5760. \end{aligned}$$

In the following table we list all arithmetic operators on the the type `int` together with their associativities and precedences.

meaning	operator	arity	precedence	associativity
sign	+	unary	15	right
sign	−	unary	15	right
increment	++	unary	15	right
decrement	−−	unary	15	right
multiplication	*	binary	13	left
division (integer)	/	binary	13	left
modulo	%	binary	13	left
addition	+	binary	12	left
subtraction	−	binary	12	left

The increment and decrement operators ++ and −− can be used in pre- and postfix notation. The difference is illustrated in the following lines of code.

```
int a = 7;
std::cout << ++a << std::endl; // outputs 8
std::cout << a++ << std::endl; // outputs also 8
std::cout << a << std::endl;   // outputs 9
```

The assignment operator = has precedence 2 and is right associative. The binary operators +, −, * and / can be combined with the assignment operator =. They read then as +=, −=, *= and /=. The meaning of the combined operators is that their right operand gets applied to their left operand, e.g., the following three statements are functionally equivalent,

`i = i+1;` and `++i;` and `i += 1;`.

1.3.2 Value range

It would be unreasonable to assume that an `int` variable can have an arbitrarily large or small value. This value somehow has to be stored in the computers memory, which is finite. In fact the value of the smallest and largest value an integer can assume depends on the computer, but C++ allows to access these values using the library `limits` and the following expressions

```
std::numeric_limits<int>::min()
```

and

```
std::numeric_limits<int>::max()
```

Here `max()` and `min()` are functions in the scope `std::numeric_limits<int>`. Only that much is necessary to know at this point: `min()` and `max()` are not functions in a nested namespace, but in a so called class scope of the class `std::numeric_limits<int>`. A program to output the values of the expressions `std::numeric_limits<int>::min()` and `std::numeric_limits<int>::min()` reads as follows.

Program: limits.C

```
// Program: limits.C
// ouputs the minimum and maximum value
// a variable of type int can take.

#include <iostream>
#include <limits>

int main()
{
  std::cout << "Minimum value for an int variable is : "
            << std::numeric_limits<unsigned int>::min()
            << std::endl
    << "Maximum value for an int variable is : "
            << std::numeric_limits<unsigned int>::max()
    << std::endl;
```

```
  return 0;
}
```

When we run the program `limits.C` on our machine (Pentium III processor) we get the following result.

```
Minimum value for an int variable is : -2147483648
Maximum value for an int variable is : 2147483647
```

We have $2147483647 = 2^{31} - 1$. That is, in its current form our program `prime.C` is not sufficient to verify Cole's result on Mersenne's conjecture, i.e., $2^{67} - 1$ is not a prime number.

1.3.3 The type `unsigned int`

A variable of type `int` can take also negative values. This is not really needed in our application `prime.C`. Using a type that allows only non-negative values might allows to extend the value range without using more of the computer's memory. C++ provides such a type, it is called `unsigned int`. We have all the arithmetic operators with the same precedences and associativities on the type `unsigned int` as on the type `int` besides the the unary sign operators. Obviously, the binary subtraction operator `-` cannot be well defined on the type `unsigned int`, i.e., the result cannot always be represented as an unsigned integer. Something similar already holds for the type `int` since subtraction can under- and addition can overflow the value range of the type `int`).

Plugging in the parameter `unsigned int` instead of `int` in the program `limits.C` and adjusting the output appropriately gives on our machine the following value range,

```
Minimum value for an unsigned int variable is : 0
Maximum value for an unsigned int variable is : 4294967295
```

It holds that $4294967295 = 2^{32} - 1$ which is still not large enough to verify Cole's result.

1.3.4 Literal integer constants

Literal integer constants of type `int` are for example `127` or `0`. Literal integer constants of type `unsigned int` are for example `127u` or `0u`. We will later see how the literal integer constants can be defined more formally.

1.3.5 Type conversion

It is allowed to have arithmetic expressions in a C++ program that involve both variables/constants of type `int` and of type `unsigned int`, e.g. $17+17u$. In such expressions the arguments of a binary operator that involves arguments of both types are converted to the more general type, which is the type `unsigned int`. That is, the expression $17 + 17u$ evaluates as follows,

$$\begin{aligned} \text{17 + 17u} &\longrightarrow \text{17u + 17u} \\ &\longrightarrow \text{39u.} \end{aligned}$$

Notice that the conversion of a negative integer value is in general not well defined, but the C++ standard prescribes that the conversion gives the least `unsigned int` value in the same congruence class modulo 2^{32} as the source `int` value. We will see the motivation for this prescription soon.

1.3.6 Binary representation of integers

The binary representation of a non-negative integer n are the coefficients in the following expansion,

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

where the coefficients b_i are in $\{0, 1\}$. Assume that on our computer we can linearly store 32 of such coefficients for the value of a variable of type `unsigned int` (a not unreasonable assumption on most of todays computers) then the value is stored as a sequence of 32 Bits. i.e., zeroes and ones, with the convention that the $(32 - i)$-th Bit is the coefficient of 2^i - the Bits are numbered from zero to 31. The largest value that we can store in this way is

$$\sum_{i=0}^{31} 2^i = 2^{32} - 1 = 4294967295.$$

A way to store negative integer values using the same 32 Bits is as follows: Use the 32nd Bit for the sign of the value. If the value is non-negative store zero in the 32nd Bit and the value in the first 31 Bits. Thus the largest positive value that can be stored is

$$\sum_{i=0}^{30} 2^i = 2^{31} - 1 = 2147483647.$$

To store negative values, take the Bit sequence that would correspond to the positive value, i.e., the value after multiplication with -1. Now invert all Bits in this sequence, i.e., a zero is turned into a one and vice versa. Finally, add binarily the value 1 to this sequence. For example -7 is represented as follows: The Bit representation of 7 is

$$00000000000000000000000000000111.$$

Inverting this Bit sequence gives

$$11111111111111111111111111111000.$$

Binary addition of

$$00000000000000000000000000000001$$

finally gives the representation

$$11111111111111111111111111111001$$

for -7. This representation is called the two's complement representation. Observe that if we binarily add

$$00000000000000000000000000000111 \quad \text{and} \quad 11111111111111111111111111111001$$

we get

$$00000000000000000000000000000000,$$

the representation of 0, (and an overflowing Bit that we ignore). That is, what we expect as the sum of 7 and -7. Hence in this representation signed integers can be easily added. Also notice that smallest signed integer that we can represent with 32 Bits has the representation

$$10000000000000000000000000000000,$$

i.e., this value is -2147483648.

Using this representation we can now better understand what happens if a negative value n gets converted in an expression that involves `int` and `unsigned int` variables/constants. The Bit sequence is just interpreted as in the unsigned case. Thus the value after conversion is `std::numeric_limits<unsigned int>::max()`$+n+1$ (here the additions are meant in the mathematical sense and not as C++ operators). This value is the least `unsigned int` value congruent to n modulo 2^{32} (given that 32 bits are reserved for values of the types `int` and `unsigned int`). Thus the prescription of the conversion of an `int` value to an `unsigned int` value is such that in two's complement representation of integers the bit sequence stays the same when converting from `int` to `unsigned int`.

We conclude this section with a program to compute the inverse (the sequence of the Bits is inverted) binary representation of an unsigned integer.

Program: binary.C

```
// Program: binary.C
// computes the inverse binary representation
// of an unsigned integer

#include <iostream>
#include <limits>
```

```
int main()
{
  // Input
  std::cout << "Unsigned integer (0 <= input <= "
            << std::numeric_limits<unsigned int>::max()
            << ")" << std::endl;
  unsigned int n;
  std::cin >> n;

  // Computation and output
  std::cout << "The inverse binary representation of "
            << n << " is: ";
  do {
    std::cout << n%2; // last Bit
    n /= 2;             // remove the last Bit
  } while (n != 0);
  std::cout << "." << std::endl;
  return 0;
}
```

1.4 Floating point numbers

When converting degrees Celsius into degrees Fahrenheit the result in general is fractional even if the input is integral. Thus it would be convenient to have a type that allows to represent rational or even real numbers. Since our computer's memory is limited it would be unreasonable to assume that any real (or rational) number can be represented. In fact, these numbers can only be represented approximately. A first but crude approximation would be the rounding to the nearest integer (or more specifically the nearest value that can be represented by a variable of type `int` or `unsigned int`). The most often used approximation is by a floating point number. C++ provides two types of floating numbers to approximate real numbers, namely, the types `float` and `double`.

1.4.1 The floating point types `float` and `double`

The types `float` and `double` provide the same arithmetic operations as the type `int`, besides the modulo operator. The precedences and associativities are exactly the same.

The type `double` is more general than the type `float`. That is, in a mixed expression a variable/constant of type `float` gets converted into a `double` value. Since this conversion is guaranteed to be safe, i.e., no information can get lost as in the case of the conversion from `int` to `unsigned int`, this conversion is called a promotion.

The floating point point types are more general than the integral types, i.e., `int` and `unsigned int`. Thus in mixed expressions the integral types get converted in involved floating point types. Notice, a conversion always goes from the less general to the more general type. But it is not guaranteed that no information gets lost in a conversion.

Examples of literal floating point constants of type `double` are `0.0` and `.1` and `1.` and `3.1415` and `1.23e-7`. Here `1.23e-7` represents the value $1.23 \cdot 10^{-7}$. Literal floating point constants of type `float` are as literal `double` constants followed by the letter f, e.g., `0.0f`, `3.1415f`, `.1f`, `1.f`, `3.1415f` or `1.23e7f`.

Our first program that involves the type `double` is very simple. It asks the user to input three real numbers, i.e., literal `double` constants, a minuend, a subtrahend and their difference. The program then checks if the difference is correct. It reads as follows,

Program: subtract.C

```
// Program: subtract.C
// Checks the subtraction of to numbers

#include <iostream>

int main()
{
  // Input
  double minuend;
  std::cout << "Minuend    =? ";
  std::cin >> minuend;

  double subtrahend;
  std::cout << "Subtrahend =? ";
  std::cin >> subtrahend;

  double result;
  std::cout << minuend << " - " << subtrahend << " =? ";
  std::cin >> result;

  // Comparison and output
  if (result == minuend - subtrahend)
    std::cout << "Correct." << std::endl;
  else
    std::cout << "Wrong, by " << minuend - subtrahend - result
 << "!" << std::endl;
  return 0;
}
```

Here is what happens if we test the program `subtract.C`.

```
Minuend    =? 1.1
```

```
Subtrahend =? 1.0
1.1 - 1 =? 0.1
Wrong, by 8.32667e-17!
```

What has happened here? Obviously we have a precision problem. At a first glance it looks like that we should have enough memory to represent the values 1.1, 1.0 and 0.1. A somewhat deeper look reveals that this is not the case. The value x of a variables of type `double` is stored as floating point number, which has the following representation,

$$x = sm2^e,$$

where $s \in \{\pm 1\}$ is the sign of x, $e \in \{e_{\min}, \ldots, e_{\max}\} \subset \mathbb{Z}$ is an exponent and m is a mantissa, which is of the form

$$m = \sum_{i=1}^{l} b_i 2^{-i},$$

with $b_i \in \{0, 1\}$ and l the length of the mantissa. Notice that the range of floating point numbers that we can represent is determined by the three parameters $e_{\min}, e_{\max}$ and l.

There is only hope that we can store the values 1.1, 1.0 and 0.1 in our computer's memory if they have a finite floating point representation. In order to have a finite floating point representation they need to have a finite binary expansion, i.e., in an expansion of the form

$$x = \sum_{i=-\infty}^{\infty} b_i 2^i,$$

with $b_i \in \{0, 1\}$, we have $b_i = 1$ only for finitely many i's. Using the division algorithm known from primary school adapted to binary expansions one finds that the binary expansion of $1.1 = 11/10$ is as follows $b_i = 0$, for $i > 0$, $b_0 = 1, b_{-1} = b_{-2} = b_{-3} = 0$ and $b_{-4j} = b_{-4j-1} = 1$ and $b_{-4j-2} = b_{-4j-3} = 0$ for $j \geq 1$. That is, 1.1 does not have a finite binary expansion and thus does also not have a finite floating point representation. That explains to a certain extent the behavior of the program `subtract.C` on input 1.1, 1.0 and 0.1.

1.4.2 IEEE Standard 754

This IEEE standard defines representations of floating point numbers. Note that the C++ floating point types need not to follow this standard. There are single precision (32 bits) and double precision (64 bits) floating point numbers. In most cases it is not wrong to assume that the type `float` corresponds to the single precision representation and the type `double` corresponds to the double precision representation. The memory layout for single precision is one bit for the sign, eight bits for the exponent and 23 bits for the mantissa. The layout for double precision is one bit for the sign, 11 bits for the exponent and 52 bits for the mantissa.

SIGN BIT. Zero denotes a positive number and one denotes a negative number.

EXPONENT. Signed integers are stored. The representation is as discussed for the type `int`, besides that a bias is added to this representation. The bias for single precision is 127 and 1023 for double precision. The all-zero pattern in the exponent (in single precision this would correspond to a value of -127) and the all-one pattern in the exponent (in single precision this corresponds to a value of 128) are reserved to represent special floating point numbers. Thus the value ranges $\{e_{\min}, \ldots, e_{\max}\}$ of the exponent are $\{-126, \ldots, 127\}$ for single precision and $\{-1022, \ldots, 1023\}$ for double precision, respectively.

MANTISSA. The mantissa is stored in normalized form. That means that it is assumed that the leading bit in the mantissa is one, i.e., the bit that denotes the coefficient before the binary point. This allows some optimization. Since we always assume that the leading bit is one we do not have to store it explicitly. As a result, the mantissa has effectively 24 bits in the case of single precision and 53 bits in the case of double precision.

SPECIAL AND DENORMALIZED VALUES. Zero is a special value, which is represented with all zeroes for the exponent bits and all zeroes for the mantissa. Notice that 0 and -0 are distinct values (though they compare as equal). If all bits in the exponent are zero then the value is a denormalized number, which does not have an assumed leading one before the binary point. Thus zero is a special form of a denormalized number. The values plus and minus infinity are represented by an exponent whose bits are all one and a mantissa whose bits are all zero. The sign bit distinguishes between plus and minus infinity. Finally, an exponent whose bits are all one and a mantissa whose bits are not all zero denotes the value `NaN` (Not a Number).

1.4.3 Value ranges

We approximate real numbers in the following value ranges using either single or double precision floating point numbers.

	denormalized	normalized
single precision	$\pm 2^{-149}$ to $\pm(1-2^{-23})2^{-126}$	$\pm 2^{-126}$ to $\pm(2-2^{-23})2^{127}$
double precision	$\pm 2^{-1074}$ to $\pm(1-2^{-52})2^{-1022}$	$\pm 2^{-1022}$ to $\pm(2-2^{-52})2^{1023}$

Thus in single precision we are not able to represent (correspondingly for double precision),

(1) negative numbers smaller than $-(2-2^{-23})2^{127}$ (negative overflow),

(2) negative numbers larger than -2^{-149} (negative underflow),

(3) positive numbers larger than $(2-2^{-23})2^{127}$ (positive underflow) and

(4) positive numbers smaller than 2^{-149} (positive overflow).

