Programming in C^{++} for Biologists

Course Manual

G. Sander van Doorn, Joke Bakker and Thijs Janzen

University of Groningen
Institute for Evolutionary Life Sciences

Contents

I Elements of the C++ Programming Language **5**

1 Getting started 7
1.1 The skill and art of C++ programming *7*
1.2 Tools for programming *8*
1.3 Compiling, linking and executing in Visual Studio Code *9*
1.4 A first functional program *10*
1.5 Exercises *13*
1.6 Solutions to the exercises *14*

2 Numeric variables and data types 15
2.1 The declaration statement *15*
2.2 Assignment *16*
2.3 Program constants *17*
2.4 Type conversion *18*
2.5 Limited precision, over- and under-flow *20*
2.6 Exercises *23*
2.7 Solutions to the exercises *25*

3 Elementary calculations 27
3.1 Operators *27*
3.2 Building up larger pieces of code *28*
3.3 The library `cmath` *35*
3.4 Exercises *35*
3.5 Solutions to the exercises *36*

4 If..., else..., or what!? 37
4.1 The conditional statement *37*
4.2 Logical expressions *39*
4.3 Exercises *40*
Alternatives to the `if-else`-statement (advanced) *42*
Exercises (advanced) *45*
4.4 Solutions to the exercises *46*

5 Doing stuff many times 49
5.1 The `for`-loop *49*
5.2 The container class `std::vector` *51*
5.3 Exercises *55*
Additional jump statements (advanced) *57*
The `do-while`-statement (advanced) *58*
Exercises (advanced) *59*
5.4 Solutions to the exercises *60*

6 Functions 63
6.1 What functions are good for *63*
6.2 Function definitions *64*
6.3 Exercises *67*
6.4 Solutions to the exercises *68*

7 Matters of style 71
7.1 Naming conventions *71*
7.2 Formatting *72*

7.3 Comments 74
7.4 Functions 74
7.5 Distributing code over multiple files 75

8 Functions continued 77
8.1 Recursive functions 83
8.2 Exercises 83
8.3 Solutions to the exercises 86

9 Working with text 89
9.1 Character constants and variables 89
9.2 Strings 90
9.3 Exercises 92
9.4 Solutions to the exercises 94

10 "Hello disk drive!" 97
10.1 Reading from and writing to a file 97
Stringstream objects (advanced) 100
10.2 Exercises 101
Customising and formatting input and output (advanced) 102
10.3 Solutions to the exercises 104

11 Home-made objects 107
11.1 Pair objects 107
11.2 Structures 109
11.3 A first look at classes 111
11.4 Operating on objects 113
11.5 Exercises 114
Building up complex data types (advanced) 115
Exercises (advanced) 118
11.6 Solutions to the exercises 119

12 Data encapsulation and object design 125
12.1 Introduction 125
12.2 Constructors 128
12.3 Sharing information between class instances: `static` members 131
12.4 Managing access to a class via methods and `friend` functions 132
12.5 Overloading operators 139
12.6 Simulating an iterated prisoner's dilemma tournament 142
12.7 Exercises 145
12.8 Solutions to the exercises 149

13 Procedural programming review 155
13.1 Variables and data types 155
13.2 Statements and program flow 157
13.3 Functions 161
13.4 Compound data types 164
13.5 Input/output 168
13.6 Exercises 172
13.7 Solutions to the exercises 176

14 Exploring the STL 179
14.1 The Standard Template Library (STL) 179
14.2 Templates 180
14.3 STL containers 187
14.4 Exercises 202

15 Designing algorithms 205
15.1 Algorithms 205
15.2 Debugging and error handling 208
15.3 Simulation algorithms for matrix population models and other discrete-time dynamical systems 211
15.4 Exercises 215

15.5 Solutions to the exercises *220*

II Implementing Biological Models in C++ **221**

16 Numerical integration of Ordinary Differential Equations (ODEs) 223
16.1 Ordinary Differential Equations (ODEs) *223*
16.2 Euler's method *224*
16.3 Runge-Kutta integration *226*
16.4 Exercises *227*
Adaptive step-size control (advanced) *230*
Exercises (advanced) *233*
16.5 Solutions to the exercises *235*

17 Stochastic models 245
17.1 Pseudo-random numbers *245*
17.2 The library `random` *247*
17.3 The Gillespie algorithm for stochastic simulations *251*
17.4 Exercises *254*
17.5 Solutions to the exercises *256*

18 Elementary individual-based simulations 259
18.1 Individual-Based Simulation (IBS) *259*
18.2 Exercises *270*

III Appendices **273**

Projects 275

Useful pieces of code 279
Index of code listings *279*
List of exercises *280*

Index 283

Part I:

Elements of the C++ Programming Language

Module 1

Getting started

Jamie explains what's on the menu

In this module you will:

- Produce your first C++program.
- Learn about compiling, linking and running a program from an Integrated Development Environment (IDE).

1.1 The skill and art of C++ programming

C++ (pronounced 'see plus plus') is a general-purpose computer programming language. It was developed in 1979 at Bell Labs by Bjarne Stroustrup as an enhancement to the C programming language (originally, C++ was named 'C with classes'). Since the 1990s, C++ has been one of the most popular commercial programming languages.

Like other programming languages, C++ is a standardised communication technique for expressing instructions to a computer. Computers can perform zillions of calculations in a matter of seconds, but they cannot *think* or *understand* like human conversation partners. For that reason, you must communicate to the computer exactly what you want it to do. You cannot rely on the computer to deduce the precise meaning of an instruction from the context in which it occurs, or to fill in the gaps in your story. Your instructions need to be accurate, unambiguous and complete, and they must obey certain semantic and syntactic rules. Like when you learn a normal language, you will therefore have to learn the rules of the programming language you want to use. In this course, you will learn the basic rules of C++.

Although you might perhaps expect otherwise, learning the rules of a programming language is not a big deal. C++ is far simpler than any natural language: it has a highly structured grammar and only a small vocabulary with a few keywords (such as `const`, `if`, and `void`) and operators (such as `+`, `/` and `*=`). In practice, the real difficulties with programming are encountered before you actually start writing your program code. Before you can put your computer to work on the problem of your interest, you must first think about a strategy to solve the problem, and then translate this strategy to a list of instructions to your computer. Usually, there are several ways to do this. Some of them will be more efficient, elegant or robust than others. The art of choosing the right way to implement your ideas is just as important as the technical skill to write a functional program. For that reason, it takes practice to become a good

programmer. To give you a head start, this manual contains many simplified example programs that are mostly inspired by biological problems. You can use them as building blocks for your own programs.

1.2 Tools for programming

There are four fundamental stages in the creation of a C^{++} program. These stages are editing, compiling, linking and executing the program. For the first three phases you need, respectively, a text-editor, a compiler and a linker. You can use different programs for each of these steps. For example, you could use the simple text-editor Notepad to write your program code and use a command-line C^{++} compiler to compile and link the program. In practice, it is much more convenient to use an Integrated Development Environment (IDE), a software package that combines a text-editor, compiler and linker with other functionalities such as a debugger (a program that helps you find mistakes in your code) and an extended help-system. An example of a widely used IDE is **Visual Studio Code**. This is the one we will use for this course, as it is cross platform.

Know however that there are many other IDEs out there. For instance, if you regularly work on different platforms, Code::Blocks or QT can be useful options: these are open source, cross-platform IDEs that can be installed on Linux, Mac and Windows machines, allowing you to work in the same environment on different platforms.

Other IDEs are platform-dependent. Windows users often prefer Microsoft Visual Studio. For iOS/OSX, you can register (for free) on the Apple Developer website and download Apple XCode (https://developer.apple.com/register/). Unix systems come equipped with the GNU C^{++} compiler, which can be operated via the command line. Many Unix programmers use code-editors such as Emacs or Vim for writing their code, while using Makefiles to interact with the compiler.

To have everyone on the same page, let's install **Visual Studio Code** on your computer, if you haven't already done so.

Getting started in Visual Studio Code

To ensure that we are all working in the same way, we will be using Microsoft Visual Code (VS Code), which can be used on Linux, Mac and Windows. We will here briefly explain how to get started with VS code. You are free however to choose another IDE following your preference. Information on other IDEs can be found on the internet, or you can ask the course assistants for help. VS Code contains many features to design, build, debug and optimise a computer program. We will learn to use some of these features later on in this course, but we will not use any functionality that is geared specifically towards developing Windows applications. By sticking to platform independent coding standards, we ensure that our software will be portable to other operating systems. First install VS Code on your computer (if you have not already done so), following the instructions provided on https://rugtres.github.io/programming4biologists/.

Creating a new project

Having installed VS Code from Brightspace / github, you can also find a template start project in the form of the folder 'hello world'. Please download this folder to your local pc. You can use this folder throughout the course as a starting point for a new project (for each new project, create a new folder can copy the contents of the hello world folder there, being the hidden folder .vscode, the files CMakeLists.txt and main.cpp, do not copy the bin and build folders). Now, from the *File* menu, choose *Open Folder* and select the folder where you placed main.cpp and CMakeLists.txt. VS Code will now ask you whether you trust the authors of the files in this folder, where you can answer 'Yes, I trust the authors'.

Now, VS Code loads the files in the folder, and is ready to start compiling and linking.

Source files are the essential core of any program, since they contain the actual program source code. They have the extension `.cpp` (e.g., `myFirstProgram.cpp`, or `main.cpp`). Header files contain additional code, usually to declare (i.e., preliminarily list without programming any functionality yet) classes and functions used in large, complex programs, and have the extension `.hpp` (e.g., `myComplexProgram.hpp`). The actual program code defining the functionality of these classes and functions is located in a corresponding source file, e.g., `myComplexProgram.cpp`.

Our example folder already has a source file set up, but if we would want to add an additional source file to our project, we can click on the file icon with a large plus, next to the name of our folder (hello world in this case). You can then type the desired file name e.g., `myFirstSourceCode`. The Explorer window on the side now shows a source file `myFirstSourceCode.cpp`, which is opened for editing in the main editor window.

Editing your program code

The first thing you should do to create a working program is to write the `main` function of the program. You can do this by copying the code listed below (or using the main.cpp file provided from GitHub). Copy only the part between the lines, and ignore the line numbers; they are included for explanatory purposes and are not a part of the program code.

Code listing 1.1: `myFirstSourceCode.cpp`

```
1 #include <iostream>
2
3 int main()
4 {
5    std::cout << "Hello CMake" << std::endl;
6    return 0;
7 }
```

Every C++ program should contain a function called `main`. The `main` function is where program execution begins. The `int` that precedes `main` specifies the type of data returned by the `main` function. The keyword `int` stands for integer; integer numbers are one of the built-in data types supported by C++. The brackets following `main` contain the arguments of the `main` function. In this case, the `main` function takes no arguments. The curly braces on line 2 and 4 enclose the body of the function main. At present, it consists of only a single statement (line 3). The statement `return 0;` terminates the `main` function and causes it to return the integer value 0 to the operating system. All of your programs should return 0 when they terminate normally, i.e., without errors.

***Nefertiti* on formatting style**

C++ is a free-form language, in the sense that it does not matter how many instructions you write on a single line and how you use white spaces and tab-characters to indent and format your code. Nevertheless, most programmers adhere to a strict formatting convention, to increase the readability of their code. It is much more difficult to find bugs in messy code than it is to spot mistakes in neatly formatted code. You are free to develop your own style, as long as it is clear and used consistently. Module 7 gives detailed suggestions on how to format your code, which are applied in the programs and program fragments throughout this manual. As you work through the examples, notice particularly how different levels of indentation are used to clarify the structure of a program.