RELATIVE ERROR. Every real number x with $|x|$ in the interval

$$2^{e_{\min}} \leq |x| \leq (2-2^{-l})2^{e_{\min}}$$

gets approximated by its nearest floating point number $\hat{x}$ with a relative error of

$$\frac{|x - \hat{x}|}{|x|} \leq \varepsilon,$$

where $\varepsilon = 2^{-l}$ (remember that l is the number of bits of the exponent) is the so called machine epsilon. The value of the machine epsilon can be obtained from the `limits` library (similar as in our program `limits.C`), calling the function

```
std::numeric_limits<float>::epsilon();
```

To get the machine epsilon for the type `double` one has to replace the template parameter `float` by the parameter `double`.

To demonstrate the effect of floating point approximations we want to show a program that computes the n-th harmonic number

$$H_n = \sum_{i=1}^{n} \frac{1}{i} = \sum_{i=1}^{n} \frac{1}{n+1-i}$$

using the single precision type `float` in the two ways as given by the two sums.

Program: harmonic.C

```
// Program: harmonic.C
// computes the n-th harmonic number in two ways.

#include <iostream>

int main()
{
  // Input
  std::cout << "Which harmonic number H_n (1 <= n <= "
            << std::numeric_limits<unsigned int>::max()
            << ") ? ";
  unsigned int n;
  std::cin >> n;

  // Forward sum
  float sum1 = 0.0f;
  for (unsigned int i = 1; i <= n; ++i)
    sum1 += 1.0f / i;

  // Backward sum
```

```
  float sum2 = 0.0f;
  for (unsigned int i = n; i >= 1; --i)
    sum2 += 1.0f / i;

  // Output
  std::cout << "Forward sum  = " << sum1 << "\n"
            << "Backward sum = " << sum2 << std::endl;
  return 0;
}
```

Before we discuss the results of this program let us have a look at the `1.0f / i` as used in the in `harmonic.C`. Here `1.0f` is a `float` literal constant and `i` is a variable of type `unsigned int`. Since `float` is the more general type than `unsigned int`, the value stored in `i` is converted to `float` and then the division operator on the type float is applied. The result of the division is again of type `float`. In the expression `1 / i` the `int` literal constant `1` would be converted to `1u` and then the division operator on the type `unsigned int` applied. The result would always be `0u` besides in the cases when `i` stores the values zero and one. Thus replacing the expression `1.0f / i` by the expression `1 / i` would change the program `harmonic.C` dramatically.

Now let us have a look at an execution of the program `harmonic.C`,

```
Which harmonic number H_n (1 <= n <= 4294967295) ? 10000000
Forward sum  = 15.4037
Backward sum = 16.686
```

The results differ significantly. The difference becomes even more apparent when we execute `harmonic.C` again with a larger input,

```
Which harmonic number H_n (1 <= n <= 4294967295) ? 100000000
Forward sum  = 15.4037
Backward sum = 18.8079
```

Notice that the forward sum did not change, which cannot be correct. Using the approximation

$$\frac{1}{2(n+1)} < H_n - \ln n - \gamma < \frac{1}{2n},$$

where $\gamma = 0.57721666\ldots$ is the Euler-Mascheroni constant, we get $H_n \approx 18.998$. That is, the backward sum gives us a reasonable approximation of the n-th harmonic number. The reason why the backward sum behaves better than the forward sum is that the very small numbers that are added in the end to the forward sum are no longer significant if the exponent is adapted to the exponent of the sum that has already been computed, i.e., when adding the value of `1.0f / i` to the value of `sum1` the exponent of the value of `1.0f / i` has to be increased in order to match the exponent of the value `sum1` and the mantissa of the value of `1.0f / i` has to be filled with leading zeroes. Soon there are more leading zeros than the number of bits in the mantissa, which results in adding zero to the sum. Thus the sum remains constant once it reached 15.4037, although $\lim_{n\to\infty} H_n = \infty$.

1.5 Boolean expressions

We have already seen two Boolean expressions in the program `prime.C`, both in conjunction with control flow structures, i.e., the expression `n%d != 0`, where we have checked if `d` is a divisor of `n`, and the expression `d == n`. In the following table we show some Boolean operators that are defined on all arithmetic types, i.e., the integer and the floating point types,

meaning	operator	arity	precedence	associativity
$<$ relation	<	binary	10	left
$\leq$ relation	<=	binary	10	left
$>$ relation	>	binary	10	left
$\geq$ relation	>=	binary	10	left
equality test	==	binary	9	left
inequality test	!=	binary	9	left

1.5.1 The type `bool`

The operators that we have shown in the table above can be used to form atomic Boolean expressions that are either true or false. C++ provides a type `bool` to represent the two Boolean values true and false. The two literal `bool` constants are `true` and `false`. The integral types `int` and `unsigned int` can be converted into the type `bool`, where the value zero is interpreted as `false` and all other values are interpreted as `true`. The type `bool` can even be promoted to the integral types, i.e., the C++ standard guarantees that no information gets lost. In the following table we list the operators defined on the type `bool`; they correspond to the logical and, or and negation operation.

meaning	operator	arity	precedence	associativity
negation	!	unary	15	right
logical and	&&	binary	5	left
logical or	||	binary	4	left

Let us see now an example how a Boolean expression gets evaluated,

```
5 * 7 <= 32 && 11 == 2 * 5 + 1 || 6 > 5
   ⟶  35 <= 32 && 11 == 2 * 5 + 1 || 6 > 5
   ⟶  false && 11 == 2 * 5 + 1 || 6 > 5
   ⟶  false || 6 > 5
   ⟶  false || true
   ⟶  true.
```

1.5.2 Short circuit evaluation

The evaluation above is surprising if we consider only precedences and priorities. The expression `11 == 2 * 5 + 1` never gets evaluated. This is due to the fact that in addition to precedences and priorities another rule applies to the evaluation of Boolean expressions, namely the so called short circuit evaluation. The short circuit evaluation rule says the following: for a binary Boolean operator the left operand always gets evaluated first. If the value of the operator is determined already after evaluating its left operand then the right operand does not get evaluated. That is, if the left hand side argument of the operator `&&` evaluates to `false`, as it is the case in our example, then the expression on the right hand side does not get evaluated and the value of the whole expression is `false`. Correspondingly, if the left hand side argument of the operator `||` evaluates to `true` then the expression on the right hand side does not get evaluated and the value of the whole expression is `true`.

1.6 Strings

Besides numbers, text is among the most often automatically processed quantities. Thus it is not surprising that C++ provides types to handle characters and text.

1.6.1 The type `char`

A variable of type `char` stores characters. Literal character constants are for example `'A'`, `'a'`, `'2'` or `'\n'`. Here `'\n'` is a special character (control sequence) that causes a line break. It might be surprising that `char` is also an arithmetic type. In arithmetic expressions involving `char` values these values get promoted to the type `int`. For example on our system both expressions `'2' - '0'` and `'2' - 48` have the `int` value two.

Encoding of characters. Unlike for numbers there is no natural encoding of characters. The encoding has to be defined in a table. One of the most popular table are the ASCII (American Standard Code for Information Interchange,) which uses seven bits for the encoding, or extended ASCII code, which uses eight bits and represents more characters.

1.6.2 The type `std::string`

A string is a sequence of characters. C++ does not really provide a built in type for strings, but we can find such a type in the standard library. To use this type we have to include the string library using the directive `#include <string>`. String constants can be defined by sequences of characters enclosed in quotation marks, e.g. `"This is a string."`.[2] In

[2] Actually, the sequence `"This is a string."` is not of the type `std::string`, but it will automatically be converted to this type when used to initialize a variable of type `std::string`. This is a subtlety going back to the `C` programming language.

order to use the character `'"'` within a string we have to escape it using the escape sequence `'\"'`. Thus the statement

```
std::cout << "\"This is a string.\"" << std::cout << std::endl;
```

writes `"This is a string."` to the standard output. The type `std::string` comes with the binary operator +, which has the same associativity and precedence as in arithmetic expression. The + operator concatenates two strings, e.g., the following sequence of statements

```
std::string s = "This is ";
std::cout << s + "a string." << std::endl;
```

results in printing `This is a string.` on the standard output device. The type `std::string` also provides (member) functions that can be applied to variables of type `std::string`. For example, the function `length()` returns the length of a string. The sequence of statements

```
std::string s = "This is a string.";
std::cout << "The string \"" << s << "\" has length "
          << s.length() << "." << std::endl;
```

results in printing `The string "This is a string." has length 17.` on the standard output device. We call a member function via an object of the string class by using the binary dot operator `.` whose left argument is a variable name and whose right argument is the name of the member function, e.g., `s.length()`.

The type `std::string` provides even more, namely, its own types. The types

`std::string::iterator` and `std::string::const_iterator`

can be used to define iterator variables, which can be used to iterate through a string and to access the characters in a string.

1.6.3 String iterators

String iterators can be seen as 'pointers' to the the characters in a string. Thus the string iterator types are equipped with unary dereference operator `*`, which when applied to an iterator returns the corresponding character. Furthermore iterators can be incremented and decremented, i.e., preceding to the next character in the string and to the previous character, respectively. Therefore, the string iterator types come with the unary increment operator ++ and the unary decrement operator `--`. The latter operators both come in pre- and postfix form. Iterators can also be compared using the binary comparison operators == and `!=`.

Let us have look at an example. On Apple operating systems the character `'\r'` causes a line break whereas on Unix operating systems this is caused by the character `'\n'` (on Microsoft's operating systems the two characters `\n\r` are needed to cause this effect). The program `mac2unix.C` converts text files from Mac (short for Apple Macintosh) format into the Unix format.

Program: mac2unix.C

```
// Programm: max2unix.C
// converts Mac text files into Unix text files.

#include <iostream>
#include <string>

int main()
{
  // Input
  std::string f;
  std::getline(std::cin, f);

  // Conversion
  std::string::iterator b = f.begin();
  std::string::iterator e = f.end();
  for (; b != e; ++b)
    if (*b == '\r') *b = '\n';

  // Output
  std::cout << f;
  return 0;
}
```

Before we discuss the iterators used in the program `mac2unix.C` let us have a look at the line

```
std::getline(std::cin, f);
```

In this line we use a function named `getline`, which is defined in the namespace `std`, to read from the input stream `std::cin`, i.e., from the standard input (in general the keyboard) into the string `f`. The string `f` is then modified in the following part of the program.

Every `string` object provides two iterators that can be accessed through the member functions `begin()` and `end()`, respectively. That is, in the lines

```
std::string::iterator b = f.begin();
std::string::iterator e = f.end();
```

we define two variables of type `std::string::iterator`, one named `b` and the named `e`, and initialize them with the value of `f.begin()` and `f.end()`, respectively. The two iterators `f.begin()` and `f.end()` define an iterator range through which we can iterate using the increment and decrement operators. The convention is that the iterator `f.begin()` points to the first character in the string `f`, whereas the the `f.end()` points behind the last character in the string, i.e., it cannot be dereferenced meaningfully. In this sense the iterator range can be seen as an half open interval [`f.begin(), f.end()`). In the `for` loop

```
for (; b != e; ++b)
  if (*b == '\r') *b = '\n';
```

we use the ++ operator in prefix form to iterate through the iterator range [`f.begin()`, `f.end()`), i.e., through the string `f`. In the body of the loop we check if the character the iterator points to is `'\r'`. To this end we use the dereference operator `*`, that we apply to the iterator `b`. If the iterator `b` points to the character `'\r'` then we replace it with `'\n'`.

Finally we output the modified string `f` by putting it on the output stream `std::cout`.

Notice that it is important in the program `mac2unix.C` that we can modify the value of the character pointed to by the the iterator `b`. Sometimes we only want to read this value but not modify it. In this case a `std::string::const_iterator` should be used. The use of such an iterator makes it possible for the compiler to ensure the the value the iterator points to does not get modified. In this sense the use of `std::string::const_iterator` is a safety mechanism.

1.6.4 Type aliasing using `typedef`

Sometimes it is cumbersome to write type names like

`std::string::const_iterator`.

C++ allows to define an alias for such a type name using the keyword `typedef`. For example we can use it to write the following variant of the program `mac2unix.C`,

Program: mac2unix-2.C

```
// Programm: max2unix.C
// converts Mac text files into Unix text files.

#include <iostream>
#include <string>

typedef std::string::iterator It;

int main()
{
  // Input
  std::string f;
  std::getline(std::cin, f);

  // Conversion
  It e = f.end();
  for (It b = f.begin(); b != e; ++b)
```

```
    if (*b == '\r') *b = '\n';

  // Output
  std::cout << f;
  return 0;
}
```

Note that that the variable b of type It, which is an alias for std::string::iterator, is local to the scope of the for loop in this version of the program. %

1.7 Vectors

A string is a container for a sequence of characters. What about containers for variables of other types? It is conceivable that we also want to store and manipulate sequences of numbers. C++ provides such a container, which can be parameterized with a type. This container is a vector, which is also provided by the standard library. To use vectors we have to include the vector library using the directive #include <vector>.

In order to use a vector, e.g., in order to declare a variable of type vector, a template parameter that specifies a sequence of what type the vector stores, has to be provided. For example in the lines

```
std::vector<int>    iv;
std::vector<double> dv;
```

we declare two vector variables, one of type std::vector<int> with the name iv and the other of type std::vector<double> with the name dv. In the first case the template parameter is int and in the second case it is double.

Vectors also provide their own iterator types, that can be used as the iterators for vectors. Let us see an example now. The sieve of Eratosthenes is a method to compute all prime numbers up to a given number n. The idea behind this algorithm is pretty simple: associate with every integer from 2 to n a marker that can take the values "true" and "false". In the beginning set all markers to the value "true". Then go linearly through the numbers from 2 to n. Whenever you find a number that is marked "true", mark all its multiples (besides the number itself) as "false". In the end only the prime numbers are marked "true".

Program: eratosthenes.C

```
// Program: eratosthenes.C
// calculates prime numbers using the sieve
// of Eratosthenes.

#include <iostream>
#include <vector>
```

```
typedef std::vector<bool> Vec;
typedef Vec::size_type    ST;

int main()
{
  // Input
  std::cout << "Compute prime numbers up to (at most "
    << std::numeric_limits<ST>::max() << ") ? ";
  unsigned int n;
  std::cin >> n;

  // Computation and output
  std::cout << "Prime numbers in [0," << n << "): ";
  Vec is_prime(n, true);
  for (ST i = 2; i < n; ++i)
    // Invariant: For all j in [2,n):
    //   (1) j is prime => is_prime[j]
    //   (2) j % k == 0 for any k in [2,i) => !is_prime[j]
    if (is_prime[i]) {
      std::cout << i << " ";
      // mark all multiples of i as not prime
      for (ST m = 2*i; m < n; m += i)
        is_prime[m] = false;
    }
  std::cout << std::endl;
  return 0;
}
```

There are three things remarkable about the program `eratosthenes.C`.

(1) The type alias `typedef Vec::size_type ST;`

(2) The declaration (and initialization) `Vec is_prime(n, true);`

(3) The expression `is_prime[i]` and the assignment `is_prime[m] = false`.

The type alias is easy. The type `Vec`, alias for `std::vector<bool>`, provides a type `size_type` to which we want to refer also as `ST`. For example the type of the return value of the member function `size()`, which gives the number of values in a vector, is `size_type`. It can very well be that `size_type` is just an alias for the type `unsigned int`. At least C++ guarantees us that `ST` behaves like an unsigned integral type, i.e., on `ST` we have the same operators with the same meaning as on `unsigned int`, but the value range could be different.

In the statement `Vec is_prime (n, true);` an object of type `Vec` is declared and initialized to store `n` values (of type `bool`) that are all `true)`. This initialization is an example of a constructor call for the type `vector` - here a constructor that takes two arguments the first one of type `ST` and the second one of type `bool`, i.e., the template parameter we used. Types provide constructors for initialization.

The expression `is_prime[i]` calls the subscript operator `[]` provided by the type `Vec` for the object `is_prime`. The subscript operator can be applied to objects of type `Vec` and takes an argument of type `ST`. The result of this operator is a reference to the $(i-1)$-th `bool` value stored in the vector `is_prime`. Notice that the indexing of a vector starts at zero. We can use this reference to read and manipulate this value. Notice that the indexing of a vector starts at zero. In the `if` statement `if(is_prime[i])` such a value is read, whereas it is manipulated in the assignment `is_prime[m] = false`, i.e., the $(m-1)$th value stored in `is_prime` will be `false` now.

Another remarkable property of vectors they share with strings of type `std::string`) is that they are dynamic, i.e., they do not have constant size and also the amount of computer memory they occupy can change. A special way to grow a string is to append new values at its end with the member function `push_back`. Have a look at the following fragment of a program.

```
typedef std::vector<int>    Vec;
typedef Vec::const_iterator Cit;

Vec a(1,1);
Cit e = a.end();

std::cout << "Length of a is " << a.size()
          << " and the value of its last element is "
          << *--e << "." << std::endl;
a.push_back(2);
e = a.end();
std::cout << "Length of a is " << a.size()
          << " and the value of its last element is "
          << *--e << "." << std::endl;
a.push_back(3);
e = a.end();
std::cout << "Length of a is " << a.size()
          << " and the value of its last element is "
          << *--e << "." << std::endl;
```

leads to the output

```
Length of a is 1 and the value of its last element is 1.
Length of a is 2 and the value of its last element is 2.
Length of a is 3 and the value of its last element is 3.
```

Note that here we use a `const_iterator` provided by the type `std::vector<int>`. We refer to this iterator type using the alias `Cit`. In the statement `Cit e = a.end();` we define a variable of type `Cit` with name `e` and initialize it with `a.end()` (an iterator that points behind the last value stored in the vector named `a`). The expression `*--e` evaluates as follows: first the iterator `e` gets decremented. The return value of this prefix decrement operation is an iterator that points to the last value in the vector `a`. The returned iterator finally gets dereferenced by a call to the dereference operator `*`. The return value of the dereference operator is the value of the last element stored in `a`. The pre-increment and the dereference operator are operators provided by the type `const_iterator`. Why do we use a `const_iterator` here and not an `iterator`, which is also provided by the type `std::vector<int>`? The reason is that we need only read access to the values pointed to by the iterator, i.e., we only send these value to an output stream, and do not intend to modify them. That is, the following would be rejected by the compiler,

```
typedef std::vector<int>    Vec;
typedef Vec::const_iterator Cit;

Vec a(1,1);
Cit e = a.end();

*--e += 1;
std::cout << "Length of a is " << a.size()
          << " and the value of its last element plus one is "
          << *e << "." << std::endl;
```

gives when compiling it with the compiler `g++-3.4.2`,

```
error: assignment of read-only location
```

whereas it works fine if we change the iterator type used,

```
typedef std::vector<int> Vec;
typedef Vec::iterator    It;

Vec a(1,1);
It e = a.end();

*--e += 1;
std::cout << "Length of a is " << a.size()
          << " and the value of its last element plus one is "
          << *e << "." << std::endl;
```

Chapter 2

Functions

2.1 The program `prime.C` revisited

We have seen that we could use the program `prime.C` to test if a number is prime or composite up to $2^{31}-1$, the maximal value of type `int` representable on our computer. Even if we would replace `int` by `unsigned int` we are not able to confirm Cole's result that Mersenne's conjecture is wrong. But according to Mersenne's conjecture, $2^{31}-1$ is a prime number, which we can confirm with our program. When we do this we realize that the test takes quite long (a couple of minutes on our computer). A simple observation might help to speed up the program. Divisors always come in pairs (besides for square numbers). From such a pair, it is enough to test if the smaller of its elements divides the input number. Thus we only have to test potential divisors up the the square root of the input number. But how do we get the square root of a number? Here is a simple algorithm that approximates the square root quite well.

2.1.1 Newton's method

Given some positive number n. We want to approximate the square root of n, i.e., a null of the the function $f(x) = x^2 - n$, which for $n \neq 0$ has two nulls (a positive and a negative one, both with the same absolute value). The idea is to compute a sequence $x_1, x_2, \ldots$ of numbers that converges to one of the nulls. To this end, fix a start value $x_1 \neq 0$ and approximate $f(x)$ by a linear function, whose null we can compute. This null will be the next number x_2 in our sequence. The tangent to the graph $(x, f(x))$ at the point $(x_1, f(x_1))$ is a linear function

$$t(z) = zf'(z) + c.$$

Plugging in $z = x_1$ gives

$$t(x_1) = f(x_1) = x_1 f'(x_1) + c.$$

Resolving for c gives $c = f(x_1) - x_1 f'(x_1)$. Hence

$$t(z) = zf'(x_1) + f(x_1) - x_1 f'(x_1) = (z - x_1)f'(x_1) + f(x_1).$$

The next number x_2 in our sequence is the null of the function t, i.e., $t(x_2) = 0$ from which we get

$$(x_2 - x_1)f'(z) + f(x_1) = 0 \text{ and thus } x_2 = -\frac{f(x_1)}{f'(x_1)} + x_1.$$

Plugging this into $f(x) = x^2 - n$ and $f'(x) = 2x$ gives

$$x_2 = \frac{1}{2}\left(x_1 + \frac{n}{x_1}\right),$$

or in general

$$x_{i+1} = \frac{1}{2}\left(x_i + \frac{n}{x_i}\right).$$

If x_1 is a positive number larger than the null then the sequence $x_1, x_2, \ldots$ converges from the right towards $\sqrt{n}$. Here is program that approximates $\sqrt{n}$ by computing such a sequence.

Program: newton.C

```
// Program: newton.C
// Newton approximation of square root

#include <iostream>
#include <limits>
#include <cassert>

int main()
{
  // Input
  std::cout << "Square root of n > 0 ? ";
  double n;
  std::cin >> n;

  // Newton iteration: x_{i+1} = 1/2 (x_i + n/x_i) We start with
  // x_1:=(n+1)/2, because (n+1)/2-sqrt(n) = (n-2sqrt(n)+1)/2 =
  //                       (sqrt(n)-1)^2/2 > 0 for x_1 > sqrt(n).
  //
  // In theory the sequence x_1, x_2, ... is monotonly decreasing.
  // Here we have to be careful, since the double mantissa is finite.

  double curr = (n + 1.0) / 2.0; // start value
  double prev;                   // prev value

  const double roundoff =
    curr * std::numeric_limits<double>::epsilon();
```

```
  do {
    prev = curr;
    assert(curr > 0);
    curr = (curr + n / curr) / 2.0;
  } while (prev - curr > roundoff);

  std::cout << "The square root of " << n
    << " is approximately " << curr "." << std::endl;

  return 0;
}
```

The value `std::numeric_limits<double>::epsilon()` is the machine epsilon for the type `double` that we pass here as a template argument. We use the machine epsilon to check if we still make progress toward the real solution in machine resolution. Note that this is an application of the formula for the relative approximation error of a real number by a floating point number. The `double` variable named `roundoff` is qualified as constant using the keyword `const`. That means that the value of the variable `roundoff` cannot be changed in the subsequent program. Any attempt to change its value would be rejected by the compiler. Variables that are qualified `const` must be initialized when they are defined. Notice also the use of the macro `assert` from the library `<cassert>`. The macro tests a condition, here `curr > 0`. If this tests fails a message is printed on an output device and the program terminates. The rest of the program should be pretty much straightforward by now. But how does this program help us to achieve our goal to speed up the program `prime.C`? One way to do so is to copy code from `newton.C` into `prime.C`. This gives the following program.

Program: fast_prime1.C

```
// Program: prime.C
// Test if a given integer is a prime number.

#include <iostream>
#include <limits>
#include <cassert>

int main ()
{
  // Input
  int n; // integer to be tested
  std::cout << "Integer larger than one? ";
  std::cin >> n;
```

```
  // Test
  // Compute upper bound for potential divisors first.
  double curr = (n + 1.0) / 2.0; // start value
  double prev;                   // prev value

  const double roundoff =
    curr * std::numeric_limits<double>::epsilon();

  do {
    prev = curr;
    assert(curr > 0);
    curr = (curr + n / curr) / 2.0;
  } while (prev - curr > roundoff);

  // Actual primality test
  curr += 1.0; // upper bound for potential divisors
  int d = 2;   // potential divisor
  while (d <= curr && n%d != 0)
    ++d;

  // Output
  if (d > curr || d >= n)
    std::cout << n << " is a prime number." << std::endl;
  else
    std::cout << n << " is not a prime number and " << d
              << " is the smallest non-trivial divisor of "
      << n << "." << std::endl;
  return 0;
}
```

Let us first comment on a couple of things before we say what we do not like about the solution `fast_prime1.C`. Firstly, this program involves arithmetic expressions that involve values of type `int` and `double`. Remember that in these expressions the `int` values get converted into `double` values. This applies to the expressions

`(n + 1.0) / 2.0` and `(curr + n / curr) / 2.0`.

The same holds for the mixed Boolean expressions `d <= curr` and `d > curr`. Secondly, we had not only to change the condition that controls the `while` loop, but also the condition in the `if` statement that controls the output. The only purpose of the condition `d >= n` left from the logical operator `||` is to take care of the case that `n == 2`.

2.1.2 Encapsulating functionality

What we do not like about the program `fast_prime1.C` is that the computation of the upper bound for potential divisors takes much more space in the code than the actual testing. Furthermore, it is conceivable that we need to approximate the square root of some number also in other programs. We do not want to copy the code of `newton.C` always, as we have done it here, in such cases. The best thing would be if the compiler could do the copying for us. Actually, this is possible if we write a function to approximate a square root, which could be copied by the compiler. A first step in this direction is the following.

Program: fast_prime2.C

```
// Program: prime.C
// Test if a given integer is a prime number.

#include <iostream>
#include <limits>
#include <cassert>

double sqrt(double x);
// PRE:  x >= 0.
// POST: retun value is an approximation of the square
//       root of x.

int main ()
{
  // Input
  int n; // integer to be tested
  std::cout << "Integer larger than one? ";
  std::cin >> n;

  // Test
  int bound = static_cast<int>(sqrt(n)) + 1;
  int d = 2;   // potential divisor
  while (d <= bound && n%d != 0)
    ++d;

  // Output
  if (d > bound || d >= n)
    std::cout << n << " is a prime number." << std::endl;
  else
    std::cout << n << " is not a prime number and " << d
              << " is the smallest non-trivial divisor of "
      << n << "." << std::endl;
```

```
  return 0;
}

double sqrt(double x)
{
  // PRE:  x >= 0.
  // POST: retun value is an approximation of the square
  //       root of x.
  assert(x >= 0);
  if (x == 0) return 0;

  double curr = (x + 1.0) / 2.0; // start value
  double prev;                   // prev value
  const double roundoff =
    curr * std::numeric_limits<double>::epsilon();

  do {
    prev = curr;
    assert(curr > 0);
    curr = (curr + x / curr) / 2.0;
  } while (prev - curr > roundoff);

  return curr;
}
```

In the global scope we declare a function `double sqrt(double x)` that has a parameter of type `double` and returns some value also of type `double`. The name of the function is `sqrt`.

2.1.3 Pre- and post-conditions

The comments

```
// PRE:  x >= 0.
// POST: retun value is an approximation of the square
//       root of x.
```

are an informal contract for the function `sqrt`. If the pre-condition `x >= 0` is fulfilled then the function guarantees the post-condition, which is this case is, that the return value is an approximation of the square root of the value of the parameter named `x`. A pre-condition usually is a restriction on the value range of the parameters of the function. Notice that the compiler has no means to check whether the function complies with its contract or not.

2.1.4 Declaration and definition

After declaration the name of the function is known below in the global scope and its subscopes. We use the function `sqrt` in the initialization of an `int` variable named `bound`. Note that the function is not defined yet, i.e., we have not provided a definition, i.e., implementation, before we use the function `sqrt` in the initialization of a variable named `bound` in the `main` function. That is, the implementation can be deferred. For variables (as we use them here) declaration (announcing the name and the type of the variable) and the definition (reserving memory) are one and the same. As we will see soon it often makes sense to separate declaration and definition of functions. We could have had defined the function `sqrt` at the place where we declared it. In this case the function need not be declared in a separate statement, i.e., we could have just moved the definition up. In this case declaration and definition are again one and the same. The definition of the function `sqrt` defines its own local scope. The name of the parameter `x` is only visible in this scope and so are the names `curr`, `prev` and `roundoff`.

2.1.5 Function call

Let us have now a closer look at the initialization of the variable `bound`.

```
int bound = static_cast<int>(sqrt(n)) + 1;
```

The `static_cast<int>` only converts the value of the expression `sqrt(n)` explicitly from type `double` to type `int`. The expression `sqrt(n)` is a call to the function `sqrt`. When the function `sqrt` is called an actual parameter, here the value of the variable `n`, is passed to the function. The value of the actual parameter is copied into the formal parameter, here variable `x`. Note that the value of the actual parameter in our example is of type `int` whereas the value of the formal parameter is of type `double`. This causes no problem, because before copying the value of the actual parameter can be converted to the type of the formal parameter if this is possible, i.e., if a conversion exists. With the function call the program control goes to function, whose statements are processed according to the usual rules for control flow. With the `return` statement the control is passed back to the calling instance. Note that the lifespan of variables local to the function is limited to the time when the function body is processed. That is, the function `sqrt` return the value of the variable `curr`, but not the associated memory. In order to reuse this value it has to be assigned to variable in the scope the function was called from. That is exactly what we do in the `main` function when we initialize the variable `bound`.