1.3 Compiling, linking and executing in Visual Studio Code

At this stage, you have already created a functional C++ program. Although your program does not yet do anything interesting, you can now compile, link and execute it.

To see what actually happens when you compile and link a program, use Windows Explorer or another directory viewer to view the contents of your project directory (`...\myFirstSolution\myFirstApplication\`). At this point, this directory should contain two files:

- `myFirstSourceCode.cpp`, the file containing your source code,
- `CMakeLists.txt`, a file used by CMake that contains the instructions for building the project,

When you compile and link the program, Visual Studio Code will generate additional folders and files, including an executable program.

Compiling

To compile your code, click on the *Build* button at the bottom of your screen (next to the gear icon), or press F5. Visual Studio Code now displays a message in the output window, showing you whether compilation was successful or not.

During compilation, Visual Studio Code has converted your source code into a language your computer can understand (*machine code*). The compiler can detect several different kinds of errors during the

translation process, and it will give warnings in some situations where it seems likely that you have made a mistake during the editing process. Successful compilation results in an object file. If all went well, your program directory should now contain an additional folder build with many new files, amongst which:

- `Makefile`, a file used by the compiler to build your program
- `cmake-project`, the resulting program

The *Debug* configuration is the default compilation option. Because extra debugging information has to be generated during compilation, this option is somewhat slower than the *Release* configuration that is commonly used when a program is thoroughly tested and debugged. We will learn later on how to create a *Release* build.

Executing

By double-clicking the program icon or the program name in Windows Explorer, you can now execute your program. Alternatively, you can type *run* in the search box of the Windows Start menu, locate the executable, and execute it by clicking the *OK* button. A third and much more convenient way to execute your program is to run it from within the Visual Studio Code environment. To do this, click on the *Play* button at the bottom of your VS Code screen, or activate the same process by pushing the shortcut key `F5`.

1.4 A first functional program

Currently, our program does not do anything except starting up, writing some output to the screen, and terminating. It is time to add some functionality. Code listing 1.2 illustrates how `myFirstSourceCode.cpp` can be transformed into an immensely useful approximate birthday calculator. To experience how it works, enter the additional lines, rebuild your project, and run the executable using `F5`.

Code listing 1.2: The approximate birthday calculator

```
1  #include <iostream>
2
3  int main()
4  {
5     const int currentYear = 2023; //adjust current year, if necessary
6
7     //read age
8     int age;
9     std::cout << "Please enter your age: ";
10    std::cin >> age;
11
12    //calculate approximate date of birth
13    std::cout << "You were born on: 01/01/" << currentYear - age << " +/- 366 days." <<
       std::endl;
14
15    return 0;
16 }
```

The result may be a bit inaccurate still, but this is a good place to start as beginners in C^{++} . Actually, for Master Yoda, the accuracy of the birthday prediction is already above 99.5%. Not bad for a first C^{++} program!

Let's now have a more detailed look at the code. The added statement **`#include`** `<iostream>` on line 1 starts with the # symbol, indicating that it is an instruction to the preprocessor, which prepares for the compilation phase. In particular, an **`#include`** statement instructs the preprocessor to read in function declarations from a library. The name of the library, `iostream`, is written between angle brackets, which indicates that it is located in the default directory for C^{++} libraries. `iostream` is part of the standard C^{++} library and contains the definitions for `cin` and `cout`, which are used to read information from the keyboard and to write information to the console window on the screen.

The statement on line 5 introduces a variable `currentYear` of type **`int`**, that is initialised with the value 2023. The qualifier **`const`** that is used in this definition specifies that the value of the variable is not allowed to be changed after the initialisation. The definition ends with a semicolon, the general

end-of-statement character in C++ (similar to how a period ends a sentence in normal written text). The statement is followed by a comment. Comments have no effect on the program; they are simply cut out of the source code before it is processed by the compiler. Yet, they are extremely important to make your code understandable for others and yourself (for example, when you need to work with code that you wrote a while ago). Single-line comments can be inserted by typing `//` followed by the text of the comment. Everything following on the same line behind the character combination `//` will be ignored by the compiler. In a similar way, the compiler will ignore text that occurs between the character combinations `/*` and `*/`. Hence, you can safely write something like `int /*C++ sucks!*/ main()`, without fearing repercussions. The comment delimiters `/*...*/` also allow you to insert comments that span multiple lines.

***Master Yoda* on variable definitions**

In C++, all variables must be *declared* before they are used. When you declare a variable, you must provide a name for the variable, and specify what type of values the variable can hold (e.g., **`int`** for integer values). From there on, the compiler will recognise the name of the variable when you use it. Unless instructed to do otherwise, the compiler will also reserve the required amount of computer memory for the variable. At that point the variable is *defined*, so that it can be initialised with a value.

***Nefertiti* on variable names**

You can use virtually any label you like to name your variables, and the names you use are totally meaningless to the computer. The only restrictions are that names of variables (or identifiers) (1) must consist of letters, numbers and underscores, (2) cannot start with a number and (3) cannot be identical to one of the reserved C++ keywords (things like **`int`** or **`return`**, which are shown in bold typeface in this manual). I recommend, nevertheless, that you adhere to a consistent naming convention. In particular, use English variable names that are descriptive and accurate, represented in mixed case starting with a lower case letter (C++ is case-sensitive!). Examples are: bodyWeight, sumOfSquares or primeNumber.

The code on lines 7–10 of listing 1.2 defines an integer variable called `age` (line 8). After printing the message `Please enter your age:` to the screen (line 9), the variable `age` is assigned a value that is read from the user's keyboard input (line 10). The so-called I/O-stream objects `std::cin` and `std::cout` provide the interfaces to interacting with the user via the the keyboard and the screen, respectively, using the input and output operators `>>` and `<<`.

Line 13 contains the code that calculates and prints the approximate birthday. Note that three pieces of information are written to the screen, separated in the source code by output operators `<<`. The first component is the string `You were born on: 01/01/`, the second is the result of the calculation `currentYear - age`, and the third is another string (`+/- 366 days.`). The final element in the output statement on line 13, `<< std::endl`, writes a newline character `\n` (line break) and flushes the output stream, causing its contents to appear on the screen. In the end, the output generated by the program looks something like this:

```
Please enter your age: 900
You were born on: 01/01/1116 +/- 366 days.
Press any key to continue . . .
```

***ET* on `std::`**

The names of the I/O-streams `cin` and `cout`, the I/O-stream manipulator `endl`, and many other objects and functions declared in the standard C++ library are members of the **`namespace`** `std` (`std` stands for 'standard'). A namespace is simply a collection of names. The fact that `cout` belongs to **`namespace`** `std` allows you to use the name `cout` for one of your own variables or functions, without losing access to the standard output stream. In such a case, you would use `std::cout` to refer to the cout that is member of the **`namespace`** `std` (i.e., the standard output stream), while you would write `cout` (without the `std::`) to refer to your own variable with that name. Be aware, however, that using the same names for different objects or functions does not necessarily improve the readability of your code.

***Arnold* on the `using` directive**

A drawback of the **`namespace`** `std` is you have to type `std::` every time you use a name from the standard namespace. One way to avoid this, is to insert the statement **`using namespace`** `std;` at the start of your source file. This imports all commands from the **`namespace`** `std` into the global namespace. If you are not brave enough to do that, you can also import individual commands, by inserting **`using`** `std::cout;`, **`using`** `std::endl;`, and so on.

1.5 Exercises

1.1. Hello world. The following program is intended to write the text "Hello world!" to the screen, but it contains two errors. Can you spot them? Check the error messages that are generated by the compiler and use them to locate and correct the errors.

```
int main()
{
    std::cout << "Hello world!" << std::endl
    return 0;
}
```

1.2. Basic calculations in C++. The operators `+`, `-`, `*` and `/` are available in C++ to perform addition, subtraction, multiplication and division operations on numbers. Write a program that reads in two integer numbers, `i` and `j`, from the keyboard and that writes the result of `i + j`, `i - j`, `i * j` and `i / j` to the screen. Do the operators behave as you would expect? In particular, can you explain the behaviour of the division operator? What happens if you divide a number by 0?

1.3. Integer division ($\star$). C++has another arithmetic operator (%) that is used in the context of integer division. Modify the program you wrote in the previous exercise to calculate `i%j` for two integers `i` and `j`. Interpret and explain the result.

Test your skill

1.4. Prime number machine ($\star$). In this exercise, you will create an incredibly useful *prime number machine*, that will allow you generate 40 different prime numbers. Write your program in such a way that it first asks the user to enter a natural number less than forty. Use this number n to generate a prime number $P(n)$ using the formula

$$P(n) = n^2 + n + 41$$

and write the resulting number to the screen. Hint: use the multiplication operator `*` to compute the square of the number n (i.e., n^2 = `n * n`).

1.6 Solutions to the exercises

Exercise 1.1. Hello world. The compiler complains about an unknown identifier `std::` or `std::cout`, because we failed to include the header `iostream`. After correcting that mistake, the compiler warns about a missing semicolon at the end of the first statement in the function `main()`. The corrected code is below:

```
#include <iostream>

int main()
{
    std::cout << "Hello world!" << std::endl;
    return 0;
}
```

Exercise 1.2. Basic calculations in C++.

```
#include <iostream>

int main()
{
    //read in numbers
    std::cout << "Please enter two integer numbers." << std::endl;

    int i;
    std::cout << "i = ";
    std::cin >> i;

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

    //perform calculations and write results to screen
    std::cout << "i + j  = " << i + j << std::endl;
    std::cout << "i - j  = " << i - j << std::endl;
    std::cout << "i * j  = " << i * j << std::endl;
    std::cout << "i / j  = " << i / j << std::endl;

    return 0;
}
```

As illustrated by the example output below, the result of a division of two integer number is an integer number as well. In this example, $5/3 = 1$ because 5 is equal to 1 times 3 plus a remainder of 2. Division by zero causes the program to crash due to a 'run-time error'.

```
Please enter two integer numbers.
i = 5
j = 3
i + j = 8
i - j = 2
i * j = 15
i / j = 1
```

Exercise 1.3. Integer division. The operator `%` is the so-called modulo operator and the expression `i % j` evaluates to the remainder of the integer division `i / j`. For example, `5 % 3 = 2` because 5 is equal to 1 times 3 plus a remainder of 2.

Module 2

Numeric variables and data types

Jamie explains what's on the menu

In this module you will:

- Become familiar with integer- and floating-point arithmetic and the C^{++} data types for representing numerical values.
- Learn how to declare variables and work with program constants
- Be made aware of some limitations caused by the way variables are stored internally, and learn to avoid associated problems.

2.1 The declaration statement

C^{++} supports a number of different kinds of variables, allowing you to work with discrete or continuous values, or with variables that represent a logical value (`true` or `false`) or a character. Before you can use a variable in your code, however, you have to introduce it to the compiler in a declaration statement. This is more than a discretionary sign of politeness, like introducing yourself at a party with other people whom you do not know yet; in C^{++}, declaration statements are needed to provide information about what kind of values a variable can take and how many bytes it will occupy in the computer's memory. This information is contained in the *type* of the variable. All different kinds of variables are stored internally as a sequence of bits. The type of the variable determines how the bits are interpreted and translated to a value. After a variable has been declared, the compiler will also recognise the name of the variable when you use it in expressions.

Two data types that you will use frequently are the type `int`, which you already encountered in module 1, and the type `double`. Additional elementary data types, such as `float`, `bool` and `char`, will be introduced at a later point. Variables of type `int` and `double` both represent numerical values, but there is an important difference between the two:

`int` The type `int` is suitable to implement discrete variables such as counters or index variables. The value of an integer variable is a positive or negative whole number, such as 8 or -528843.