2.1.6 Compilation units

As we said, it is very likely that we need a function to approximate the square root of a number also in other programs. So far we have not reached the goal to reuse our implementation of the function `sqrt`. What we need to do is to implement the function in a different file and then include this file into our program that is exactly what the `#include` directive does. For example we declare the function `sqrt` in a file `sqrt.h` and then include the declaration in our program file `fast_prime.C` using one of the directives

`#include "sqrt.h"` or `#include <sqrt.h>`.

The difference between the two directives is only for the compiler where to look out for a file named `sqrt.h`. In the form `#include "sqrt.h"` the compiler looks in the directory from where it is called, whereas in the form `#include <sqrt.h>` the compiler needs to be given a list of directories where to look for include files. As we said in order to use the name of the function `sqrt` it is enough that it was declared before it is called. We ensure this with the `include` directive, which just copies the declaration from the file `sqrt.h` into the file that contains the `include` directive. See the following example.

Program: fast_prime3.C

```
// Program: prime.C
// Test if a given integer is a prime number.

#include <iostream>
#include "sqrt.h"


int main ()
{
  // Input
  int n; // integer to be tested
  std::cout << "Integer larger than one? ";
  std::cin >> n;

  // Test
  int bound = static_cast<int>(sqrt(n)) + 1;
  int d = 2;   // potential divisor
  while (d <= bound && n%d != 0)
    ++d;

  // Output
  if (d > bound || d >= n)
    std::cout << n << " is a prime number." << std::endl;
  else
    std::cout << n << " is not a prime number and " << d
              << " is the smallest non-trivial divisor of "
      << n << "." << std::endl;
  return 0;
```

```
}
```

The actual implementation of the function `sqrt` is then in a corresponding file `sqrt.C`. Since this file does not contain a main function it cannot be compiled into an executable program but only into an object file that can be linked to an executable file. Therefore the compiler calls another program called the linker. One object file can be linked to many executable files.

2.1.7 Namespaces

What happens if we include also a file `math.h` into the file `fast_prime3.C` that declares amongst other mathematical functions a function `double sqrt(double x);`? We are in trouble then, because the name `sqrt` cannot be linked unambiguously with one definition of the square root function. A way to prevent such name clashes is to use namespaces. Namespaces are named scopes and when used, names should be qualified with the name of the scope by using the scope operator `::`. The file `sqrt.h` would then look as follows

Library: sqrt.h

```
// Library: sqrt.h
// provides function declaration for a function to
// approximate the square root of a double value.

namespace inf1 {
  double sqrt(double x);
  // PRE:  x >= 0.
  // POST: retun value is an approximation of the
  //       square root of x.
} // namespace inf1
```

and we would have to replace the line in which the function `sqrt` is called in the program `fast_prime3.C` as follows

```
int bound = inf::sqrt(n) + 1;
```

2.1.8 Qualified name lookup

Here `inf::sqrt` is a qualified name. If the function name `sqrt` would not be qualified then the name look up would work as for variable names, i.e., the compiler first looks in the scope where the function name appears, e.g., where the function is called. If it does not find a declaration in this scope it continues its search in the next enclosing super-scope. This continues until a scope is found that contains an appropriate declaration or the top most scope, the global scope, is reached. This process is called unqualified name lookup. As we have seen names can be qualified using the binary scope operator `::`. The left hand side argument of the scope operator `::` must be either the name of a namespace or the name of a class. Its right hand side argument can be any name, e.g., the name of namespace, function or variable. The binary scope operator is left associative and has precedence 17. In the qualified name lookup, the name on the left hand side of the scope operator is looked up unqualified, as described above, and then the name on its right hand side is looked up in the scope (named scope, i.e., namespace, or class scope) given as the left hand side argument. It is only looked up in top-down manner in the latter scope but not in its sub-scopes.

2.2 Call by reference and `void` functions

Functions as we seen them so far basically act as "mathematical functions" in the sense that they map the values of the function parameters to the return value. Sometimes it is desirable to have more powerful functions which could be used to do something like the following.

```
int x = 7;
int y = 8;
inf::swap(x, y);
std::cout << "Value of x = " << x
          << " and value of y = " << y
          << "." << std::endl;
```

should produce the following output.

```
Value of x = 8 and value of y = 7.
```

That is, the the function `swap` swaps the values of the variables `x` and `y`. But that seems to be impossible if the the function `swap` only works with copies of the values of `x` and `y`. Another thing is remarkable about our use of the function `swap`. If it has a return value then we do not make use of it. Actually, to serve its purpose namely to swap the values of the variables `a` and `b` the function `swap` need not have a return value at all. It is possible in C++ to write a function that provides exactly the functionality of `swap` as we have used it above and nothing more.

Library: swap.h

```
namespace inf1{
  void swap(int& a, int& b)
    // POST: the values of a and b have been swapped
```

```
    {
      int c = a;
      a = b; b = c;
    } // swap
} // namespace inf1
```

2.2.1 The type `void`

The function `swap` as implemented above has no return value. This is indicated in the declaration

```
void swap(int& a, int& b);
```

of the function `swap`. Here the type of the return value is `void`, which is another built-in type whose use is very restricted. Note that we do not explicitly return a value of type `void`, there are no such values. Equivalently, we could have added as the last statement in the body of the function `swap`

```
return;
```

which indicates that program control is given back to the calling instance, but no value is returned to this instance.

2.2.2 Reference types

If the function `swap` does not a return a value then its functionality must be implicit or as one says, it is a side effect. Note that the parameters of the function `swap` are of type `int&`. The type `int&` is a reference type. A variable of reference type just serves as another name for an already defined variable, i.e., the name of the reference variable is an alias for an already defined variable. When used as function parameters that means that the names of the formal parameters of the function that are of reference type serve as an alias for the actual parameters. In our swap example the variable name `a` is an alias for the variable `x` and the name `a` is an alias for the variable `y`. The function swap can now access and modify the values associated with `x` and `y` by addressing them under the names `a` and `b`, respectively. That is how the function `swap` can swap the values of its actual parameters. One says that the function `swap` calls its parameters by reference opposed to the ordinary ”call by value”, where the values of the actual parameters get copied into the formal parameters and thus only the values of the actual parameters get passed to the function. Just to avoid confusion a function can have both reference and value parameters and it can also have both, a return value and side effects.

Any type (besides `void`) has a corresponding reference type. Syntactically a type and its reference type are distinguished by the symbol `&` used for reference types. Reference types cannot only be used as function parameters though that is where they make the most sense. They can be used as ordinary variables to serve as a variable name alias, but some care has to be taken. The following points are important when dealing with types.

(1) A variable of reference type must be initialized. Since the only purpose of such a variable is to serve as an alias for another variable its sole name (as in a declaration without initialization) is meaningless.

(2) On reference types we have the same operations as on the corresponding (ordinary) type. But the effect of these operations can affect the value of the variable aliased by the reference variable as the following example shows.

```
double a = 1.0;
double &b = a;
b += 1.0; // The value of a is now 2.0
double c = 3.0;
b = c; // The value of a is now 3.0
```

2.3 Recursion

In our final implementation of the program `fast_primes.C` we have called the function `inf::sqrt` from the function main. This is an example where a function calls another function. A function can call also itself, either directly or indirectly via other function calls. Such a function is called a recursive function.

Many mathematical functions have a quite natural recursive definition. We will see two examples in this section. Firstly, the greatest common divisor of two natural numbers and secondly, the Fibonacci numbers.

2.3.1 Greatest common divisor

The function

$$gcd : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

gives the greatest common divisor of its two arguments and has the following remarkable property.

Lemma 1 $gcd(a, b) = gcd(b, a\%b)$ *if* $b > 0$.

PROOF. Let $d = gcd(a, b)$ and $c = gcd(b, a\%b)$. Our goal is to show that c and d are mutual divisors of each other. But then it must hold $d = c$.

Let us first show that d divides c. It holds $a\%d = b\%d = 0$, because by definition d is a divisor of both a and b. It also holds for the remainder $a\%b$ of the integer division a/b,

$$a\%b = a - (a/b) * b.$$

This implies

$$\begin{aligned}(a\%b)\%d &= (a-(a/b)*b)\%d \\ &= a\%d-((a/b)*b)\%d \\ &= 0-0 \;=\; 0.\end{aligned}$$

Thus d is a divisor of both b and $a\%b$. But then it must also be a divisor of the greatest common divisor of b and $a\%b$. Thus d divides c.

Now let us show that c divides d. It holds $b\%c = (a\%b)\%c = 0$, because by definition c is a divisor of both b and $a\%b$. It also holds

$$a = a\%b + (a/b) * b.$$

This is, actually, the same formula for the remainder $a\%b$ that we have used above only resolved for a. We get

$$\begin{aligned}a\%c &= (a\%b+(a/b)*b)\%c \\ &= (a\%b)\%c+((a/b)*b)\%c \\ &= 0+((a/b)*b)\%c \\ &= 0+0 \;=\; 0,\end{aligned}$$

where $((a/b) * b)\%c = 0$ follows from $b\%c = 0$. Thus c is a divisor of both b and a. But then it must also be a divisor of the greatest common divisor of b and a. Thus c divides d. □

If we stick to the convention $gcd(b, 0) = gcd(0, b) = 0$ then the lemma suggests the following recursive function to compute the greatest common divisor of two `unsigned int` values. This function is an implementation for the type `unsigned int` of Euclid's algorithm to compute the greatest common divisor of two natural numbers.

Library: gcd.h

```
namespace inf1{
  unsigned int gcd(unsigned int a, unsigned int b)
    // POST: return value is the greatest common divisor
    //       of a and b, where gcd(n,0):=gcd(0,n):=n.
    {
      if (b == 0) return a;
      return gcd(b, a % b);
    } // gcd
} // namespace inf1
```

A natural question to ask is whether there can be an infinite sequence of function calls to `gcd` triggered by one call to it, i.e., the question is whether the algorithm that is encoded in the function `gcd` always terminates. For Euclid's algorithm, termination is easy to see. The termination condition is `b == 0` since in this case there is no recursive call to `gcd`. In the recursive calls to the function `gcd`, the second argument always gets smaller and decreases by at least one. Furthermore the argument cannot get smaller then zero. This proves that there can only be finitely many recursive calls to the function `gcd` and thus Euclid's algorithm terminates.

Note that the termination condition is crucial. Every recursive function needs such a condition to prevent an infinite recursion. Let us now turn to another example.

2.3.2 Fibonacci numbers

Fibonacci numbers are a sequence of natural numbers that are recursively defined as follows:

$$F_0 = 0, F_1 = 1, F_i = F_{i-1} + F_{i-2}, i \geq 2.$$

This recursive definition can immediately be translated into a C++ function.

Library: fibonacci.h

```
namespace inf1{
  unsigned int fib(unsigned int n)
    // POST: return value is the n'th Fibonacci number F_n,
    //       where F_0:=0, F_1:=1 und F_i:=F_{i-1}+F_{i-2},
    //       for i > 1.
    {
      if (n == 0) return 0;
      if (n <= 2) return 1;
      return fib(n-1) + fib(n-2);
    } // fib
} // namespace inf1
```

This function looks innocent at a first glance, but it can be shown that the computation of the n'th Fibonacci number leads to exponentially many (in n) recursive function calls of the function `fib`. Thus it is conceivable that the evaluation of call to `fib` will be slow. Such a call consumes also much of (another) limited resource, namely, memory, though here the consumption is not exponential in n. To see this, imagine that a recursive function call opens another local scope (the scope associated with the function). The variables of the function are local to this scope and hide the variable names in the calling scope. Notice that here the only local variable is `n`. That is, for every open scope, i.e., function body that has not been fully processed, there is a copy of the local variable `n` associated with its own memory and value (because the parameters of the function `fib` are called by value). The memory associated with a copy of `n` will only be deallocated (freed) when the program control leaves the function's scope, i.e., when all statements in the function body have been processed. Since the allocation and deallocation of a local variable happens automatically it is called automatic memory allocation. Note that all local variables are allocated and deallocated automatically. There is nothing special about the local scope associated with a function. The life span of a local variable is from the point when the program control reaches its definition until it leaves the scope in which it is defined.

Often, there is a better iterative alternative to a recursive implementation of a function. For example, the following function also computes Fibonacci numbers but is much faster (the running time should be proportional to the value of the parameter n whereas it is proportional to F_n for the recursive implementation) and consumes less memory than the recursive implementation. In fact, in the implementation below, the memory consumption is known at compile time whereas in the recursive implementation it depends on the (at compile time probably unknown) parameter n.

Library: fibonacci2.h

```
namespace inf1{
  unsigned int fib(unsigned int n)
    // POST: return value is the n'th Fibonacci number F_n,
    //       where F_0:=0, F_1:=1 und F_i:=F_{i-1}+F_{i-2},
    //       for i > 1.
    {
      if (n == 0) return 0;
      if (n <= 2) return 1;
      unsigned int a = 1, b = 1;
      for (unsigned int i = 3; i <= n; ++i) {
        unsigned int c = a; a = b; b += c;
      }
      return b;
    } // fib
} // namespace inf1
```

2.4 Parser for arithmetic expressions

A programming language like C++ is a formal language. That is, it follows formal rules that can be checked by a compiler. The compiler checks if a program that we submit to the compiler obeys the rules of the language. If this is not the case then it stops the compilation and issues a syntax error. The rules of a formal language are encoded in a grammar. But first we should say what a language is.

2.4.1 Languages

A language consists of words of an alphabet. Here an alphabet is a finite set (of symbols) Σ, i.e., the ordinary English alphabet is just a special case of an alphabet as we want to consider them here. A word of length n is an element of the set $\Sigma^n = \Sigma \times \cdots \times \Sigma$, i.e., of the n-th Cartesian product of Σ with itself. We set $\Sigma^0 = \{\epsilon\}$, where ϵ is the empty word, and

$$\Sigma^* = \bigcup_{i \geq 0} \Sigma^i.$$

A language L is a subset of Σ^*.

2.4.2 Grammars

Grammars are a way of specifying a language in T^*, where T is an alphabet, whose elements are called terminal symbols. In order to specify the language, a second alphabet N is used. The symbols in N are called non-terminal symbols. N contains a special symbol s called the starting symbol. The last ingredient of a grammar is a set of production rules. Production rules are of the form $\alpha \to \beta$, where $\alpha, \beta \in (T \cup N)^*$. The rules have to be read as "α can be replaced by β". We say that γ can be derived directly from δ if $\delta = \rho\alpha\sigma$ and $\gamma = \rho\beta\sigma$ with $\rho, \sigma \in (T \cup N)^*$ and there is a production $\alpha \to \beta$. In this case we write $\delta \Rightarrow \gamma$. A word γ can be derived from a word δ if there is a sequence of words $\sigma_1, \ldots, \sigma_k$, all in $(T \cup N)^*$, such that

$$\delta \Rightarrow \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_k \Rightarrow \gamma.$$

In this case we write $\delta \Rightarrow^* \gamma$. To summarize, a grammar is a 4-tuple $G = (T, N, s \in N, P)$, where P is the set of production rules. The language $L(G)$ defined by G is the following set

$$L(G) = \{\sigma \in T^* \mid s \Rightarrow^* \sigma\}.$$

That is, $L(G)$ contains all words that can be derived from the start symbol s (here considered as a word of length one) and contain only terminal symbols.

2.4.3 Examples: C++ integer and floating point literals

The integer and floating point literals are defined by grammars in the C++ standard. In a (simplified) version the grammar for integer literals reads as follows. Note that in the way of writing productions below, the arrows are replaced by colons (`:`), where each separate line after the colon represents one way of replacing the symbol before the colon. Also, `ONE OF` is a shorthand for listing all the given alternatives in separate rows.

TERMINAL SYMBOLS.
`0, 1, 2, 3, 4, 5, 6, 7, 8, 9,`
`u, U .`

NON-TERMINAL SYMBOLS.
<u>*integer-literal*</u>, *integer-suffix*,
suffix, *digits*, *digit*.

PRODUCTIONS.

integer-literal:
  digit digits integer-suffix

digits:
  digit digits
  ϵ

integer-suffix:
  suffix
  ϵ

digit: ONE OF
  `0 1 2 3 4 5 6 7 8 9`

suffix: ONE OF
  `u U`

Let us check using this grammar if our examples for integer literal constants are correct. The first example was `127` for an `int` literal constant. We have to show that `127` can be derived from the starting symbol *integer-literal*. The following is a possible derivation of `127`.

$$\begin{array}{rcl} \textit{integer-literal} & \Rightarrow & \textit{digit digits integer-suffix} \\ & \Rightarrow & \texttt{1}\ \textit{digits integer-suffix} \\ & \Rightarrow & \texttt{1}\ \textit{digit digits integer-suffix} \\ & \Rightarrow & \texttt{12}\ \textit{digits integer-suffix} \\ & \Rightarrow & \texttt{12}\ \textit{digit digits integer-suffix} \\ & \Rightarrow & \texttt{127}\ \textit{digits integer-suffix} \\ & \Rightarrow & \texttt{127}\ \textit{integer-suffix} \\ & \Rightarrow & \texttt{127}\ \textit{suffix} \\ & \Rightarrow & \texttt{127} \end{array}$$

As an example for an `unsigned int` literal constant we had given `0u`, which can be derived as follows from the starting symbol *integer-literal*.

$$\begin{array}{rcl} \textit{integer-literal} & \Rightarrow & \textit{digit digits integer-suffix} \\ & \Rightarrow & \texttt{0}\ \textit{digits integer-suffix} \\ & \Rightarrow & \texttt{0}\ \textit{integer-suffix} \\ & \Rightarrow & \texttt{0}\ \textit{suffix} \\ & \Rightarrow & \texttt{0u} \end{array}$$

We can even argue that `1.0` cannot be derived from the starting symbol since the dot `.` is not a terminal symbol for integer literals. But what about the word `7u3`, which consists only of terminal symbols? Here we can argue that there is no production to put digits behind the suffix, which would be needed here.

As our second example for a grammar let us now turn to floating point literals. Again we present a simplified version here.

TERMINAL SYMBOLS.

`0, 1, 2, 3, 4, 5, 6, 7, 8, 9,`
`., e, E, −, +, f, F .`

NON-TERMINAL SYMBOLS.

float-literal, *float-exponent*, *exponent*, *float-suffix*, *suffix*, *exp-sign*, *sign*, *digits*, *digit*.

PRODUCTIONS.

float-literal:
  digit digits . digits float-exponent float-suffix
  . digit digits float-exponent float-suffix

digits:
  digit digits
  ϵ

float-exponent:
  exponent exp-sign digit digits
  ϵ

exp-sign:
  sign
  ϵ

float-suffix:
  suffix
  ϵ

digit: ONE OF
  `0 1 2 3 4 5 6 7 8 9`

exponent: ONE OF
  `e E`

sign: ONE OF
  `− +`

suffix: ONE OF
  `f F`

First of all it is easy to see that every floating point literal constant has to contain a dot (`.`), because all words that can be derived from the starting symbol *float-literal* have to contain a dot. Furthermore, floating point literal constants need not have digits both before and after the dot, but there must at least be either a digit before or after the dot. As an example let us derive `1.2e−7f`.

$$
\begin{aligned}
\textit{float-literal} &\Rightarrow \textit{digit digits}\ \texttt{.}\ \textit{digits float-exponent float-suffix}\\
&\Rightarrow \texttt{1}\ \textit{digits}\ \texttt{.}\ \textit{digits float-exponent float-suffix}\\
&\Rightarrow \texttt{1.}\ \textit{digits float-exponent float-suffix}\\
&\Rightarrow \texttt{1.}\ \textit{digit digits float-exponent float-suffix}\\
&\Rightarrow \texttt{1.2}\ \textit{digits float-exponent float-suffix}\\
&\Rightarrow \texttt{1.2}\ \textit{float-exponent float-suffix}\\
&\Rightarrow \texttt{1.2}\ \textit{exponent exp-sign digit digits float-suffix}\\
&\Rightarrow \texttt{1.2e}\ \textit{exp-sign digit digits float-suffix}\\
&\Rightarrow \texttt{1.2e}\ \textit{sign digit digits float-suffix}\\
&\Rightarrow \texttt{1.2e-}\ \textit{digit digits float-suffix}\\
&\Rightarrow \texttt{1.2e-7}\ \textit{digits float-suffix}\\
&\Rightarrow \texttt{1.2e-7}\ \textit{float-suffix}\\
&\Rightarrow \texttt{1.2e-7f}
\end{aligned}
$$

2.4.4 Parsing Arithmetic Expressions

One thing remarkable about both integer and floating point literal constants is that they are all non-negative. The negative values can be obtained by applying the unary sign operator − to the positive constants. Another remarkable thing about these grammars is the following: if we read a word from left to right that can be derived from the starting symbol, then the symbol that we read unambiguously determines which production rule to apply. If we do so for a word that cannot be derived from the starting symbol then we will get stuck at some point. This really remarkable fact is also true for the following grammar for simple additive arithmetic expressions.

TERMINAL SYMBOLS.
`0, 1, 2, 3, 4, 5, 6, 7, 8, 9,`
`+, -` .

NON-TERMINAL SYMBOLS.
<u>*add-expr*</u>, *add-expr-aux*,
number, *pm-expr*, *digits*, *digit*.

PRODUCTIONS.

add-expr:
    number add-expr-aux

add-expr-aux:
    + *number add-expr-aux*
    − *number add-expr-aux*
    ε

number:
    digit digits

digits:
    digit digits
    ε

digit: ONE OF
    `0 1 2 3 4 5 6 7 8 9`

Let us also shortly discuss three examples. We want to check if `12+3` and `1+a` and `1a` are words in the language described by the grammar above. Let us start with `12+3`.

$$
\begin{aligned}
\textit{add-expression} &\Rightarrow \textit{number add-expr-aux}\\
&\Rightarrow \textit{digit digits add-expr-aux}\\
&\Rightarrow \texttt{1}\ \textit{digits add-expr-aux}\\
&\Rightarrow \texttt{1}\ \textit{digit digits add-expr-aux}\\
&\Rightarrow \texttt{12}\ \textit{digits add-expr-aux}\\
&\Rightarrow \texttt{12}\ \textit{add-expr-aux}\\
&\Rightarrow \texttt{12+}\ \textit{number add-expr-aux}\\
&\Rightarrow \texttt{12+}\ \textit{digit digits add-expr-aux}\\
&\Rightarrow \texttt{12+3}\ \textit{digits add-expr-aux}\\
&\Rightarrow \texttt{12+3}\ \textit{add-expr-aux}\\
&\Rightarrow \texttt{12+3}
\end{aligned}
$$

That shows that `12+3` is in our language. What about `1+a`?

$$
\begin{aligned}
\textit{add-expression} &\Rightarrow \textit{number add-expr-aux}\\
&\Rightarrow \textit{digit digits add-expr-aux}\\
&\Rightarrow \texttt{1}\ \textit{digits add-expr-aux}\\
&\Rightarrow \texttt{1}\ \textit{add-expr-aux}\\
&\Rightarrow \texttt{1+}\ \textit{number add-expr-aux}\\
&\Rightarrow \texttt{1+}\ \textit{digit digits add-expr-aux}
\end{aligned}
$$

Since the non-terminal symbol *digit* cannot be replaced by the symbol `a` the word `1+a` cannot be in our language. Our final example is `1a`.

$$\begin{aligned} \textit{add-expression} &\Rightarrow \textit{number add-expr-aux} \\ &\Rightarrow \textit{digit digits add-expr-aux} \\ &\Rightarrow \texttt{1}\ \textit{digits add-expr-aux} \\ &\Rightarrow \texttt{1}\ \textit{add-expr-aux} \end{aligned}$$

The non-terminal symbol *add-expr-aux* cannot be replaced by the symbol `a`. Thus the word `1+a` cannot be in our language.

We want to write a program to check the correctness (with respect to the grammar) of an additive arithmetic expression and to evaluate this expression if it is correct. The program makes use of the property that if we read an arithmetic expression from left to right, it is always uniquely determined by the currently read symbol which production rule to apply. If no rule applies then the expression cannot be correct.

Library: calc-simple.h

```
// Library: calc-simple.C
// Parser for additive arithmetic expressions.

// Uses the following grammar:
//   add_expr:         number add_expr_aux
//   add_expr_aux:     '+' number add_expr_aux
//                     '-' number add_expr_aux
//                     epsilon
//
//   number:           digit digits
//
//   digits:           digit digits
//                     epsilon
//
//   digit:            '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'

#include <iostream>
#include <string>
#include <cctype>
#include <stdexcept>

namespace inf1{
  // Parse functions for every non-terminal symbol:
```

```
// --------------------------------------------------

// To every non-terminal symbol of the grammar there is a
// function, which has the name of the symbol. All these
// functions have to following pre- and post conditions.
//
// PRE: [b,e) is a valid range.
// POST: ???

// NT for evaluation of expression
typedef int NT;

// iterator type
typedef std::string::const_iterator SSCI;

NT digit(SSCI& b, SSCI e)
  // PRE: b can be dereferenced.
{
  return *(b++) - '0';
}

NT digits(SSCI& b, SSCI e, const NT& left_op)
{
  if (b == e || !std::isdigit(*b))
    return left_op;
  NT val = left_op * 10 + digit(b, e);
  return digits(b, e, val);
}

NT number(SSCI& b, SSCI e)
{
  if (b == e || !std::isdigit(*b))
    throw std::invalid_argument("digit expected.");
  NT val = digit(b, e);
  return digits(b, e, val);
}

NT add_expr_aux(SSCI& b, SSCI e, const NT& left_op)
{
if (b == e) return left_op;
if (*b == '+') {
  NT right_op = number(++b, e);
  return add_expr_aux(b, e, left_op + right_op);
}
```

```
  if (*b == '-') {
    NT right_op = number(++b, e);
    return add_expr_aux(b, e, left_op - right_op);
  }
  return left_op;
  }

  NT add_expr(SSCI& b, SSCI e)
  {
    NT value = number(b, e);
  return add_expr_aux(b, e, value);
  }
} // namspace inf1
```

Below is a little program written to test the parser.

Program: calc-simple.C

```
#include "calc-simple.h"

int main()
{
  std::cout << "Eingabe: ";
  std::string input;
  std::cin >> input;

  SSCI beg = input.begin();
  SSCI end = input.end();
  SSCI cur = beg;
  try {
    std::cout << add_expr(cur, end) << std::endl;
    if (cur != end)
      throw std::invalid_argument("Unexpected character.");
  } catch (std::invalid_argument err) {
    std::cerr << "Parse error: " << err.what() << "\n"
              << std::string(beg, cur)
              << "|-here->"
              << std::string(cur, end) << std::endl;
    return 1;
  }
  return 0;
}
```

Let us check how this program behaves under our three examples `12+3`, `1+a` and `1a`.

On input `12+3` the function call sequence looks as follows: The *main* function calls the function for *number*, which calls *digits*, which calls *digit*, which "consumes" the symbol `1` from our input string and returns control to the calling instance of the function *digits*, which then calls another instance of *digits*, which calls *digit*, which "consumes" the symbol `2` from the input string and returns control to its calling function *digits*, which returns control to its calling instance of the function *number*, which itself returns control to its calling instance of the function *add-expr*, which now calls *add-expr-aux*, which "consumes" the symbol + from the input string and calls *number*, which calls *digits*, which calls *digit*, which "consumes" the symbol `3` from the input string and returns control to its calling instance of the function *digits*, which returns control to its calling instance of the function *number*, which itself returns control to its calling instance of the function *add-expr-aux*, which itself returns control to its calling instance of the function *add-expr*, which finally returns control the function *main*.

On input `1+a` the function call sequence looks as follows: The *main* function calls *number*, which calls *digits*, which calls *digit*, which "consumes" the symbol `1` from the input string returns control to its calling instance of the function *digits*, which itself returns control to its calling instance of the function *number*, which returns control to its calling instance of the function *add-expr*, which now calls *add-expr-aux*, which "consumes" the symbol + from the input string and calls *number*. Since the next symbol in the input string is not a digit the function *number* enters an exceptional state and throws an exception object of type `std::invalid_argument`. The execution of the function *number* is stopped at this point and the program control goes through the scope hierarchy until either the exception object gets caught with a `try catch` statement or it reaches the top most level, in which case the program terminates. In our example the exception object is caught within the main function where an error message is issued to the stream object `std::cerr`, which like `std::cout` usually is linked to the monitor as output device. Afterwards the program is terminated.

On input `1+a` the function call sequence looks as follows: The *main* function calls the function *number*, which calls *digits*, which calls *digit*, which "consumes" the symbol `1` from the input string and returns control to its calling instance of the function *digits*, which itself returns control to its calling instance of the function *number*, which itself returns control to its calling instance of function *add-expr*, which now calls *add-expr-aux*. Since the next symbol in the input string is neither + nor − the function*add-expr-aux* returns control to the function *add-expr*, which itself returns control to the `main` function. In the `main` function an exception object of type `std::invalid_argument` is thrown and also caught.

2.4.5 Exceptions

Let us say a little bit more about exceptions. When a program throws an exception, execution of the program stops at that part of the program and control is passed to another part of the program together with an exception object. In our example the exception object is of type `std::invalid_argument`, which is defined in the standard library. To use it we have to include the `stdexcept` library. Exception objects can be caught in a `try catch` statement. The general form of a `try catch` statement is.

```
try {
  statement_1
      ...
  statement_n
} catch (execption-type name) {
  statement_(n+1)
      ...
  statement_m
}
```

If in the execution of any of the statements in the `try` block an exception is thrown then the execution of the `try` block stops at that point. If the thrown exception is of type `exception-type` then the statements in the catch block get executed. The thrown exception object can be accessed under the name `name` in the `catch` block. An exception object provides a member function `what` which can be called through the object using the exceptions objects name and the dot operator `.`. The value returned by `what` in general reports more specifically what caused the exception. It is possible to omit the name. Then the thrown exception cannot be accessed and only a general, i.e., independent of the specific object, exception handling based only on the exception type can be performed. If this is not done, the program continues behind the `catch` block. Also, if no exception is thrown when executing the `try` block then the program continues behind the `catch` block.