`double` The type `double` emulates continuous variables which are represented as floating point numbers. For example, a variable of type `double` may hold the (approximate) value of the mathematical constant π, `3.141592`, or very small or very large numbers like $-2.61 \cdot 10^{-16}$ or $8.29 \cdot 10^{25}$.

The syntax of a declaration statement is as follows: first comes the name of a C++ data type, followed by the new variable's name (its *identifier*) or multiple identifiers, separated from each other by comma's. The declaration statement is terminated by a semicolon. For example,

```
double mutation_rate;
```
and
```
int population_size, number_of_chromosomes;
```
are two valid declaration statements. The declaration of a variable does not yet give it an initial value. In order to initialise a variable at the time of its declaration simply insert a = behind the name of the variable, followed by its initial value, or insert the initial value between brackets behind the variable's name. For example

```
int number_of_chromosomes = 46;
```
and
```
int number_of_chromosomes (46);
```
are equivalent declaration/initialisation statements.

The type of a variable can be modified by various type modifiers, such as **const**, which specifies that the value of a variable cannot be changed. Type modifiers apply to all variables in the same declaration statement. For example, after the declarations

```
const int i = 2, j = 1; int k = 3;
```
`i` and `j` are **const** but `k` is not. Variables that are declared to be **const** *must* be initialised at their declaration. For non-**const** variables, the combination of declaration and initialisation is optional. However, it is important to explicitly assign value to a variable before it is used in a calculation. According to the C++ standard, you cannot rely on the assumption that uninitialised variables have a specific default value (such as 0)! This, and the fact that your program will run slightly faster when you declare variables as late as possible, leads to the recommendation that the declaration and initialisation of a variable should be combined at the point where the variable is first needed.

2.2 Assignment

As you have seen in module 1, input read from `std::cin` (which is the keyboard by default) can be used to set the value of a variable. There are also other ways to modify the value of a variable after its declaration. One common way to achieve this is to use the assignment operator =. Its use is illustrated in the following program, which calculates the molecular weight of the stimulant drug 1,3,7-trimethylpurine-2,6-dione, an essential resource for creative computer programming. The pharmacological data necessary for the calculation is provided by our biological expert, Kermit the Frog:

***Kermit the Frog* on 1,3,7-trimethylpurine-2,6-dione**

1,3,7-Trimethylpurine-2,6-dione, also known as caffeine, is a central nervous system stimulant frequently used by computer programmers to boost their performance. According to its molecular formula $C_8H_{10}N_4O_2$, one molecule contains eight carbon atoms (atomic weight 12.011 g/mol), ten hydrogen atoms (atomic weight 1.008 g/mol), four nitrogen atoms (14.007 g/mol), and two oxygen atoms (atomic weight 15.999 g/mol).

Code listing 2.1: The molecular weight of 1,3,7-trimethylpurine-2,6-dione

```
1  #include <iostream>
2
3  int main()
4  {
5      // declaration of variables
6      double weight_caffeine;
7      const double    weight_c = 12.011, weight_h =  1.008,
8                      weight_n = 14.007, weight_o = 15.999;
9
10     // calculation of molecular weight of
11     // 1,3,7-trimethylpurine-2,6-dione (C8H10N4O2)
12    weight_caffeine =
13                     8 * weight_c + 10 * weight_h +
```

```
14                    4 * weight_n +  2 * weight_o;
15
16    // generate output
17    std::cout   << "1,3,7-Trimethylpurine-2,6-dione (caffeine) weighs "
18                << weight_caffeine << " g/mol."
19                << std::endl;
20
21    // terminate program
22    return 0;
23 }
```

The program in listing 2.1 consists of three parts. In the first part (lines 6-8) we declare five variables of the type **double**. The first one of them (`weight_caffeine`) is not initialised, the others (`weight_c`, `weight_h`, and so on) are initialised upon their declaration with the appropriate values for the atomic weights. To indicate that these values will not be changed in the rest of the program, the corresponding variables are declared to be **const**. In the second part of the program (lines 12-14), we compute the molecular weight of caffeine. The procedure is simple: just add the total weights of the different kinds of atoms that occur in the drug, and assign the resulting value to the variable `dWeightCaffeine` using the assignment operator (`=`). The final part of the program (line 17-19) generates output to the screen reporting the result of the calculation to the user.

An expression like `x = y`, should be read as 'assign the value of `y` to the variable `x`' or 'change the value of `x` to the value of `y`'. Note that an assignment `x = y` in C^{++} is different from a mathematical equation $x = y$. Something like $x + 1 = y$, which is a perfect mathematical equation, makes no sense as a C^{++} expression `x + 1 = y`, because `x + 1` is not a single variable you can assign a value to (the compiler will complain that `x + 1` is not an *lvalue*, i.e., an object that persists after the evaluation of an expression). Conversely, `x = x + 1` is a valid C^{++} assignment (it simply instructs the computer to increase the value of the variable `x` by 1), but $x = x + 1$ has no solution when interpreted as a mathematical equation.

A second distinction to be aware of is that there is a subtle difference between the use of `=` on line 7-8 of listing 2.1 versus its use on line 12. In the first case, the operator `=` appears in a *declaration statement*. Such a statement is clearly recognisable by the fact that it starts with a type name (e.g., **int** or **const double**). Here, the `=` sign is used to provide an initial value for a newly created variable. This can occur only once, since a variable cannot be declared multiple times. On line 12 of listing 2.1, `=` appears in an *assignment*. Here, the value of an already existing (non-**const**) variable is modified, which can occur arbitrarily often.

2.3 Program constants

Apart from variables, a typical C^{++} program will also contain so-called literals, fixed values that are known at compile time and that are not altered by the program. Also here the distinction between various types is important. For example, the expression `12.011`, which provides an initial value for `dWeightC` on line 7 of listing 2.1, is interpreted as a constant of type **double**, whereas the coefficients `8`, `10`, `4`, and `2` on lines 13-14 are interpreted as integer constants by the compiler. Finally, `"1,3,7-Trimethylpurine-2,6-dione (caffeine) weighs "` and `" g/mol."` on line 17 and 18, respectively, are literals that are interpreted as two strings of characters of type **char** (you will learn more about this data type in module 9). String literals are easily recognisable by the string delimiters `"..."`. In a slightly more subtle way, the compiler also carefully distinguishes between constants of type **int** and **double**. It does so by interpreting numeric constants according to the following rules

int Literals of type **int** can be represented in three different notations: decimal (base 10), octal (base 8) and hexadecimal (base 16). Integer literals in decimal notation contain exclusively the numbers 0 to 9, and do not start with a zero. For example, the declaration statement **int** `i = 1000;` initialises an integer variable `i` with the value 1000. Integer literals starting with zero, followed by other numbers from 0 to 7, are interpreted as numbers in octal notation. For example, the declaration statement **int** `i = 01750;` initialises the integer variable `i` with the decimal value 1000 as well, because 1000 is written as 1750 in octal notation ($1000 = 1 \cdot 8^3 + 7 \cdot 8^2 + 5 \cdot 8^1 + 0 \cdot 8^0$). Finally, an integer literal that starts with 0x (zero-x), is interpreted as a number in hexadecimal notation. Hence, instead of **int** `i = 1000;` or **int** `i = 01750;`, we may also write **int** `i = 0x3E8;`, given that $1000 = 3 \cdot 16^2 + 14 \cdot 16^1 + 8 \cdot 16^0$. Note that the E in the above declaration stands for 14. Similarly, A, B, C, D, and F (or a, b, c, d, f) represent the decimal values 10, 11, 12, 13 and 15, respectively. Octal notation is rarely encountered in program code, but you may occasionally see numbers written in hexadecimal notation, which are frequently used to represent memory

addresses.

double Literals of the type **double** always contain a decimal point, and are written in standard or scientific notation. For example, **double** dDouble = 1000; and **double** dDouble = 1.0e3; are equivalent declaration statements.

2.4 Type conversion

An aspect of C++ that can be a bit confusing to novice programmers is that some functions and operators may work on variables of one type but not another, or that they may give different results depending on the type of variable they are applied to. This applies, for example, to the modulo operator (%; see exercise 1.3) which cannot be applied to **double**s, or to the division operator (/), which implements either integer division or floating-point division, depending on the type of its operands. Consider, for example, the following statements

```
1 const int int_div = 1 / 2;
2 const double double_div = 1.0 / 2.0;
3
4 std::cout << "int_div = "     << int_div    << std::endl;
5 std::cout << "double_div = " << double_div << std::endl;
```

and the corresponding console output

```
iInt = 0
dDouble = 0.5
```

Here, the result of the calculation is different, because the division operation on line 1 is implemented as an integer division (which discards the fractional part, leading to an integer result), whereas the division operation on line 2 implements floating-point division in which the fractional part is retained. Which one of the two implementations is used depends on the type of the two operands. If both are integers, the division is executed as an integer division; otherwise, floating-point arithmetic is used.

***ET* on integer division**

An annoying problem with integer division is that different compilers have different opinions on the value of `i / j` and `i % j` when either one or both of the integers `i` or `j` are negative, so it is better to avoid integer division with negative numbers.

In expressions that contain elements of different type, the compiler will attempt to convert the different types in such a way that the expression can be evaluated with no, or a minimal loss of precision. This process is referred to as implicit type conversion. As an example, consider the sum on line 13-14 of listing 2.1, `8 * weight_c + 10 * weight_h + 4 * weight_n + 2 * weight_o`, which contains 4 variables of type **double** and 4 literals of type **int**. In the evaluation of this expression, the multiplications are done first, in accordance with the rule that multiplication precedes addition. Each multiplication is performed using floating-point arithmetic, after first converting the integer literal to a **double** value. This way, the calculation is much more precise than when the variables would have been rounded off to the nearest integer value before executing the multiplication using integer arithmetic.

The precise rules for implicit type conversion used by the compiler are rather intricate, but designed to maximise the accuracy of calculations. Therefore, you can generally rely on the compiler's default way of dealing with variables without worrying too much about the type conversions that are happening behind the scenes. In some cases, however, it may be necessary to change the type of an expression before performing a calculation to avoid unintended consequences of implicit type conversion. Suppose, for example, that you need to calculate the sex ratio in a population (expressed as percentage males) given a certain number of males and females provided by the user. A first-time programmer may approach the problem as follows:

Code listing 2.2: An integer-division mistake

```
1  /* This program calculates the sex ratio expressed as percentage males.
2     Written on Friday January 13th by A. Novice. */
3
4  #include <iostream>
5
6  int main()
7  {
8      // read in numbers of males and females
9      int num_males, num_females;
10     std::cout << "Enter the number of males" << std::endl;
11     std::cin  >> num_males;
12     std::cout << "Enter the number of females" << std::endl;
13     std::cin  >> num_females;
14
15     // calculate sex ratio
16     const int pop_size = num_males + num_females;
17     const double male_fraction = num_males / pop_size;
18
19     // generate output
20     std::cout   << 100.0 * male_fraction
21                 << "% of the population consists of males" << std::endl;
22
23     // terminate program
24     return 0;
25 }
```

For any strictly positive number of males and females entered by the user, the output of this program is always the same:

```
0% of the population consists of males
```

The reason for the erroneous result is that the division on line 17 of the listing involves two integers. Therefore, the division is executed as an integer division, which evaluates to the integer value 0, given that the number in the denominator is larger than the one in the numerator. To avoid this problem it suffices to convert one of the operands to a double, which will induce the compiler to implement floating-point division. There are several ways of doing this. One solution is to replace `pop_size` by a double:

```
const double pop_size = num_females;
const double male_fraction = num_males / pop_size;
```

This way, the integer value `num_males + num_females` will be used to initialise a variable of type **`double`** (this step involves an implicit conversion), which is subsequently used in the calculation of the sex ratio.