2.5 Overloading function names

We have already described how the compiler looks up a declaration of a variable or function whose name is used at some place of a program. We have distinguished between unqualified and qualified name lookup. They differ in how the scope where to search for the name is defined. What we have not said so far is, what happens if there is not only one but several declarations for the name that is looked up in the scope where the name is found. Is it even possible to have several declarations for the same name? For variables this is not possible. That is, a variable name can be declared only once in the same scope. As the following example shows anything else would not make sense.

```
int a = 7;             // First declaration using the name a
unsigned int a = 8u; // Illegal declaration using also the
                       // name a
a += 1.0;              // which variabledo we mean?
```

Note that it is possible though to declare the same variable name also in another scope. Which variable is associated with a name is then determined by the name lookup rules. We have already seen examples of this.

For functions, the situation is different. The same function name can be used several times in declarations in the same scope. All these functions need to have different parameter lists. It is not enough that the functions differ only in the type of their return value. See the following examples.

```
int foo(); // declaration of function foo;
void foo(); // illegal redeclaration (a function with the same
            // name and parameter list was already declared in
            // the same scope).
void foo(int& a); // legal redeclaration of function foo
void foo(int a);  // legal declaration of function foo (no
                  // function with the same name and parameter
                  // list is declared in the same scope).
void foo(int& a){} // legal defintion of function foo
void foo(int& b){  // illegal redefinition of function foo
  std::cout << "foo." << std::endl;
}
```

Function overload resolution. If we call a function named `foo`, the compiler has to attach to the call a definition of a function `foo`. Therefore, the compiler first looks up the name `foo` using the rules for name look up that we have discussed earlier. It is possible that the compiler finds several declarations of a function named `foo` among which one has to be chosen if possible. All functions found in the name lookup are candidates to be attached to the call. The function overload resolution mechanism consists of the following three steps to find the function to attach, or to decide that the function call is ambiguous.

(1) Remove all functions from the candidate list where the number of formal parameters is different from the number of actual parameters.

(2) Remove all functions from the candidate list where the types of the actual parameters do not match or cannot be converted into the types of the formal parameters.

(3) Select the unique function that best matches the call. If no unique best match exists then the call is ambiguous.

It remains to clarify what a best match is. For every single parameter, ”best” is defined with respect to a four level hierarchy of conversions:

(1) EXACT MATCH. The type of the actual and the formal parameter are the same. For a given type `T`, the types `T&` and `const T&` are considered the same as `T` for the purpose of overload resolution.

(2) PROMOTION. These are conversions that are guaranteed to be safe by the C++ standard, i.e., no information gets lost in such a conversion. The only promotions we will encounter here are: from `char` to `int`, from `bool` to `int` and from `float` to `double`.

(3) CONVERSIONS. Not necessarily safe, but according to the C++ standard allowed conversions. Examples are: from `int` to `unsigned int` or from `int` to `double`.

(4) USER DEFINED CONVERSIONS. These we will encounter later in the chapter on classes.

Here, of course, an EXACT MATCH is better than a PROMOTION, which is better than a CONVERSION, which is better than a USER DEFINED CONVERSION.

Now, a function is called a *best match*, if it is for no argument worse then any other candidate (from the final candidate list), but it is better in at least on argument compared to any other candidate. Under this definition, it may happen that there is no best match, in which case the call is ambiguous, and the program is rejected by the compiler.

We conclude this section with an example for function overload resolution.

```
void foo(double d){
  // POST: Prints foo(double) on standard output
  std::cout << "foo(double) << std::endl;
}

void foo(unsigned int u){
  // POST: Prints foo(unsigned int) on standard output
  std::cout << "foo(unsigned int) << std::endl;
}

float f = 1.0f;
foo(f); // attached to foo(double), because float can be
        // promoted to double
int i = 1;
foo(i); // ambiguous function call, since int to double
        // and int to unsigned int are both conversions
```

Chapter 3

Classes

3.1 Example: Rational Numbers

Suppose we want to use *rational numbers* in a program, i.e. numbers of the form n/d, where both numerator and denominator are integers. `C++` has no built-in type for rational numbers, so we have to implement them ourselves.

In order to do this, we need some mechanism to introduce new types (the `typedef` statement we have seen only introduces a *new name* for an already existing type). Ideally, the new type `Rational` (which we plan to put into namespace `inf`) should be usable like other number types; the following piece of code demonstrates how this could look like.

```
int main() {
  inf::Rational q(1, 2);        // q = 1/2
  inf::Rational r(2, 3);        // r = 2/3
  inf::Rational x = q + r;      // x = 7/6
  inf::Rational y = q - r;      // y = -1/6

  std::cout << "q   = " << q << "\n"
            << "r   = " << r << "\n"
            << "q+r = " << x << "\n"
            << "q-r = " << y << std::endl;

  return 0;
}
```

In particular, we expect that rational numbers can be added and subtracted from each other, and that they can be written to standard output. These are expectations on the *functionality* of rational numbers. In `C++`, the concept of *classes* is used to define a new type *through* its functionality. In designing a class we therefore have to think about the desired functionality first.

We have already seen another type which is actually realized as a class, namely the type `std::vector<T>`. Other people have thought about the functionality vectors should have, and they have realized it through types and functions belonging to the class. Here are some examples we have seen.

class	functionality	realization
`std::vector<T>`	size of `v`	`v.size()`
	start iterator of `v`	`v.begin()`
	past-the-end iterator of `v`	`v.end()`

Because the realization is only the second step, let's draft the first two columns of a similar table for `inf::Rational`. This table may not be complete for the rational numbers the reader has in mind, but it lists some functionality that makes the type already quite useful.

class	functionality
`inf::Rational`	construction of `q` from two integers
	construction of `q` from one integer (conversion)
	access to numerator of `q`
	access to denominator of `q`
	addition of `q` and `r`
	subtraction of `q` and `r`
	multiplication of `q` and `r`
	division of `q` and `r`

The reason for not listing input (from `std::cin`, say) and output (to `std::cout`, say) is that this is more naturally realized as functionality of the respective streams (we'll see why).

3.2 Class Declaration

The informal specification of the functionality attached to the rational numbers can more or less directly be translated into a `C++` class declaration. Within namespace `inf`, we would write the following piece of code, parts of which will still be obscure to the reader and only explained below. The point here is that there is essentially a one-to-one mapping between the informal functionality specified in the previous section and the `public` part of the formal declaration in terms of a class. This mapping is only 'essentially' one-to-one, because we have the four arithmetic operators only in the variants `+=`, `-=`, `*=` and `/=`; we will see that there is good reason for this.

```
// Class for representing rational numbers
class Rational {
public:
  typedef int NT; // allows us to replace 'int' easily

  Rational(const NT& n, const NT& d);
  // PRE: d != 0.
  // POST: rational number initialized as n / d.
```

```
  Rational(const NT& n);
  // POST: rational number initialized as n / 1.

  const NT& n() const;
  // POST: return value is the numerator

  const NT& d() const;
  // POST: return value is the denominator

  Rational& operator+=(const Rational& x);
  // POST: x was added to *this

  Rational& operator-=(const Rational& x);
  // POST: x was subtracted from *this

  Rational& operator*=(const Rational& x);
  // POST: *this was multiplied by x

  Rational& operator/=(const Rational& x);
  // PRE: x.n() != 0
  // POST: *this was divided by x

private:

  void normalize();
  // POST: the value remains unchanged, but numerator and
  // denominator are relatively prime, and d is positive

  NT n_; // numerator
  NT d_; // denominator

  // Invariant: d_ > 0 and (n_,d_) is normalized
};
```

3.2.1 Member functions

The functions listed in the scope of the class declaration are called *member functions* of the class, because they 'belong' to the class. The meaning of this is that member functions are always called *for* an object of the class, and this object is an implicit parameter of the function. For example, the call `q.n()` returns the numerator of the rational number `q`. The `const` qualifier *behind* the declaration of a member function refers to this implicit parameter. In case of the member function `const NT& n() const`, for example, the qualifier indicates that `q` will not be changed through a call to `q.n()`.

If the member function needs to access the implicit parameter for which it was called, some special syntax is necessary; unlike the other parameters, the implicit one does not have a name within the function's scope. The keyword `this` encodes a 'pointer' to the implicit parameter, and just like with iterators, `*this` is the parameter itself, obtained through dereferencing.

Just to avoid confusion with the various `const`'s, recall that the *first* `const` in the declaration of `n()` refers to the return type `const NT&`: a reference to the numerator which does not allow write access to the referenced variable. We could have replaced `const NT&` globally with `NT` without changing the way the class can be used, and if `NT` is an alias for `int`, there is not much of a difference between the two variants. However, if we later decide to replace `int` by some other type representing integers, the call by value semantics might be very costly in case objects of this other type need large amounts memory for some reason.

The class declaration might as well contain the full member function definition, rather than just the declaration. However, clean pre- and postconditions should already specify the behavior of the member function, in which case the actual implementation is not relevant for a user of the class `inf::Rational`. Therefore, it is good practice to define the member functions outside of the class declaration, prefixed (within namespace `inf`) with the qualifier `Rational::`. To be consistent with our earlier way of building a library, we should put the class declaration as given above into some `.h` file, while the implementations of the member functions appear in a corresponding `.C` file.

3.2.2 Data members

The variables listed in the class declaration are called *data members* of the class. Each object of the class is physically (i.e. in memory) represented by corresponding variables. In case of the class `inf::Rational`, declaring two data members of type `int` means that each object of the type `inf::Rational` is represented by two `int`'s, which is just what we expect. The syntax for accessing data members is the same as for member functions: `q.n_` is the numerator of the rational number `q`. Within a member function of the class, data members and also member functions can be accessed unqualified (without the '.'), in which case they refer to the implicit parameter the function is called with. For example, when we implement the member function `n()`, we would simply `return n_;` instead of `return (*this).n_;` which would also be allowed but is less compact.

3.2.3 Public and Private

If a member function or data member is declared `private`, it can only be accessed within member functions of the class itself. For example, to the piece of code in the beginning of the chapter that shows the usage of rational numbers, we might add

```
std::cout << "Numerator of q =   " << q.n() << "\n"
          << "Denominator of q = " << q.d_  << std::endl;
```

This will give an error message, because the data member `d_` is `private`; it cannot be called outside a member function of `inf::Rational`, but we are trying to call it from main(). Calling q.n(), on the other hand, is allowed, because the member function n() is public, meaning that there are no restrictions to its use.

Typically, the private section of a class declaration corresponds to functionality the class does not want to present to the outside world. In our case, the member function normalize is of this type. We assume that the user of inf::Rational is interested only in the *value* of a rational number, but not in its *representation*. Because normalize does not change the value, it is not a public method. You could argue that the functions n() and d() should not exist (or at least not be public) for the same reason. On the other hand, you can say that it should be possible to get *some* representation of the value as a pair of integers. In this (and many other cases), the borderline between the public and the private part of a class is not obvious to draw; rather, it depends on the ideas of the class designer, the intended users, and other factors.

Even if we agree to offer public access to the numerator and denominator of a rational number, you could say that the corresponding member functions n() and d() are superfluous: why not make the corresponding *data members* n_ and d_ public, at the same time renaming them to n and d? Then, a user could simply (and more compactly) write q.n instead of q.n().

The problem with this approach is that a user could then also write q.d=0, putting q into an undefined state. You as an educated user would of course never do this, but the point of *controlled* access via the public and private keywords is to prevent less educated users from doing things they should not do. Even you might write q.d=2, not knowing that after this assignment, q could violate the (internal) invariant that numerator and denominator are relatively prime.

We are *only* allowed to make data members public if there are no preconditions on their joint values. For example, if you declare a type inf::complex whose data members are

```
double re; // the real part
double im; // the imaginary part
```

any combination of values of re and im leads to a legal complex number, so it's at least no obvious blunder to make these data members public. It's still a bad idea, though: a complex number is *one* entity with certain functionality, and not just a pair of real numbers—this is exactly the point of building a class representing complex numbers.

Good programming style is therefore to *always* declare data members private and allow controlled acccess (whether it only reads a data member or modifies it) through public member functions.

3.2.4 Constructors

These are special member functions with a special declaration and call syntax. They carry the name of the class, and they have no return type. A constructor call declares and initializes a variable of its respective class. For example, the constructor

```
Rational(const NT& n, const NT& d)
```

```
// PRE: d != 0.
// POST: rational number initialized as n / d.
```

is called when you write `inf::Rational q(1,2);`. This statement defines a variable `q` of type `inf::Rational` and initializes it with the rational number $1/2$. A class can (and typically will) have several constructors, and the regular rules of overloading resolution apply.

Constructors with one parameter play a special role: they are *user-defined conversions*. For example, the constructor

```
Rational(const NT& n);
// POST: rational number initialized as n / 1.
```

is a user-defined conversion from `NT` (in our case `int`) to `inf::Rational`, and this conversion will be considered by the compiler during the process of finding the overloaded function that best matches the actual call. For instance, under the presence of this constructor we may write

```
Rational q(1,2);
q += 1; // q = 3/2
```

In this case, the actual parameter (the `int` value 1) is converted to the type of the formal parameter (`inf::Rational`) of the member function

```
Rational& operator+=(const Rational& x);
```

3.3 Operators

Most operators existing for built-in types in `C++` can be overloaded to work for other types as well, and this is particularly useful in connection with user-defined classes.

For example, we have seen that the (binary) multiplication operator `*` is defined for all built-in number types, but we would like to use it for objects of the class `inf::Rational` as well. There are two different ways of achieving this: we can declare the operator as a one-argument member function of the class `inf::Rational` (as you can see from the class declaration above, this is the variant *not* chosen here), or we can declare it as a two-argument non-member function. In both cases, we can reduce the computation to the member function `*=`. The benefit of operator overloading is the possibility of simply writing `q*r` (*infix* notation). Only this feature makes it possible to reach our initial goal of making our own rational numbers as easy and convenient to use as built-in numbers.

The first variant looks like this (class declaration and implementation in namespace `inf`).

```
// class declaration
class Rational {
    ...
```

```
   Rational operator*(const Rational& x) const;
   // POST: return value is the product of *this and x
   ...
};

// class implementation
Rational Rational::operator*(const Rational& x) const {
   Rational z = *this;
   z *= x;
   z.normalize();
   return z;
}

// main program
int main() {
   ...
   Rational s = q*r; // equivalent to s = q.operator*(r);
   ...
}
```

In this variant, the first argument of the binary operator `*` is an implicit parameter—the object the operator was called for. The second variant is this:

```
// global function in namespace inf
Rational operator*(const Rational& x, const Rational& y) {
   Rational z = x;
   z *= y;
   z.normalize();
   return z;
}

// main program
int main() {
   ...
   Rational s = q*r; // equivalent to s = operator*(q,r);
   ...
}
```

In this second variant, both arguments to the binary operator `*` are explicit parameters. From a user's perspective, there seems to be no difference between the two variants—in both cases, one can write `q*r`, and the effect is the same. Still, the second variant is preferred in practice, because it is symmetric in the two arguments. Then, with the user-defined conversion from `int` to `inf::Rational` at our disposal, both expressions `q+1` and `1+q` are well-defined and equivalent, if `q` is of type `inf::Rational`. In contrast, if `operator*` comes as a member function, the symmetry is broken, because the implicit argument is the first one and the explicit one the second. In this situation, `q+1` will work, but `1+q` won't! The reason is that `1` is not an object of the class `inf::Rational`, so `operator*` cannot be called for it. The user-defined conversion doesn't help here—in order to make use of it, the compiler would first have to find it, by going through all possible classes `X` that have

- an `operator*` with parameter of type `inf::Rational`, and
- a user-defined conversion from `int` to `X`.

Even if in many cases (including ours) only one such class `X` exists, this type of lookup is not supported by `C++`. Member functions of a class `X` can only be called for objects of class `X`.

3.4 Input and Output

As already indicated, input and output of rational numbers are most naturally realized as stream-manipulating global operators `operator>>` (for `std::cin`) and `operator<<` (for `std::cout`). It would also be possible through a member function of `inf::Rational` to tell a rational number how to write itself to a given stream, but then the 'standard' notation

```
std::cout << q; // equivalent to operator<<(std::cout, q)
```

would not be possible, because the implicit argument `q` would have to come first.

3.5 Class Implementation

Here is the implementation of the class `inf::Rational` (in namespace `inf`), along with the global functions for input, output and the arithmetic operations that logically belong to the class. In addition, overloaded operators are available for comparison of two rational numbers.

The only feature not explained so far is the constructor syntax. Most obviously, constructors have no `return` type; beyond that, they have a special section between the signature and the function body, called the *initializer list*. In this list, the data members are set from the constructor's parameters; the task of the constructor's body is then to perform additional computations that are necessary for proper initialization (in our case, normalization).

```
Rational::Rational(const NT& n, const NT& d) : n_(n), d_(d)
{ normalize(); }

Rational::Rational(const NT& n) : n_(n), d_(1) {}

const Rational::NT& Rational::n() const { return n_; }
const Rational::NT& Rational::d() const { return d_; }

void Rational::normalize()
```

```
{
  NT na = n_;
  if (n_ < 0) na = -na;
  NT g = inf::gcd(na, d_);
  n_ /= g;
  d_ /= g;
  if (d_ < 0) {
    d_ = -d_;
    n_ = -n_;
  }
}

Rational& Rational::operator+=(const Rational& x)
{
  n_ = n_ * x.d_ + x.n_ * d_;
  d_ *= x.d_;
  normalize();
  return *this;
}

Rational& Rational::operator-=(const Rational& x)
{
  n_ = n_ * x.d_ - x.n_ * d_;
  d_ *= x.d_;
  normalize();
  return *this;
}

Rational& Rational::operator*=(const Rational& x)
{
  n_ *= x.n_;
  d_ *= x.d_;
  normalize();
  return *this;
}

Rational& Rational::operator/=(const Rational& x)
{
  assert(x.n() != 0);
  n_ *= x.d_;
  d_ *= x.n_;
  normalize();
  return *this;
}
```

```
Rational operator+(const Rational& x, const Rational& y)
// POST: return value is  x + y.
{
  Rational z = x;
  z += y;
  return z;
}

Rational operator-(const Rational& x, const Rational& y)
// POST: return value is x - y.
{
  Rational z = x;
  z -= y;
  return z;
}

Rational operator*(const Rational& x, const Rational& y)
// POST: return value is x * y.
{
  Rational z = x;
  z *= y;
  return z;
}

Rational operator/(const Rational& x, const Rational& y)
// POST: return value is x / y.
{
  Rational z = x;
  z /= y;
  return z;
}

//
// global functions
//

bool operator==(const Rational& x, const Rational& y)
{ return x.n() * y.d() == x.d() * y.n(); }

bool operator!=(const Rational& x, const Rational& y)
{ return !(x == y); }

bool operator<(const Rational& x, const Rational& y)
```

```
{ return x.n() * y.d() < x.d() * y.n(); }

bool operator>(const Rational& x, const Rational& y)
{ return y < x; }

bool operator<=(const Rational& x, const Rational& y)
{ return !(y > x); }

 bool operator>=(const Rational& x, const Rational& y)
{ return !(x < y); }

std::ostream& operator<<(std::ostream& o, const Rational& x)
{ return o << x.n() << "/" << x.d(); }
```

3.6 Pointer Types

Up to now, we have argued that all objects of a given type `T` have some fixed memory size, known to the compiler. Whenever a variable of that type is defined within some scope, or when it is passed to a function as a parameter, the required memory is allocated. After the variable has 'run out of scope', or after the function call has been fully processed, that memory is automatically freed again. All memory managed in that way is kept on the so-called *stack*.

However, we have already met one type whose size is not fixed: the type `vector<T>`, or `vector<int>`, to be concrete. Indeed, adding elements to the vector using its member function `push_back` increases the vector's size at runtime of the program, so the compiler can't possibly know the size beforehand. In order to implement the `push_back` functionality of the type `vector<int>`, there must be some mechanism to get memory for a new variable of type `int`. Most importantly, the lifetime of that variable must be independent from the current scope—we *don't* want its memory to be freed when the call to `push_back` has been processed.

In `C++`, such a *dynamic storage management* mechanism is realized through the concept of `pointers`.

If `T` is any type, a variable of the derived type `T*` has as its value the *address* of an object of type `T` in memory. `T*` is a *pointer type*, and objects of this type are *pointers to* `T`.

Pointers are not a brand-new concept for us. We have already worked with *iterators* to quite some extent, and they behave similarly.

Dereferencing. Pointers can be dereferenced in the same way as iterators: if `p` is of type `T*` and stores some address, then `*p` is the object of type `T` at that address. Here, `*` is a *unary* operator. A precondition for this to work is that an object of type `T` *actually lives at that address*. Unfortunately, pointers are a rather primitive concept (inherited from the `C` language), and they provide no way of checking this precondition. Any programmer working with pointers will at some point see his or her program crash with the error message `segmentation fault`.

This often means that a pointer to some type T was dereferenced, but that nobody (or an object of another type, or even just a part of some larger object) lives at the corresponding address. The simplest program that may evoke such a crash is this:

```
int main() {
   int* p;  // uninitialized pointer
   *p = 5;  // uh-oh, we write to some 'random' address
   return 0;
}
```

The address operator. The operation inverse to dereferencing is the operation of *taking the address*. When we write

```
int i = 5;
int* p = &i;
std::cout << *p; // outputs 5
```

we define p to be a pointer to int which stores the address &i of the variable i. Dereferencing that pointer yields the value of i, in this case 5.

3.6.1 The new statement

Given any type T with constructor T(...), the statement new T(...)

- allocates new memory for a variable of type T,
- initializes this variable by calling the constructor T(...), and
- returns the address of the new variable.

For example, when we write

```
inf::Rational* q_ptr = new inf::Rational(1,2);
```

memory is allocated for a new variable of type inf::Rational, and this variable is initialized with the value $1/2$. The address of the new variable is returned and stored in the pointer variable q_ptr. Even when the pointer variable q_ptr runs out of scope, *the newly allocated memory will remain blocked until it is explicitly deleted.*

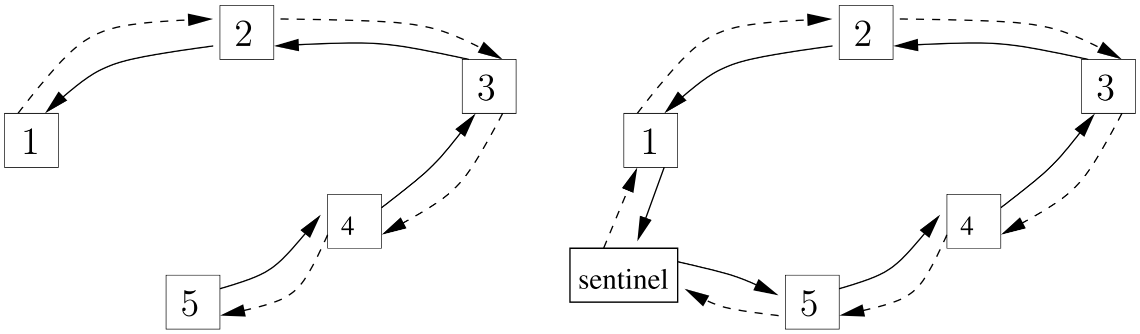
3.6.2 The delete statement

The delete statement is the counterpart to the new statement. If p is a pointer to type T, delete p frees the memory for the object of type T at address p, *under the precondition that memory for this object has been allocated at this address using the new statement.* Furthermore, every piece of memory allocated with new must later be freed with delete. Thus, new and delete always come in matching pairs.

Forgetting to `delete` is a common error, because it passes unnoticed in many cases. The effect is a so-called 'memory leak' which over time could lead to computer memory being full, although most of that memory is not needed anymore. Calling `delete` more than once for the same memory location is another typical error, but this one is more dangerous and leads to corrupt memory structure.

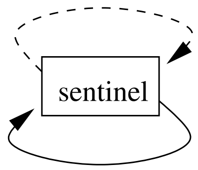
3.7 Example: Lists

We want to demonstrate how to work with pointers using the `list` concept. Like the vector, a list is a data structure for storing sequences of objects, but unlike a vector, it allows efficient insertion and deletion of elements at *any* position in the list (recall that a vector supports only insertion and deletion *at the end*).The idea to achieve this is the following: the elements of the list are no longer at consecutive positions in memory; instead, they may be scattered all over the memory, but each element knows the *address* of its predecessor and its successor in the list. That way, the sequence of elements can be recovered, even though the elements are not sequentially arranged in memory. The picture for this situation is as follows.

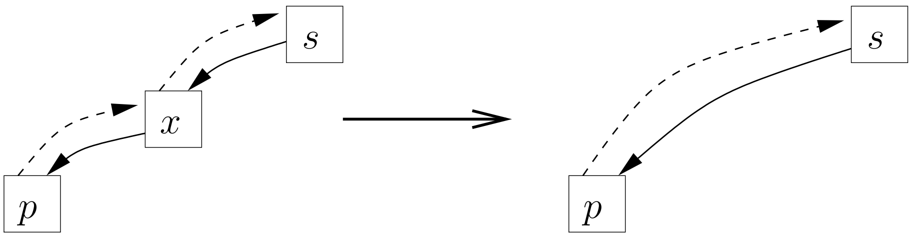


Consider the left picture first: each element has two arrows, one pointing to its predecessor (solid), and one to its successor (dashed). We call this a *doubly-linked list*.

A practical implementation uses a *sentinel* to avoid special cases when dealing with an empty list, or with the first and last elements. The sentinel is an artificial list element whose successor is the first 'real' element of the list, and whose predecessor is the last real element of the list. With the sentinel, an empty list is represented like this:



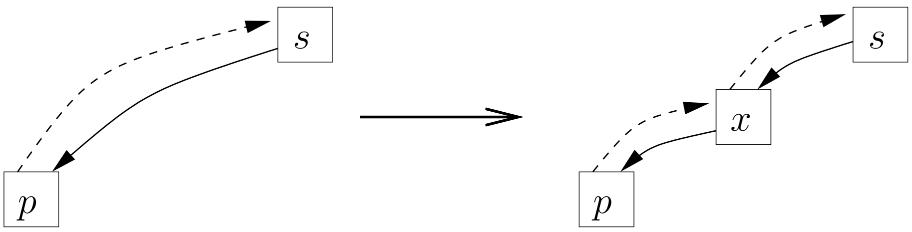
Let's discuss how we can efficiently delete an arbitrary list element x. Assume, p is the predecessor of x and s its successor. Then we want to transform the left into the right situation in the following picture.



To do this, we

- make s the new successor of p,
- make p the new predecessor of s,
- delete x by freeing its memory

Next, we consider insertion of an element x *before* another element s. Let p be the predecessor of s. The insertion then corresponds to the transformation in the following picture, which is exactly inverse to the previous deletion.



To perform this transformation, we

- allocate new memory for x and initialize it with x,
- make p the predecessor of x, and s its successor, and
- make x the new successor of p as well as the new predecessor of s.

It is clear that the work to be done for insertion and deletion does not depend on the number of elements currently stored in the list—it is constant. There is a price to pay, though: unlike the vector, the list does not offer *random access* anymore: we cannot directly 'jump' to the element at position i, but we have to start at the sentinel and walk along the successor arrows until we have found it (we could also walk along the predecessor arrows, but if the element is somewhere in the middle of the list, both variants take long). The following table summarizes the situation and shows that the list is not globally better than the vector, and vice versa. Which data structure to use therefore depends on the application.

operation	vector	list
insertion of elements	only at the end	anywhere
deletion of elements	only at the end	anywhere
time to access elements	constant	proportional to element's position

3.8 Lists in C++: The Basics

Let us see how the above list with its functionality (efficient insertion and deletion) can be realized in C++. It will not come as a surprise to you that we will write a `class List` (as usual, within namespace `inf`), and that we represent successor and predecessor arrows as pointers; but before that, we focus on the building block of the list, namely the *list element* with its predecessor and successor arrows. This building block deserves a class of its own, with the following (preliminary) declaration.

```
class ListElement {
   public:
      int data; // let's assume we want to store int's

      // constructor
      ListElement (int x, ListElement* prev, ListElement* next);
      // POST: initializes the element with x, sets the
      //       predecessor pointer to p and the successor
      //       pointer to s

   private:
      ListElement* prev_; // predecessor pointer
      ListElement* next_; // successor pointer
};
```

We see that the class has three data members, a public one for the `data` item stored, and two private pointer variables, representing the predecessor arrow (`prev_`) and the successor arrow (`next_`). Pointers are just what we need to realize our doubly-linked list. Recall that with every list element, we want to store the address of its predecessor, and the address of its successor (which both are of type `ListElement`). Equivalently, we want to store two pointers to `ListElement`, and this is exactly what we have declared above.

```
class List {
   public:
      void insert (int x, ListElement* s_ptr);
      // PRE: s_ptr points to some list element s
      // POST: x has been inserted into the list before s

      void push_back (int x);
      // POST: x has been appended to the list

      void erase (ListElement* x_ptr);
      // PRE: x_ptr points to some list element x != sentinel
      // POST: x has been deleted from the list
```

```
   private:
      ListElement sentinel;
};
```

In implementing the member functions `insert` and `erase` of the class `List`, we precisely follow the informal procedures given in the previous section. Let us start with the insertion.

3.8.1 Insertion

```
void List::insert (int x, ListElement* s_ptr) {
   // allocate new memory for x and initialize it with x;
   // make p the predecessor of x, and s its successor
   ListElement* x_ptr = new ListElement (x, s_ptr->prev_, s_ptr);

   // make x the new successor of p
   x_ptr->prev_->next_ = x_ptr;

   // make x the new predecessor of s
   x_ptr->next_->prev_ = x_ptr;
}
```

This function contains two new features.