Another solution is to calculate the percentage of males in one go, in the following way:

```
// calculate sex ratio
const int pop_size = num_males + num_females;
const double sex_ratio = 100.0 * num_males / pop_size;

// generate output
std::cout << sex_ratio << "% of the population consists of males" << std::endl;
```

This solution relies on the fact that the operators $*$ and / have the same precedence, such that the compiler first evaluates `100.0 * num_males` and then uses the result of this expression to evaluate the division (more about operator precedence rules will follow in section 3.2). The result of the multiplication, which involves an integer variable (`num_males`) and a floating point literal (`100.0`), is internally represented as a **`double`**, forcing the subsequent division to follow floating-point arithmetic.

A third solution is to explicitly force a type conversion by applying a so-called cast operation.

```
const int pop_size = num_males + num_females;
const double male_fraction = static_cast<double>(num_males) / pop_size;
```

Here, the value of `num_males` is explicitly converted to a value of type **`double`** before it is used in the division operation. Again, the result is that an integer division is prevented because only one of the operands is of integer type.

An advantage of explicit type conversions using **`static_cast`**`<...>(...)` is that it makes very clear that an expression is converted to some other type, whereas alternative solutions are more cryptic about the implicit type conversion that is taking place. However, explicit type casts looks rather ugly. This is no coincidence: by choosing a complicated notation, Bjarne Stroustrup, the developer of C++, wished to point out that you should try to avoid type casts altogether and choose alternative solutions that do not require type conversion. In fact, in many cases, type conversions can be avoided relatively easily, and it is a sign of a proper programming style to do so. For example, rather than writing **`double`** `param1 = 0`, it is better to write **`double`** `param1 = 0.0`, in order to avoid an implicit conversion from **`int`** to **`double`**.

2.5 Limited precision, over- and under-flow

When a variable is declared, the compiler reserves a part of the computer's memory to store the value of the variable. The amount of memory that the compiler will reserve is determined by the variable's type, the compiler used, and the specific the machine used, but not by its value! Typically, storing an **`int`** takes four bytes of memory space. Four bytes (32 bits) can represent at most $2^{32} = 4294967296$ different values. Although this is a large number, it is not infinitely large, implying that a variable of type **`int`** can only represent a limited number of values. In fact, most machines represent variables of the type **`int`** in such a way that they can represent zero, the first $2^{31} - 1$ strictly positive numbers, and the first 2^{31} strictly negative numbers.

Jamie's recipes for disaster: **integer overflow**

What happens if you go beyond the limits of the type **`int`** is different for different compilers and different machines, but in all cases overflow (attempting to store a value that exceeds the range of a datatype) is something you want to avoid! A dramatic example of what can go wrong is provided by the crash of the maiden flight of the Ariane 5 rocket, which was ultimately caused by an overflow error in the engine steering software.

The following program illustrates what can happen if you attempt to create an integer value that exceeds the capacity of an **`int`**

Code listing 2.3: Integer overflow

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

int main()
{
    int largest_int = std::numeric_limits<int>::max(); // largest possible value available on your
    machine, typically 2^31 - 1

    std::cout << "largest integer     = " << largest_int     << std::endl;
    std::cout << "largest integer + 1 = " << largest_int + 1 << std::endl;

    return 0;
}
```

The program produces the following output (or something equally weird, depending on which platform you are using):

```
iLargest       = 2147483647
iLargest + 1 = -2147483648
```

Although this result makes sense if you consider how integers are represented in the computer's memory, it is clearly not in line with what the programmer intended when he wrote the expression `largest_int + 1`. If you run the above program on your computer, you will also note that the compiler will not warn you if you accidentally work with values that cannot be represented by an **`int`**.

Interestingly, this behaviour changes completely when you switch to an unsigned int Unsigned integers do not reserve a bit to store the sign of the number (e.g. whether it is negative or not). When we exceed the maximum range of an unsigned int, the number rolls over to the start of the range, e.g. 0:

Code listing 2.4: Unsigned Integer overflow

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

int main()
{
   unsigned int largest_uint = std::numeric_limits<unsigned int>::max(); // largest possible value available on your machine, typically 2^31 - 1

   std::cout << "largest unsigned integer     = " << largest_uint     << std::endl;
   std::cout << "largest unsigned integer + 1 = " << largest_uint + 1 << std::endl;

   return 0;
}
```

This generates the following output:

```
largest unsigned integer     = 4294967295
largest unsigned integer + 1 = 0
```

Which, coincidentically is exactly twice the maximum we found for the normal **int**, indeed the byte indicating the sign of the number has been used to extend the range. Yet, $2^{31} - 1$ may still limit your use of a number. Generally, C++ programmers tend to use the type `size_t`, which the compiler translates into the maximum size integer number available on the machine, but assuring that the number is not signed. This can often be much larger than an **unsigned int**. Try it out yourself by substituting `size_t` into the code of section 2.4!.

Floating-point variables

The internal representation of floating-point variables is somewhat more complicated than that of integer variables. We will not go into the details here. It is important to know, however, that also floating-point variables have a fixed memory size. This implies that floating-point variables have a limited precision. This limitation can cause problems, particularly when you add or subtract floating-point values that differ by several orders of magnitude.

The name of the datatype **double** derives from the fact that it allows for double-precision floating-point arithmetic. Consequently, storing variables of the type **double** requires quite a lot of memory space. However, modern machines typically have a huge memory capacity, so memory usage is not of primary concern for most applications. When memory usage is an issue and high precision is not required, you can fall back on the data type **float**, which usually requires less memory space. The official C++ standard only requires that the size of a **double** (in bytes), is at least as large as the size of a **float**, but, in fact, most compilers reserve 4 bytes to represent a **float**, and 8 bytes to represent a **double**. The precision of the type **double** is therefore usually larger than that of the type **float**. Here is an example program to illustrate this point

Code listing 2.5: A floating-point underflow error

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

int main()
{
    double double_val = 1.0e6;
    float  float_val        = 1.0e6f;  // literals of type float are written as
                                       // a floating point number in standard or scientific
                                       // notation followed by the character f

    // add a large number to both variables
    const double large_number = 100.0;
```

```
13     double_val = double_val + large_number;
14     float_val  = float_val  + large_number;
15     std::cout << "double_val - float_val = " << double_val - float_val << std::endl;
16
17     //add a small number to both variables
18     const double small_number = 0.01;
19     double_val = double_val + small_number;
20     float_val  = float_val  + small_number;
21      std::cout << "double_val - float_val = " << double_val - float_val << std::endl;
22
23     return 0;
24 }
```

The output generated by this program is given by

```
double_val - float_val =  0
double_val - float_val = 0.01
```

Apparently, adding one hundred to one million (lines 12-13) gives the same result irrespective of whether this operation involves single- (**float**) or double-precision (**double**) floating-point arithmetic. The story is different when we add one hundredth to (slightly more than) one million (line 18-19). For this operation, single-precision floating-point arithmetic is insufficiently precise. At line 18, the **double** variable `double_val` increases as was intended, but the **float** variable `float_val` remains unchanged after the statement at line 19, resulting in a unintended difference between the values of the two variables. An error like this, where the relative change of a floating point variable is smaller than the precision of the data type is called an underflow error. We can prevent this error from happening by assuring that the value we add to our focal floating point value is never smaller than the associated accuracy of the floating point value. We can acess that associated accuracy with the function `std::numeric_limits<float>::epsilon()` for **float**s and `std::numeric_limits<double>::epsilon()` for **double**s. Notice that this accuracy scales with the value used!

Code listing 2.6: Demonstrating maximimal accuracy

```
1    #include <iostream>
2    #include <limits>
3
4    int main() {
5       double double_val = 1.0e6;
6       float float_val   = 1.0e6f;
7
8       std::cout << "double accuracy of 1e6 = " << std::numeric_limits<double>::epsilon() *
      double_val << std::endl;
9       std::cout << "float accuracy of 1e-6 = "  << std::numeric_limits<float>::epsilon()  *
      float_val  << std::endl;
10
11
12      double double_val2 = 1.0e-6;
13      float float_val2   = 1.0e-6f;
14
15      std::cout << "double accuracy of 1e-6 = " << std::numeric_limits<double>::epsilon() *
      double_val2 << std::endl;
16      std::cout << "float accuracy of 1e-6 = "  << std::numeric_limits<float>::epsilon()  *
      float_val2  << std::endl;
17
18      return 0;
19   }
```

The output generated by this program is given by