The -> operator. Writing `s_ptr->prev_` is an abbreviation of `(*s_ptr).prev_`. The operation of calling a member function (or—in this case—accessing a data member) of the object at a given address is so frequent that this shortcut has been introduced in order to make code more readable.

In the `insert` function above, `s_ptr->prev_` is therefore the address of p, the predecessor of s. Consequently, the constructor call

```
ListElement (x, s_ptr->prev_, s_ptr)
```

generates a new list element with content x, predecessor p and successor s, as desired.

After executing the `new` statement, the variable `x_ptr` therefore holds the address of the new list element, and the two subsequent statements of `insert` use this address to reset p's successor and s's predecessor to this new element (note that after the constructor call, p and s are accessible through `x_ptr->prev_` and `x_ptr->next_`).

Access to private members of another class. We have said before that member functions and data members declared `private` in some class can only be accessed by member functions of that class. In our implementation of `insert` above, this rule is violated: a member function of the class `List` accesses the private data members `prev_` and `next_` of some

ListElement object. There is a mechanism to make this legal: the class List can be a friend of the class ListElement, and as a friend, it has access to the private members. To implement this, we need to add the line

```
   friend class List;
```

to the public part of the ListElement's declaration. Unlike in real life, this friendship relation is usually not symmetric, and for our purposes it is absolutely not necessary to make the class ListElement a friend of the class List. An alternative to the *friend* mechanism is to declare the functionality in question as public, but this is like sharing one's love-sickness with the whole world.

The push_back member function is easily reduced to insert.

```
void List::push_back (int x) {
   insert (x, &sentinel);
}
```

3.8.2 Deletion

Here is the counterpart to the List's insert method, the erase method. Again, we closely follow the informal procedure given in the previous section.

```
void List::erase (ListElement* x_ptr) {
   // make s the new successor of p
   x_ptr->prev_->next_ = x->next_;

   // make p the new predecessor of s
   x_ptr->next_->prev_ = x->prev_;

   // delete x by freeing its memory
   delete x_ptr;
}
```

3.8.3 Object Lifetime

Although we have talked about object lifetime above, let us try to make this concrete for the case of lists. Recall that there are two mechanisms for defining new variables. The one we are used to is the *automatic* allocation which would take place when we write

```
   ListElement l(x,  s_ptr->prev_, s_ptr);
```

This introduces a variable l of type ListElement which is local to the scope in which it was defined. Outside of this scope, l is not accessible anymore, and the memory occupied by l is automatically freed (to be reused by the system later) when l runs out of scope.

The *dynamic allocation* takes place when we write (as above)

```
ListElement* x_ptr = new ListElement (x, s_ptr->prev_, s_ptr);
```

This introduces a variable of type `ListElement` at the address `x_ptr`, but the difference to the automatic allocation is that this variable exists until its memory is explicitly freed. The pointer variable `x_ptr` that holds the address of the list element has been automatically allocated, so this *pointer* to the list element is not accessible outside the scope in which it was defined. The list element itself persists, but we must make sure that it's address is not forgotten. In the function `insert` above, this is done by putting the address into the successor pointer of p and the predecessor pointer of s, thereby integrating the element into the list.

To further clarify the point, let us consider the following version of insert which performs the same computations as the one above. Instead of allocating x dynamically, we do it automatically, but we still copy its address to the appropriate pointers of s and p.

```
void List::insert (int x, ListElement* s_ptr) {
   // allocate new memory for x and initialize it with x;
   // make p the predecessor of x, and s its successor
   ListElement l(x, s_ptr->prev_, s_ptr);

   // make x the new successor of p
   l.prev_->next_ = &l;

   // make x the new predecessor of s
   l.next_->prev_ = &l;
}
```

The trouble is that in this version, the list becomes corrupt as soon as the call to `insert` has been processed. As a automatically allocated object, `l` runs out of scope at the end of `insert`, and its memory is automatically freed. As a consequence, the address `&l` stored in the successor pointer of s and the predecessor pointer of p is no longer the address of an existing list element. Dereferencing the corresponding pointer leads to a fatal runtime error.

3.8.4 Destructors

As a consequence of the above consideration regarding object lifetime, we must carefully think about who is responsible for explicitly freeing memory allocated with `new`. We must avoid double responsibilities, but we must also not forget to put *someone* in charge. In case of our class `List` above, we haven't done that. Consider the following piece of code.

```
int main () {
   inf::List list;
   list.push_back(5);
}
```

During the call to `list.push_back(5)`, memory has been allocated with `new`, but there is no corresponding `delete` statement. The logical solution would be that these deletions take place when the automatic variable `list` runs out of scope. We haven't mentioned it yet, but when a class variable runs out of scope, naturally all its data members (in this case, just the `sentinel`) run out of scope. This means, the memory for the sentinel is being freed automatically, but *not the memory for the proper list elements*.

In order to handle this situation elegantly, we can implement a *destructor* for the class `List`. This member function is a counterpart to any constructor and is *automatically called* for any object of the class that runs out of scope or is explicitly deleted. The purpose of the destructor is to implement clean-up work beyond the default operation of freeing the memory for the data members.

A destructor is named after its class, but with a leading '˜', and with no arguments (it is called automatically, so there is no way of providing arguments). To the declaration of the class `List` (`public` part), we add

```
~List ();
// POST: memory for all list elements has been freed
```

The implementation is as follows: starting from the successor of the sentinel, we go through the list and `delete` its elements. We stop as soon as the sentinel is reached. In the code below, it is important to remember the successor pointer of an element *before* deleting it, because after deletion, this information is not accessible anymore.

```
List::~List () {
   for (ListElement* p = sentinel.next_; p != &sentinel;) {
      ListElement* s;
      p = p->next_;
      delete s;
   }
   // now the list is empty
}
```

3.9 A full List in C++

3.9.1 Iterators

The primitive list we have implemented above is not usable as is; for example, there is no functionality to iterate over all elements of the list, and there is also no way of obtaining a pointer to a specific list element. Currently, we can only add elements to the list using `push_back`, but later, they are unaccessible. The natural solution is to add *iterators* to the list. The benefit of this is that many iterator-based functions that work for other containers (like `vector<T>`) immediately work for our class `List` as well – we simply need to change the iterator type.

One iterator-based function we have seen before finds the smallest element within a range, and we have used it for vectors. Having a type `List::const_iterator` with the right functionality lets us apply the code to lists as well:

```
typedef List::const_iterator It;
It min_element(It b, It e)
// PRE: [b,e) is a valid range.
// POST: return value is e when the range is empty, otherwise
//   it is the first iterator i in [b,e), for which the
//   following holds: for all j in [b,e): *i <= *j.
{
  if (b == e) return e;
  It m = b++;
  for (; b != e; ++b)
    if (*b < *m) m = b;
  return m;
}
```

To realize list iterators, we define a class `ListConstIterator`, with (part of its) functionality as follows. Essentially, an iterator *is* a pointer, with added functionality. The most interesting new features are the ++ operators (pre- and postincrement, the latter with an artificial `int` argument to distinguish between them—this argument is not provided during the call).

```
class ListConstIterator {
public:
  typedef ListConstIterator This; // just an abbreviation

  // construction from pointer
  ListConstIterator(const ListElement* p) : ptr_(p)
  {}

  // operator== is a friend of mine
  friend bool operator==(const This& x, const This& y);

  // Dereferencing
  const int& operator*() const {
    return ptr_->data;
  }

  // Preincrement
  This& operator++() {
    ptr_ = ptr_->next_; return *this;
  }

  // Postincrement
  This  operator++(int)  {
    This t = *this; ++*this; return t;
```

```
    }

    // class List is a friend of mine
    friend class List;

  private:
    // const pointer; object at that address cannot be modified
    // through this pointer
    const ListElement* ptr_;
  };
```

In order for this to work, the class ListConstIterator needs to be a friend of the ListElement, otherwise, the iterator class could not access the next_ data member of the ListElement.

After adding (to the public part of the declaration of class List)

```
  typedef ListConstIterator  const_iterator;
  const_iterator begin() const {
    return const_iterator(sentinel.next_);
  }
  const_iterator end() const {
    return const_iterator(&sentinel);
  }
```

and writing the global comparison operators

```
bool operator==(const ListConstIterator& x,
                const ListConstIterator& y)
{ return x.ptr_ == y.ptr_; }
bool operator!=(const ListConstIterator& x,
                const ListConstIterator& y)
{ return !(x == y); }
```

the above piece of code for finding the minimum element within a range already works for our lists. A full-fledged implementation of the doubly-linked list iterator concept has more features: there is also a non-const type ListIterator, there are the decrement operators --, and there are additional types required for an iterator to work properly within all the iterator-based algorithms from the standard namespace.

Once we have list iterators, we should also adapt the functions insert and erase to deal with them (for this we need the non-const version ListIterator). In case of insert, this easy modification looks as follows. Observe that the class List is a friend of the ListIterator, so it can access the private data member ptr_ of the iterator object s_it.

```
void List::insert (int x, ListIterator s_it) {
   // allocate new memory for x and initialize it with x;
   // make p the predecessor of x, and s its successor
   ListElement* x_ptr = new ListElement
                           (x, s_it.ptr_->prev_, s_it);

   // make x the new successor of p
   x_ptr->prev_->next_ = x_ptr;

   // make x the new predecessor of s
   x_ptr->next_->prev_ = x_ptr;
```

3.9.2 The Copy Constructor

A frequent operation in a program is that of making a copy of an object. For example, when we write `int i = j;`, the value of `j` is copied to the variable `i`. We should be able to do the same also with objects of a class: in writing `List l2 = l1;` we expect `l1` to become a copy of the list `l2`. Indeed, the compiler will accept this statement, but the effect is not the one we want. By default, the *data members* of `l1` are copied, which is not the same as copying the list: the sentinels of `l1` and `l2` will be different objects, but these are the only objects copied by default; both sentinels have *the same* successor and predecessor pointers, meaning that the lists share their elements. On top of that, `l2` will not even be a well-formed list, because the successor of its last element is still the sentinel of `l1`.

In a real copy, we want to *duplicate* the list elements, and this is what the *copy constructor* is for. In general, if `T` is some type, the (unique) constructor of the form

```
T (const T& t);
```

is the copy constructor, and it is called whenever you write `T t2 = t1`, or equivalently `T t2(t1)`. Not providing this constructor leads to the default behavior described above.

In order to implement this constructor, let's first add a function

```
void append (const_iterator b, const_iterator e);
// PRE: [b,e) is a valid range of list elements
// POST: the range has been appended to *this
```

to the class `List`. The important thing is that this range could be from some *other* list. The implementation is straightforward.

```
void List::append (const_iterator b, const_iterator e) {
   for (; b != e; ++b)
      push_back (*b);
}
```

Given this, the copy constructor looks as follows.

```
List (const List& l)
  : sentinel (0, 0, 0)
{
   sentinel.next_ = &sentinel;
   sentinel.prev_ = &sentinel;
   append (l.begin(), l.end();
}
```

In the initializer list, we call the constructor for the `ListElement` (and this is the only place where we can call it). The first `0` refers to the data item being stored (which is irrelevant for the sentinel). The other two `0`'s are *null pointers*. Setting a pointer to `0` is the common way of initializing it without having an actual object it points to. It is guaranteed that `0` is never the address of an object, and that calling `delete` with a null pointer as argument has no effect.

Within the body of the constuctor, the successor and predecessor pointers of the sentinel are properly initialized to yield an empty list, to which the elements of the list to be copied are then appended.

The copy constructor is also called when a `List` object is passed to a function as a call-by-value argument, and when a function returns an object of type `List`.

3.9.3 The Assignment Operator

When you write

```
int i;
i = j;
```

this is in `C++` terminology an *assignment*. While the copy statement `int i=j` initializes a fresh variable that has no previous value, the assignment `i=j` *replaces* the current value of the variable by another value. When `l1` and `l2` are objects of the class `List`, the statement `l2 = l1;` invokes the *assignment operator* of the class `List`. By default (i.e. if no assignment operator is present), the assignment is again member-wise, i.e. `l2` would as before become a corrupt 'pseudo-copy' of `l1`.

In general, if `T` is some class `T`, the (unique) member function of the type

```
T& operator=(const T& t)
```

is the assignment operator. In principle, it works like the copy constructor, but it has to take care of some extra things:

1. 'Destroy' the old value before replacing it with the new value. This is of particular importance when the old value involves dynamically allocated memory, like in the case of `List`.

2. If `&t = this`, the two objects involved in the assignment are physically the same object, and nothing should happen. This is not only a performance issue: consider the situation in which you write `l=l` for a `List l`. First destroying `l` in order to set it to its new value `l` then doesn't work, because by that time, `l` is already gone.

3. Return the new value. This allows chains of assignments like `l3 = l2 = l1;`

It seems that the destructor of the class might come in handy for the task in 1, but since it cannot explicitly be called, we need a (typically private) member `destroy` which can also be called by the destructor then. In case of the `List`, it doesn't cost anything extra to have a slightly more general variant that destroys a range.

```
void destroy (iterator b, iterator e);
// PRE: [b,e) is a valid range in *this
// POST: *[b,e) has been removed from *this
```

The implementation mimics the destructor. Here, the post-increment operator leads to a compact way of writing the code.

```
void List::destroy (iterator b, iterator e) {
   while (b != e)
      erase((b++).ptr_);
}
```

Finally, we can provide the assignment operator

```
List& operator= (const List& l);
// POST: l has been assigned to *this
```

and its implementation.

```
 List& List::operator= (const List& l) {
    if (&l != this) {
       destroy (begin(), end());
       append (l.begin(), l.end());
    }
    return *this;
}
```

3.9.4 The Default Constructor

The constructor with no arguments (empty argument list) is called the *default constructor*. A class is not required to have a default constructor. For example, the class `inf::Rational` doesn't, because it is not clear what a default rational number is. `0` and `1` are good candidates, of course, but which one we want depends on the context.

If, on the other hand, the class `T` has a default constructor, we can define a variable of the class by simply writing `T t;`, and the default constructor is called to initialize `t`. For example, the class `List` should have a default constructor

```
List();
// POST: creates an empty list
```

with the following implementation.

```
List::List()
   : sentinel (0, 0, 0)
{
   sentinel.next_ = &sentinel;
   sentinel.prev_ = &sentinel;
}
```

A class *without any explicit constructors* has a default constructor by default, simply initializing the data members through *their* default constructors. Also, a data member not appearing in the initializer list of a constructor is default-constructed when the constructor in question is called. Default construction of built-in types (which are *not* classes) is defined to generate unitialized variables (like in `int i;`).

If a class has some explicitly provided constructors, *the default constructor must also explicitly be declared* in order to exist. This can lead to the counterintuitive phenomenon that some code does not compile anymore after adding a constructor to some class. Namely, all of a sudden, that class may no longer have the default constructor needed in some other part of the code.