```
double accuracy of 1e6 = 2.22045e-10
float accuracy  of 1e6 = 0.119209
double accuracy of 1e-6 = 2.22045e-22
float accuracy  of 1e-6 = 1.19209e-13
```

2.6 Exercises

2.1. Integer versus floating-point arithmetic. What is the value of the integer variables `i` and `j` after the following statements?

(a) `i = 2 / 5 * 3; j = 3 * 2 / 5;`

(b) `i = 2.0 / 5 * 3; j = 2 / 5 * 3.0;`

The evaluation of the expressions in (b) involves implicit type conversion.

(c) Rewrite the expressions such that all type conversions are explicit.

2.2. Floating point accuracy. Marco is performing a number of operations. Determine which of these operations can be safely executed using **`float`**s, and which require to use the **`double`** datatype:

(a) `float num1 = 1e9; float num2 = 100; float num3 = num1 + num2;`

(b) `float num1 = 52378; float num2 = 0.42; float num3 = num1 + num2;`

(c) `float num1 = 7;l float num2 = 275960397; float num3 = num1 + num2;`

2.3. Implicit type conversion I. When literals and variables of different types are mixed in expressions, they are converted to the same type. For example, in the expression `dDouble * 2.1f`, where `dDouble` is declared as a **`double`**, both operands of the multiplication operator are converted to the same type, which is, at the same time, the type of the result of the multiplication operation. One can think of different rules for type conversion in expressions. For example, one could imagine that operands in an expression are converted to the type of the first value or literal in the expression. Alternative options are type promotion (always convert to the largest type), or type degradation (always convert to the smallest type). Moreover, in complicated expressions, types could be converted at one go for the whole expression, or on an operation-by-operation basis. Try to figure out what rules for type conversion are implemented in C++. To do this, first complete the following table. Write small programs to check the type of the operands and results of simple expressions, e.g., `dDouble * fFloat`. Then consider the more complicated expressions in exercise 2.1) to determine whether type conversion occurs at one go for the whole expression, or on an operation-by-operation basis. Summarise your findings in a few general rules. You can display the type of an operand using **`typeid`**`(dDouble).name()`. Use this operand smartly to fill out the following table:

1st Operand		2nd Operand		Result	
Type	**`typeid`**`().name()`	**Type**	**`typeid`**`().name())`	**Type**	**`typeid`**`().name()`
`double`		**`float`**			
`int`		**`float`**			
`double`		**`int`**			
`int`		**`double`**			

2.4. Implicit type conversion II. When an expression is converted from one type to the other, not only the type, but also the value may change. Convert between **`int`** and **`double`** using explicit type-casts, and try to figure out the rules that C++ uses to convert integral to floating-point values and vice versa.

2.5. The Quetelet index. The Quetelet, or Body Mass Index (BMI), is used in medicine as a normalised measure of tissue mass in an individual, allowing that person to be classified as underweight, normal weight or obese. The Quetelet index Q is defined as the body mass (M, measured in kilograms) divided by the square of the body height (h, measured in meters), i.e., $Q = M/h^2$. Write a program that asks the user to enter weight and height via the keyboard. After reading in those values, the program should calculate the Quetelet index and report the result back to the user. You should start worrying about your weight, or about bugs in your program, if the result does not fall in the range 18.0-30.0.

Test your skill

2.6. MasterMix ($\star$). A researcher is preparing a Mastermix containing multiple primers to be used in a Polymerase Chain Reaction (PCR). After combining the different primers, she has 103 ml of MasterMix prepared. Per sample she needs to pipet 7mL of the Mastermix solution. Write code to calculate how many samples she can pipet from 103 ml of Mastermix. You will notice that after pipetting the maximum number of samples, there is some Mastermix remaining. Write code that calculates how much ml MasterMix remains after pipetting the maximum number of samples. Mastermix is quite expensive, how much more Mastermix should she prepare in order to have no remainder? Can you add this calculation to your code?

2.7 Solutions to the exercises

Exercise 2.1. Integer versus floating-point arithmetic.

(a) Since division and multiplication have the same precedence, the expressions are evaluated from left to right. Therefore, `i = (2 / 5) * 3 = 0 * 3 = 0` and `j = (3 * 2) / 5 = 6 / 5 = 1`. Note that the divisions are executed as integer divisions, because all operands are of integral type.

(b) These expressions contain literals of type **double**, so some of the evaluations are executed in floating-point arithmetic: `i = (2.0 / 5) * 3 = 0.4 * 3 = 1.2` so that `i` takes the value 1 after the implicit type conversion of 1.2 to an integer value. For the second expression, `j = (2 / 5) * 3.0 = 0 * 3.0 = 0.0` which is converted to the integer value 0. Here, the first division is executed as an integer division.

(c) The equivalent statements with explicit type conversion are:

`i = static_cast<int>(2.0 / static_cast<double>(5) * static_cast<double>(3))`

and `j = static_cast<int>(static_cast<double>(2 / 5) * 3.0)`.

Exercise 2.2. Floating point accuracy.

(a) The numeric limit of 1e9 is 119. 100 is smaller than tat, so **double** should be preferred.

(b) The numeric limit of 52378 is 0.006. 0.42 is larger than that, using **float** is safe.

(c) The numeric limit of 275960397 is 32. 7 is smaller than that, so **double** should be preferred.

Exercise 2.3. Implicit type conversion I.

1st Operand		2nd Operand		Result	
Type	`typeid().name()`	**Type**	`typeid().name()`	**Type**	`typeid().name()`
double	d	**float**	f	**double**	d
int	i	**float**	f	**float**	f
double	d	**int**	i	**double**	d
int	i	**double**	d	**double**	d

The rules for type conversion in expressions are:

- integral and floating point promotion: **float** values are automatically elevated to **double**
- arithmetic conversion: if the operands of an operator do not match in type after type promotion, the compiler finds the highest priority operand and converts the other operand to match. Here, the priority of operands is: **double** (highest), **float**, **int**.
- type conversion is performed on an operation-by-operation basis.

Exercise 2.4. Implicit type conversion II. During conversions from floating-point to integral types, floating point values are rounded of to the closest integral value between zero and the floating point value. For example, `static_cast<int>(6.8)` evaluates to `6`, `static_cast<int>(-6.8)` evaluates to `-6`. Converting the other way occurs as you would expect: e.g., `static_cast<double>(6)` evaluates to `6.0`.

Exercise 2.5. The Quetelet index.

```
1  /* This program calculates the Quetelet index
2     based on body weight (in kg) and height (in m)
3     provided by the user.
4     Written on 24--08--2016 by Master Yoda. */
5
6  #include <iostream>
7
8  int main()
9  {
10    // read weight and height
11    std::cout << "Please enter your weight (in kg)" << std::endl;
```

```
12      double weight;
13      std::cin >> weight;
14
15      std::cout << "Please enter your height (in m)" << std::endl;
16      double height;
17      std::cin >> height;
18
19      // calculation of the Quetelet index
20      const double quetelet_index = weight / (height * height); //use brackets to control
21                                                                //which part of an expression
22                                                                //is evaluated first
23      // generate output
24      std::cout   << "Your Quetelet index is "
25                  << quetelet_index << " kg/m^2." << std::endl;
26
27      // terminate program
28      return 0;
29  }
```

Module 3

Elementary calculations

Jamie explains what's on the menu

In this module you will:

- Become familiar with many of the C++ operators.
- Learn how to build up larger pieces of code using expressions, statements and compound statements.
- Be introduced to the mathematical functions in the `cmath` library.

3.1 Operators

Besides the standard arithmetic operators `+`, `-`, `*` and `/`, the modulo operator `%` and the assignment operator `=`, C++ has many additional operators that can be used in calculations. This section introduces most of them and discusses the rules for combining them in larger expressions.

Compound assignment

C++ programmers commonly use a kind of shorthand notation that combines an assignment with an arithmetic operation. This is made possible by the so-called compound assignment operators `+=`, `-=`, `*=`, `/=` and `*=` (additional compound assignment operators exist for logical operators, which are introduced in Module 4). For example, the expression `myAge += 1` is equivalent to `myAge = myAge + 1`, meaning that the statement `myAge += 1;` will cause the value of `myAge` to be increased by one. In a similar way, the statement

```
myMoney *= 1.0 + interestRate;
```

instructs the compiler to increase the value of `myMoney` by a factor `1.0 + interestRate`, exactly as if we had written

```
myMoney = myMoney * (1.0 + interestRate);
```

`myAge += 1` is not only shorter to write, it also used to execute faster than `myAge = myAge + 1` on old-fashioned processors. Nowadays, clever optimisation by the compiler will typically eliminate the difference in execution time, but using compound assignment operators is still considered *best practice* by the C++ programming community. For experienced programmers, compound assignment is easier to write and more intuitive.

There are special operators for incrementing or decrementing an integral value by one, which allow some common compound assignment expressions to be abbreviated even further. In particular, instead of writing `myAge += 1`, an experienced C++ programmer would probably write `++myAge` or maybe `myAge++`. Similarly, rather than writing

```
daysToNextBirthday -= 1; or even
daysToNextBirthday = daysToNextBirthday - 1;
```

an experienced programmer would write

```
--daysToNextBirthday; or daysToNextBirthday--;
```

The operators `++` and `--` appear frequently in iterative statements (see Module 5), which occur in almost every C++ program. They are examples of unary operators, i.e., operators that have a single operand. Most other operators are binary (these require two operands), and there is one C++ operator, the conditional operator (`?:`, introduced in Module 4), that requires three.

As indicated in the above examples, `++` and `--` may occur before or after the variable that is modified. In the old days, there used to be a subtle difference between the pre- and postfix versions of the two operators. If you used `i++` or `i--` (postfix), the compiler would create a copy of `i`, increase or decrease the copied value and assign the result to `i` at the end of the statement. By contrast, `++i` or `--i` (prefix) instructed the compiler to increase or decrease the value of `i` directly, without first creating a copy. As a consequence, the prefix versions of the increment and decrement operators (`++i` and `--i`) were slightly faster than their postfix versions (`i++` and `i--`). Again, modern compilers use optimisation methods that eliminate the difference between the pre- and postfix versions, at least, when `i` is a simple variable of integral type. Nevertheless, it is considered *best practice* to use `++i`, unless you have a specific reason to use `i++`.

ET on the pre- and postfix increment and decrement operator

The difference between pre- and postfix increment/decrement still matters if you use `++i` and `i++` as parts of larger expressions. For example, `int j = ++i + 1;` and `int j = i++ + 1;` do not initialise `j` with the same value. In addition, it is allowed to assign a value to `++i` and `--i`, but not to `i++` and `i--`. In other words, something like `++i *= 10` is allowed, but `i++ *= 10` is not. In practice, it is a good idea to simply avoid expressions in which pre- and postfix operators have different effects. This can be achieved by performing complicated calculations in multiple steps.

3.2 Building up larger pieces of code

Expressions

Applying an operator one or two variables or literals (depending on whether the operator has one or two operands) results in a C++ expression. Expressions always evaluate to a value that is stored (temporarily) in the computer's memory, allowing you to use expressions as building blocks of more complex expressions. For the standard arithmetic operators, this works in the obvious way: the expression `2 + 3` evaluates to the value 5, allowing us to write a more complex expression like `2 + 3 - 4`. This expression is then evaluated as `2 + 3 - 4` → `5 - 4` → `1`. Perhaps more surprisingly, also assignments and compound assignments evaluate to a value, allowing assignments to be concatenated, as in

```
iToes = iFingers = 10
```

In order to interpret such an expression, you need to read from right to left and use the value of the variable after an assignment as the value of the assignment expression. In other words, the expressions above evaluate as follows

```
iToes = iFingers = 10 → iToes = 10 → 10
```

after which `iToes` and `iFingers` both have the value 10. A more complex example is

```
1 int iTmp = 1, iSumSquare = 0;
2 iSumSquare += iTmp *= iTmp;
3
4 iTmp = 2;
5 iSumSquare += iTmp *= iTmp;
6
7 iTmp = 3;
8 iSumSquare += iTmp *= iTmp;
```

```
9
10 iTmp = 4;
11 iSumSquare += iTmp *= iTmp;
12
13 std::cout << "iSumSquare = " << iSumSquare << std::endl;
14 }
```

which produces the output `iSquare = 30` ($= 1^2 + 2^2 + 3^2 + 4^2$). To make sense of this result, let us consider the statement on line 5 and work out how its is resolved:

`iSumSquare += iTmp *= iTmp` → `iSumSquare += 4` → `5`

First, `iTmp` is multiplied with itself. The resulting value (4) becomes the new value of `iTmp`, which is also the result of the compound multiplication. The value 4 is then added to `iSumOfSquare`, which gets the value $5 (= 1^2 + 2^2)$.

Operator precedence rules

To avoid ambiguity in the meaning of complex expressions that involve multiple operators, C++ adheres to the following rules for deciding on the order in which operations are executed

Priority Operations are executed in the order of operator priority. For example, multiplication and division have higher priority than addition and subtraction. Therefore, `1 + 2 * 3` evaluates to 7 (1 + (2 * 3)), and not to 9 ((1 + 2) * 3). The highest priority C++operators that you have seen so far are the increment and decrement operators `++` and `--`, whereas assignment `=` and compound assignment operators have almost the lowest priority. A list of C++operators and their priority levels can be found in table 3.1 on page 33.

Association An expression that contains operators of the same priority is evaluated according to the rule of association. Most operators associate *from left to right*. For example, the expression `4 * 1 / 2` will be interpreted as `(4 * 1) / 2`, which evaluates to 2, and not as `4 * (1 / 2)`, which would evaluate to 0 (recall the integer division rules). A few operators associate *from right to left*. These include all unary operators, all (compound) assignment operators and the conditional operator `?:`.

You can always overrule these operator precedence rules by using brackets. Even when they are not strictly necessary, brackets can make code more easily readable, and using them can be helpful if you would otherwise not be entirely sure about how a complex expression is evaluated.

***Nefertiti* on brackets and the KISS! rule**

With more than 50 different operators and well-defined rules for combining them, C++ offers nearly unlimited possibilities for composing long and complicated expressions. Whether anyone other than the compiler will be able to make sense of such expressions is a different story. The KISS! rule (Keep It Simple, Stupid!) provides a valuable guideline for writing C++ code. Split complicated expressions into small parts, and use brackets whenever that helps to avoid misunderstanding.

Expression statements

When an expression is terminated by a semi-colon, it turns into an expression statement, one of the basic building blocks of a C++ program. Some examples of expression statements are:

```
dPi = 3.141592;
iAgeInMonths = 12 * iAgeInYears;
std::cout << "An important message!";
```

also the empty statement, which consists only of a `;`, is a valid expression statement.

Typically, an expression statement contains an expression that corresponds to a single step in the process executed by the program. However, the comma operator provides a way to pack multiple independent expressions into a single expression statement. For example,

```
dHeight = 1.96, dWeight = 78.0;
```

is a single expression statement, in which values are assigned to two variables. The comma operator is helpful when the C++ syntax rules prohibit the use of multiple statements, but in all other cases its use is strongly discouraged. Occasionally, you may see the use of the comma operator inside a larger expression, as in

```
dResult = (dHeight = 1.96, dWeight = 78.0);
```

This expression is equivalent to

```
dHeight = 1.96;
dResult = dWeight = 78.0;
```

In other words, in expression statements, the comma behaves as a sequencing operator that ensures evaluation from left to right. The value assigned to a composite comma expression is equal to the value of its right-most element.

Compound statements

Multiple statements can be grouped together by enclosing them in curly braces, which turns them into a compound statement. A compound statement is a statement itself, and it can be part of another compound statement (i.e., compound statements can be nested). Since a compound counts as a single statement, enclosing code in curly braces allows you to insert multiple instructions at places where the C++ syntax rules prescribe that only a single statement should be present. The function body of the function `main` (and that of other functions) is a compound statement by default:

```
int main()
{
   // program code defining main body
   return 0;
}
```

The scope of variables

Compound statements provide an important way of structuring your code, because they allow you to control the scope of variables. You can think of the scope of a variable as that part of the program where the variable is known to the compiler. Variables are unknown before they are declared. In other words, the scope of a variable starts at its declaration. The scope ends at the closing brace of the compound statement that contains the declaration of the variable.

The three integer variables in the following example program have three different scopes. Variable `iGlobal` is a so-called global variable. It is declared outside the function `main()`. Its scope ranges from lines 3 to 14 Variable `iLocal` is declared locally within the function `main()`. Its scope ranges from lines 7 to 14. Variable `iVeryLocal` is declared within a compound statement. It scope ranges only from line 9 to line 12.

Code listing 3.1: The scope of variables

```
#include <iostream>

int iGlobal = 0;

int main()
{
    int iLocal = 0;
    {
        int iVeryLocal = 7;
        std::cout   << iGlobal << iLocal << iVeryLocal
                    << " is a famous spy." << std::endl;
    }
    return 0;
}
```

The output generated by the program is:

```
007 is a famous spy.
```

When the scope of a variable ends, the compiler will erase the variable from the computer's memory. The memory is then available again for other purposes. Moreover, the more variables you have declared in a certain scope, the more difficult is it to keep an overview. Enclosing pieces of code in curly braces allows you to put temporary variables in their own scope, so that they live only for as long as you need them, e.g.,

```
int iValue1 = 10, iValue2 = 20;
{ //swap the two values
  int iTemp = iValue1;
  iValue1 = iValue2;
  iValue2 = iTemp;
}
```

***Nefertiti* on variable scope**

It is a good idea to declare variables as local as possible. Avoid global variables, unless there is no other solution. Global variables can be changed from everywhere in your program, and it will be very hard to localise errors involving global variables. Moreover, global variables occupy memory space throughout the execution lifetime of the program. Local variables allow for a more efficient use of memory space, which may increase the speed of your program by up to 25%.

Being, on average, not the most imaginative types of persons, C++ programmers often have trouble to think of original, descriptive and accurate names for variables. Moreover, to increase the readability of their code, they will often use the same names for different variables at different points in their code. This is allowed, if the scopes of the different variables end at different points. For example, the compiler will complain that you declare the same variable multiple times when you attempt something like this:

```
{
   int iSumOfSquares = 0;
   //many lines of highly imaginative code...

   int iSumOfSquares = 2;
   //even more imaginative code...
}
```

However, it is perfectly fine to write something like this:

```
{
   {
      int iSumOfSquares = 0;
      //many lines of boring code...
   }

   {
      int iSumOfSquares = 2;
      //even more extremely boring code...
   }
}
```

Here, the two variables can be distinguished by their different scopes, which allows you to use the same name for them. The scopes of different variables with the same name need not be completely separated as in the previous code fragment. Consider, for example, the mysterious program

Code listing 3.2: Deep Thought

```
1  #include <iostream>
2
3  int iMysteriousVariable = 4;
4
5  int main()
6  {
7      std::cout << iMysteriousVariable;
8
9      int iMysteriousVariable = 2;
10     std::cout << iMysteriousVariable << std::endl;
11
12     return 0;
13 }
```

which represents a highly simplified emulation of the operating system of the computer Deep Thought from Douglas Adams' the Hitchhiker's Guide to the Galaxy. It produces the output

```
42
```

Note that the scope of the first `iMysteriousVariable`, declared on line 3, completely encompasses the scope of the second integer with the same name, which is declared as a local variable on line 9. At line 7, the second `iMysteriousVariable` has not yet been declared, and the value 4 will be displayed on the screen. At line 10, both the local and the global `iMysteriousVariable` exist. The compiler resolves the resulting ambiguity by choosing the most local of the two variables, which has the value 2.

For applications other than finding *the ultimate answer*, recycling names with overlapping scopes is not recommended. You should always be aware of the scope of variables, but do not count on others to understand the intricacies of this issue as well. Re-use names only for variables with non-overlapping scopes that represent similar things, such as counters.

Table 3.1: List of C++ operators

Name	Symbol	Example	Priority[1]	Reference[2]
scope/namespace resolution	`::`	`std::cout`	18	
array index	`[]`	`array[2]`	17	51
initialisation	`()`	`int i(34)`	17	16
function call	`()`	`doSomething()`	17	64
postfix increment/decrement	`++ --`	`i++`	17	27
type cast	`static_cast<>()`[3]	`static_cast<double>(i) / j`	17	19
logical not	`!`	`!true`	16	39
unary plus or minus	`+ -`	`-1`	16	
multiplication/division	`* / %`	`i * 2 / j`	14	
addition/subtraction	`+ -`	`i + 2 - j`	13	
input/output	`>> <<`	`std::cout << i`	12	
relational	`> >= < <=`	`i > 1`	11	39
equality/non-equality	`== !=`	`i != 0`	10	39
logical AND	`&&`	`i > 0 && j < 1`	6	39
logical OR	`||`	`i > 0 || j < 1`	5	39
conditional	`? :`	`i < j ? i : j`	4	42
assignment	`=`	`i = 1`	3	16
compound assignment[4]	`+= -= *= /= %=`	`i /= 2`	3	27

1 Operators with higher priority are executed first.
2 Some operators are introduced at a later point. Consult the indicated page for more information.
3 Additional cast operators are `dynamic_cast<>()` and `const_cast<>()`.
4 There are also compound assignment operators for all bitwise logical operators (e.g. `&=`, `^=` and `<<=`).

Table 3.2: A selection of mathematical functions defined in the library `cmath`

Mathematical formula	C++ implementation	Function prototype[1]	Description
Trigonometric functions			
$z = \sin(x)$	`z = sin(x)`	`double sin(double)`	Returns the sine of an angle in radians
$z = \cos(x)$	`z = cos(x)`	`double cos(double)`	Returns the cosine of an angle in radians
$z = \tan(x)$	`z = tan(x)`	`double tan(double)`	Returns the tangent of an angle in radians
$z = \arcsin(x)$	`z = asin(x)`	`double asin(double)`	Inverse of the sine function
$z = \arccos(x)$	`z = acos(x)`	`double acos(double)`	Inverse of the cosine function
$z = \arctan(x)$	`z = atan(x)`	`double atan(double)`	Inverse of the tangent function
Exponential and logarithmic functions			
$z = e^x$	`z = exp(x)`	`double exp(double)`	Computes the exponential function
$z = \ln(x)$	`z = log(x)`	`double log(double)`	Returns natural logarithm of a number
$z =^{10} \log(x)$	`z = log10(x)`	`double log10(double)`	Returns the common (base-10) logarithm of a number
Power functions			
$z = x^y$	`z = pow(x, y)`	`double pow(double, double)`	Raise to a power
$z = \sqrt{x}$	`z = sqrt(x)`	`double sqrt(double)`	Returns square root of a number
$z = \sqrt{x^2 + y^2}$	`z = hypot(x, y)`	`double hypot(double, double)`	Returns the hypotenuse of a right-angled triangle whose legs are x and y^2
Rounding and remainder			
$z = \lfloor x \rfloor$	`z = floor(x)`	`double floor(double)`	Rounds down towards the nearest integer value
$z = \lceil x \rceil$	`z = ceil(x)`	`double ceil(double)`	Rounds up towards the nearest integer value
$z = \text{trunc}(x)$	`z = trunc(x)`	`double trunc(double)`	Rounds towards zero, to the nearest integer value[2]
$z = \text{round}(x)$	`z = round(x)`	`double round(double)`	Returns the integral value nearest to x, with halfway cases rounded away from zero[2]
$z = \text{remainder}(x/y)$	`z = fmod(x, y)`	`double fmod(double, double)`	Returns the floating-point remainder of the division x/y (rounded towards zero)
Other functions			
$z = \max(x, y)$	`z = fmax(x, y)`	`double fmax(double, double)`	Returns the largest of the values x or y^2
$z = \min(x, y)$	`z = fmin(x, y)`	`double fmin(double, double)`	Returns the smallest of the values x or y^2
$z = \lvert x \rvert$	`z = fabs(x)`	`double fabs(double)`	Returns absolute value of a number

1 Provides information about the number of arguments, their type and the type of the value returned by the function.
2 This function is a recent addition to the C++ standard library available from version C++11 onwards.

3.3 The library cmath

Additional functionality for providing calculations is provided by the functions in the standard C++ library cmath, which will be linked with your code if you include the preprocessor directive **#include**<cmath>. This library contains functions to perform common mathematical operations and transformations. Table 3.2 lists a selection.

3.4 Exercises

3.1. C++ operators. What is the value of the integer variables i and j after the following statements?

(a) `i = 7; i %= 3;`

(b) `i = 7; j = i *= 2;`

(c) `j = 4; j *= (i = 2);`

3.2. Too many brackets. Eliminate as many brackets from the following expression statements, without changing the final values of the integer variables i and j;

(a) `j = (4 * (i %= (3 * 2)) / 5);`

(b) `(j = (5 * (-(i += 5))));`

(c) `j = (2 * (++i)++) / (3 + 2);`

3.3. Correct scoping. What is the value of i and j at the end of the following code?

```
int i = 1;
int j = 5;
{
   int i = 0;
   int j = i + 1;
   i += j;
   j++;
}
i++;
j++;
```

3.4. Gompertz law for tumour growth. For many types of cancer, the size of a tumour is well described by Gompertz law for tumour growth

$$x(t) = x_{\max} \exp\left(\ln\left(\frac{x_0}{x_{\max}}\right)\exp(-\alpha t)\right)$$

Here, $x(t)$ is the size of the tumour at time t, x_0 is its initial size and $x_{\max}$ is the maximum size it can reach given the availability of nutrients. Write a program that calculates the size of a tumour at a given time point, assuming that $x_0 = 0.01$, $x_{\max} = 1.0$ and $\alpha = 0.2$. How long does it approximately take for the tumour to grow to 95% of its maximum size?

Test your skill

3.5. The *abc* formula. The solutions x^* of the second order polynomial equation $a\,x^2 + b\,x + c = 0$ can be found by applying the *abc* formula

$$x^* = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Write a program that reads three coefficients a, b and c from keyboard input, and that computes the locations of the two roots of the polynomial $a\,x^2 + b\,x + c = 0$. For the moment, assume that the program is only used when two solutions exist (i.e., when $b^2 > 4ac$).

3.5 Solutions to the exercises

Exercise 3.1. C++ operators.

(a) `i` has value 1.

(b) `i` has value 14, `j` has value 14.

(c) `i` has value 2, `j` has value 8.

Exercise 3.2. Too many brackets.

(a) `j = 4 * (i %= 3 * 2) / 5;`

(b) `j = 5 * -(i += 5);`

(c) `j = 2 * (++i)++ / (3 + 2);`

Exercise 3.3. Correct scoping. i = 2 and j =6

Exercise 3.4. Gompertz law for tumour growth.

```
/* This program calculates the size of a tumour
   that grows according to the Gompertz law.
   Written on 25?08?2016 by Master Yoda. */

#include <iostream>
#include <cmath>

int main()
{
    //read time point
    std::cout << "Please enter a time point: ";
    double dTime;
    std::cin >> dTime;

    //define parameters of the growth function
    const double dSize0   = 0.01;
    const double dSizeMax = 1.0;
    const double dAlpha   = 0.2;

    //calculate size at time point dTime

    const double dSize = dSizeMax * exp(log(dSize0 / dSizeMax) * exp(-dAlpha * dTime));

    //generate output
    std::cout << "x(" << dTime << ") = " << dSize << std::endl;

    //terminate program
    return 0;
}
```

For the given parameters, it takes approximately 22.5 time units for the tumour to grow to 95% of its maximal size.

Module 4

If..., else..., or what!?

Jamie explains what's on the menu

In this module you will:

- Learn to control program flow by means of logical expressions and conditional statements.
- Become acquainted with the type `bool` and the logical operators

So far, the programs we wrote were all executed statement by statement, from start to end. Such sequential program flow is far too rigid to allow for the development of simulation software, since almost any algorithm requires conditional or iterative instructions. In this module, we will look at conditional statements, which allow you to direct the program flow along different paths. Module 5, *Doing stuff many times*, will introduce iterative statements, which allow for the creation of program loops. Together, they enable you to implement almost any algorithm you like.

4.1 The conditional statement

The most commonly used conditional statement in C^{++} is the `if`-statement. An `if`-statement consists of the following elements:

(1) first the keyword `if`,

(2) then a pair of brackets `()` that enclose a *logical condition*,

(3) followed by a statement enclosed in brackets {} that will be executed if the logical condition is true.

This construct can be followed by

(4) the keyword `else`,

(5) followed by a statement enclosed in brackets {} that will be executed if the logical condition of the matching `if` is false.

Note that including the **else**-block (elements (4) and (5)) is optional. Furthermore, note that variables declared within the compound statements delinated by {} brackets, only exist with the scope of their own compound statement.

Here is an example[1] of an **if**-statement with a matching **else**:

Code listing 4.1: The if-else-statement

```
1
2 if (dTemperature > 20.0) {
3     goSwimming();
4 }
5 else {
6     std::cout << "It's too cold to go swimming!" << std::endl;
7 }
8 // continue regular program flow
```

The **if**-statement can be nested to check for additional conditions after evaluating the initial condition. For example,

```
1
2 if (dTemperature > 20.0) {
3     goSwimming();
4 }
5 else if(dTemperature < 0.0) {
6     goIceSkating();
7 }
8 else {
9     std::cout   << "It's too cold for swimming but too warm for ice skating. I'm bored!"
10                << std::endl;
11 }
12
13 // continue regular program flow
```

Here, a second **if**-statement has been inserted after the first **else**. You can also insert a nested **if-else** block for the statement following the **if**. In fact, an equivalent way of writing the above code that makes use of this possibility is:

```
1
2 if (dTemperature >= 0.0) {
3     if(dTemperature > 20.0) {
4         goSwimming();
5     }
6     else {
7         std::cout   << "It's too cold for swimming and too warm for ice skating. I'm bored!"
8                     << std::endl;
9     }
10 }
11 else {
12     goIceSkating();
13 }
14 // continue regular program flow
```

Instead of creating a compound statement delinated by {} brackets, an **if** statement can also execute an inline statement. This can be particularly useful to shorten your code and make it more readable. However, this can also easily cause confusion as to which code is exactly executed if the **if** statement resolves to true, therefore it is typically advised to always add {} brackets.

[1] The statement `goSwimming();` in the code fragment invokes a user-defined function that we have not written out. If you would like to execute the code, include it in the body of a function `main()` and replace `goSwimming();` by something like `std::cout << "Let's go swimming!" << std::endl;`. The same approach can be applied to the other code snippets in this Module.

```
if (dTemperature >= 0.0) goSwimming();
// continue regular program flow
```

***Nefertiti* on indentation**

The use of indentation, as done in the previous examples, is optional, given that C++ is a freeform language. Yet, it is highly advisable to develop a clear formatting style and to use it consistently. Indentation will help you keep an overview of the structure of a program, which would otherwise be a challenge, especially when there is a large compound statement between **if** and **else**, or when multiple **if-else** blocks are nested. Many code editors provide automatic indentation that produces program code consistent with the formatting style used in this manual.

***Jamie's recipes for disaster:* a semicolon too many**

Novice programmers sometimes develop the habit of ending every line of code with a semicolon, and may then write something like this:
`if(...);`
`{/*some statements*/}`
The tragic result is that the **if** now controls an empty statement, so that the following compound statement is executed always.

4.2 Logical expressions

The condition in an **if**-statement frequently consists of a logical expression containing one of the relational operators >, >=, <, <=, == or !=. These have the following meaning

>	is larger than;	<	is smaller than;
>=	is larger than, or equal to;	<=	is smaller than, or equal to;
==	is equal to;	!=	is not equal to;

More complex logical expressions can be built by combining expressions with the logical operators && (AND) and || (OR). For example, if you would like the program to execute `goRunning()` if `dTemperature` is between 10.0 and 20.0, you can write

```
if (dTemperature > 10.0 && dTemperature < 20.0) goRunning();
```

You might be tempted to write `if (10.0 < dTemperature < 20.0) goRunning();` to achieve the same effect, but it will soon become clear that that will not behave as you had probably expected.

The operator ! (logical NOT) is available to negate the result of a logical expression (turning true into false and vice versa). For example, a statement that is logically equivalent to the previous example is

```
if(!(dTemperature <= 10.0 || dTemperature >= 20.0)) goRunning();
```

i.e., execute `goRunning()` if it is not the case that `dTemperature` lies below 10.0 or above 20.0.

The type `bool`

Logical expressions are evaluated as either **true** or **false**, the two values that can be represented by a variable of the type **bool**. Variables of type **bool** have several applications, including that they can store the result of a logical expression in order to control an **if**-statement at some later point. Here is an example:

Code listing 4.2: The type bool

```
int iCurrentHour = 22;

const bool isNight = iCurrentHour >= 23 || iCurrentHour < 7;
const bool isFullMoon = true;

```

```
if(isNight) {
    if(isFullMoon) {
        doWereWolfThings();
    }
    else {
        doSleep();
    }
}
else {
   doEatWorkOrWatchTV();
}
```

which, luckily, results in the execution of `doEatWorkOrWatchTV()`.

Implicit type conversion can convert the Boolean value **`true`** to the integer value 1; the Boolean value **`false`** is converted to the integer value 0. Conversion can also occur in the other direction: in that case, all values other than zero are converted to **`true`**, whereas the value zero converts to **`false`**. This allows you to use a numeric value as a condition in an **`if`**-statement, as illustrated below:

```
int iNumerator, iDenominator;
// some lines of code that determine the values of iNumerator and iDenominator...

int iQuotient;
if(iDenominator) {
    iQuotient = iNumerator / iDenominator;
}
else {
    std::cout << "warning: division by zero; unable to compute iQuotient" << std::endl;
}
```

Here, the warning message will be printed if the condition in the **`if`**-statement evaluates to false, which happens when `iDenominator` has value zero.

The flexibility of C++ in dealing with logical expressions opens up the possibility for a nasty programming error that almost all beginning programmers will make a couple of times: mixing up the assignment and equality operator.

Jamie's recipes for disaster:
mixing up = and ==

Assigning a value and checking for equality are two different things, so C++ has separate operators for them (= and ==, respectively). However, since the result of an assignment evaluates to a value and any numerical value can be interpreted as a logical **`true`** or **`false`**, the compiler will not complain if you accidentally write **`if`**`(iInt = 1)` instead of **`if`**`(iInt == 1)`. In the first case, the statement following the **`if`** will always be executed, because the assignment `iInt = 1` evaluates to 1, which is interpreted as **`true`**. By contrast, `iInt == 1` evaluates to **`false`** unless the value of `iInt` is equal to 1.

The issue highlighted by Jamie also relates to the point made earlier: do not use expressions like **`if`** `(10.0 < dTemperature < 20.0) goRunning();`, because they are likely to produce unintended results. In particular, if the value of `dTemperature` is larger than 20.0, the logical expression is evaluated as `(10.0 < dTemperature) < 20.0`, in the following steps:

(1) `10.0 < dTemperature` → **`true`**

(2) **`true`** → 1 → 1.0 (implicit conversion from **`bool`** to **`int`** to **`double`**)

(3) `1.0 < 20.0` → **`true`**

so that the statement following **`if`** is executed.

4.3 Exercises

4.1. What kind of critter is this?. Write a C^{++} program that implements the following simplified determination table from a children's field guide to small creatures:

1. Does the bug fly? — *If yes,* ***Answer: It's an insect****; if no, continue*
2. Does the bug have 8 legs? — *If yes,* ***Answer: It's a spider****; if no, continue*
3. Does the bug have 6 legs? — *If yes,* ***Answer: It's an insect****; if no, continue*
4. Does the bug have no legs? — *If yes,* ***Answer: It's a worm or a larva****; if no, continue*
5. ***Answer: It might be a millipede***

As a shortcut, use variables `doesFly` and `iNumberOfLegs` with preassigned values, instead of allowing the user to set these values via keyboard input.

4.2. Exclusive OR. Exclusive OR (XOR) is a logical operation that acts on two logical input values and that evaluates to true if one of the two inputs is true, but not both. Write a C^{++} program that implements an XOR switch. Define two logical variables, `bool isInputA, isInputB`, and print `Hurrah!` to the screen if and only if XOR(`isInputA`, `isInputB`) is true. Write a first implementation of the XOR switch and try all four combinations of the values **`true`** and **`false`** for the two input variables to verify that your implementation is correct.

The XOR switch can be implemented in several different ways. Modify your code to create three alternative implementations:

(a) with nested **`if-else`** statements and without using the operators `&&`, `||`, `!`, `==` or `!=`

(b) with a single **`if-else`** statement and without using `==` or `!=`

(c) with a single **`if-else`** statement and without using `&&`, `||` or `!`

4.3. The optimist/pessimist quiz ($\star$). Write a program that collects answers to three multiple choice questions from the user, and that classifies him/her as either an optimist or a pessimist based on the answers provided. Each multiple choice question should have three alternative answers; at least one of them should be rated as an optimist answer and another one as a pessimist answer, and these should not be listed in the same order across the three questions.

Test your skill

4.4. Temperature converter and weather advisory ($\star$). Write a program to convert the temperature from Celsius to Fahrenheit, or from Fahrenheit to Celsius, depending on the chosen input by the user (see formula's below). The program should print both temperatures. Then, provide weather advice depending on the temperature in degrees Celsius.
If the converted temperature is below 0 degrees, print "Advisory: It is freezing outside, dress warmly!"
If the temperature is between 0 and 10 degrees, print "Advisory: It is chilly outside, you might want to wear mittens!"
If the temperature is between 10 and 20 degrees print "Advisory: the weather can still be cool, a light jacket is advised.'"
If the temperature is between 20 and 30 degrees print "Advisory: The weather is pleasant, enjoy your day!".
Lastly, if the temperature is above 30 degrees, print: "Advisory: it's hot! Stay hydrated and avoid direct sunlight".

$$F = 32 + C * \frac{9}{5} \tag{4.1}$$

$$C = (F - 32) * \frac{5}{9} \tag{4.2}$$

Alternatives to the `if-else`-statement

The conditional operator

Apart from the usual **`if-else`**-statement, C++ also features the conditional operator, which provides some of the functionality of the **`if-else`**-statement in a condensed form. Its use is mainly restricted to cases like the following

```
double dVal1, dVal2;
// some lines of code that determine the values of dVal1 and dVal2...

double dLargest;
if(dVal1 > dVal2) {
    dLargest = dVal1;
}
else {
    dLargest = dVal2;
}
```

where the instructions in the **`if`** and **`else`** block simply assign two different values to the same variable. Using the conditional operator `?:`, the same code can be written much more concisely as follows:

Code listing 4.3: The conditional operator

```
double dVal1, dVal2;
// some lines of code that determine the values of dVal1 and dVal2...

const double dLargest = dVal1 > dVal2 ? dVal1 : dVal2;
```

The code fragment `dVal1 > dVal2 ? dVal1 : dVal2` consists of a single expression with three parts. The part before the question mark is evaluated as a logical expression; if its value is **`true`**, the expression will assume the value of the part between the question mark and the colon; if the condition is **`false`**, the expression will assume the value of the part after the colon.

A second example illustrates the conditional operator in the context of an output operation. First, an implementation based on the normal **`if-else`**-statement:

```
int iInt;
// some lines of code that determine the value of iInt...

if(iInt % 2 == 0) {
    std::cout << "The number " << iInt << " is even." << std::endl;
}
else {
    std::cout << "The number " << iInt << " is odd." << std::endl;
}
```

which simply shows on the screen whether `iInt` is even or odd. The implementation with the conditional operator requires only a single output statement:

```
int iInt;
// some lines of code that determine the value of iInt...

std::cout << "The number " << iInt << (iInt % 2 == 0 ? " is even." : " is odd.") << std::endl;
```

The `switch`-statement

The **switch**-statement provides another alternative for the **if-else**-statement that is useful when program flow has to branch into more than two paths. To illustrate such a situation, consider the following code fragment from the implementation of a role playing game:

```
std::cout   << "Where do you want to go?" << std::endl
            << "left (1), right (2), up (3) or down (4): ";
int iDirection;
std::cin >> iDirection;

if(iDirection == 1) {
    std::cout << "You see a beautiful garden." << std::endl;
}
else if(iDirection == 2) {
    std::cout << "You are killed by a monster." << std::endl;
}
else if(iDirection == 3) {
    std::cout << "You get lost in a swamp and drown." << std::endl;
}
else if(iDirection == 4) {
    std::cout << "You discover a secret treasure." << std::endl;
}
else {
    std::cout   << "unknown direction: "
                << "enter 1, 2, 3 or 4." << std::endl;
}

// continue regular program flow
```

The **switch**-statement replaces the series of nested **if-else**-statements in the above example by a single block with a list of case labels, each followed by a specification of what to do in each particular case. In particular, the **switch**-statement equivalent of the code above is given by:

Code listing 4.4: The switch statement

```
std::cout   << "Where do you want to go?" << std::endl
            << "left (1), right (2), up (3) or down (4): ";
int iDirection;
std::cin >> iDirection;

switch (iDirection) {
    case 1: {
        std::cout << "You see a beautiful garden." << std::endl;
        break;
    }
    case 2: {
        std::cout << "You are killed by a monster." << std::endl;
        break;
    }
    case 3: {
        std::cout << "You get lost in a swamp and drown." << std::endl;
        break;
    }
    case 4: {
        std::cout << "You discover a secret treasure." << std::endl;
        break;
    }
    default : {
        std::cout   << "unknown direction: "
                    << "enter 1, 2, 3 or 4." << std::endl;
        break;
    }
```

```
}

// continue regular program flow
```

As you can see, a **switch**-statement contains the following elements:

(1) first, the keyword **switch**,

(2) followed by a pair of brackets `()` that enclose an expression,

(3) and finally, a *compound statement* with a list of cases.

Each item in the list of cases consists of

(a) the keyword **case**, followed by

(b) a label representing a potential value of the switch expression,

(c) a colon, and

(d) an arbitrary number of statements,

(e) which, in typical cases, have a break statement (**break**;) at their end.

In addition, there can be one item in the list that contains

(a) the default label **default**, followed by

(b) a colon, and

(c) an arbitrary number of statements,

(d) which, in typical cases, have a break statement (**break**;) at their end.

The expression controlling a **switch**-statement must evaluate to an integer or character value (more about characters, and the associated data type **char** will follow in Module 9), and there can be no more than 16384 different cases. When the **switch**-statement is executed, the value of the **switch**-expression is tested against the list of **case** labels. When an exact match is found, statements will be executed from that point onwards until a **break** or the end of the switch block is reached. Hence, if there is no **break** at the end of the statement sequence belonging to a **case**, the code corresponding to the next **case** will be executed as well. Programmers sometimes make use of this feature by stacking multiple **case**s, like in the example below. Including a **default** label is optional. If you do, its associated code will be executed when the switch expression does not match with any of the **case** labels.

Code listing 4.5: `switch`-statement with stacked `case`-labels

```
std::cout   << "Please enter a number in the range from 2 to 9: ";
int iInput;
std::cin >> iInput;

switch (iInput) {
    case 2: case 3:
    case 5: case 7:
        std::cout << "The number you entered is prime." << std::endl;
        break;
    case 4: case 6:
    case 8: case 9:
        std::cout << "The number you entered is not prime." << std::endl;
        break;
    default:
        std::cout   << "The number you entered is not in the range from 2 to 9." << std::endl;
        break;
}
```


More exercises

4.5. The conditional operator. Use the conditional operator to replace the **if-else** statement in the following program fragment

```
std::cout << "Please enter a number: ";
double dVal;
std::cin >> dVal;

double dClip = dVal;
if(dVal < 0.0) {
   dClip = 0.0;
}
```

4.6. The switch-statement. Replace the **if-else** ladder in the following program fragment by a **switch**-statement.

```
std::cout   << "Who is most resistant to alcohol?" << std::endl
            << "(1) Master Yoda" << std::endl
            << "(2) E.T." << std::endl
            << "(3) Nefertiti" << std::endl
            << "(4) Arnold Schwarzenegger" << std::endl
            << "(5) Jamie Oliver" << std::endl
            << "    Your answer: ";

int iAnswer;
std::cin >> iAnswer;

if (iAnswer == 1 || iAnswer == 2) {
     std::cout << "wrong! (extraterrestrials don't drink alcohol)" << std::endl;
}
else if (iAnswer == 3) {
    std::cout << "wrong! (Nefertiti only drinks the milk of she-donkeys)" << std::endl;
}
else if (iAnswer == 4) {
    std::cout << "correct!" << std::endl;
}
else if (iAnswer == 5) {
    std::cout << "wrong! (Jamie has a sip of wine now and then, but usually spits it out after
    tasting)" << std::endl;
}
else {
    std::cout << "invalid answer" << std::endl;
}
```

4.4 Solutions to the exercises

Exercise 4.1. What kind of critter is this?.

```
const bool  doesFly         = false;
const int   iNumberOfLegs   = 5;

if(doesFly || iNumberOfLegs == 6) {
    std::cout << "It's an insect." << std::endl;
} else if(iNumberOfLegs == 8) {
    std::cout << "It's a spider." << std::endl;
} else if(iNumberOfLegs == 0) {
    std::cout << "It's a worm or a larva." << std::endl;
} else {
    std::cout << "It might be a millipede." << std::endl;
}
```

Exercise 4.2. Exclusive OR.

```
const bool isInputA = true;    //try all 4 combinations of true and false to verify that the implementations are
    correct
const bool isInputB = false;

// (a) XOR with nested if-else statements
if(isInputA) {
    if(isInputB) {
        std::cout << "Booo!" << std::endl;
    } else {
        std::cout << "Hurrah!" << std::endl;
    }
} else {
    if(isInputB) {
        std::cout << "Hurrah!" << std::endl;
    } else {
         std::cout << "Booo!" << std::endl;
    }
}

// (b) XOR with single if-else statement, without == or !=
if((isInputA || isInputB) && !(isInputA && isInputB)) {
    std::cout << "Hurrah!" << std::endl;
} else {
    std::cout << "Booo!" << std::end
}
// (c) XOR with single if-else statement, without &&, || or !
if(isInputA != isInputB) {
    std::cout << "Hurrah!" << std::endl;
} else {
    std::cout << "Booo!" << std::endl;
}
```

Exercise 4.3. The optimist/pessimist quiz.

```
/* This program classifies the user as optimist
   or pessimist, based on a response to
   three multiple choice questions.
   Written on 01?09?2016 by Master Yoda. */

#include <iostream>

```

```cpp
int main()
{
    std::cout   << "Welcome to this test." << std::endl
                << "Please answer the following questions:" << std::endl;

    // collect answers and accumulate score
    // question 1
    std::cout   << std::endl
                << "A. How has the weather been this summer?" << std::endl
                << "   1: excellent" << std::endl                          //optimist
                << "   2: some good, some bad weather" << std::endl        //realist
                << "   3: horrible" << std::endl                           //pessimist
                << "            Answer: ";

    int iAnswer, iScore = 0;
    std::cin >> iAnswer;
    if(iAnswer == 1) {
      ++iScore;
    } else if(iAnswer == 3) {
        --iScore;
    }

    // question 2
    std::cout   << std::endl
                << "B. What is your favourite animal?" << std::endl
                << "   1: puppy" << std::endl                              //realist
                << "   2: rat" << std::endl                                //optimist
                << "   3: I don't like animals at all." << std::endl       //pessimist
                << "            Answer: ";

    std::cin >> iAnswer;
    if(iAnswer == 2) {
      ++iScore;
    } else if(iAnswer == 3) {
      --iScore;
    }

    // question 3
    std::cout   << std::endl
                << "C. How do you rate your progress in this course so far?" << std::endl
                << "   1: I'm sure I will never master C++ programming." << std::endl
    //pessimist
                << "   2: I'm already considering a career as IT professional." << std::endl
    //optimist
                << "   3: So far, so good." << std::endl
    //realist
                << "            Answer: ";

    std::cin >> iAnswer;
    if(iAnswer == 2) {
      ++iScore;
    } else if(iAnswer == 1) {
      --iScore;
    }

    // users with scores below zero are classified as pessimists
    if(iScore < 0) {
        std::cout << "Result: I guess you're a pessimist." << std::endl;
    }
    else {
         std::cout << "Result: I guess you're an optimist." << std::endl;
    }
    return 0;
}
```

Exercise 4.5. The conditional operator.

```
std::cout << "Please enter a number: ";
double dVal;
std::cin >> dVal;

const double dClip = dVal < 0.0 ? 0.0 : dVal;
```

Exercise 4.6. The switch-statement.

```
std::cout   << "Who is most resistant to alcohol?" << std::endl
            << "(1) Master Yoda" << std::endl
            << "(2) E.T." << std::endl
            << "(3) Nefertiti" << std::endl
            << "(4) Arnold Schwarzenegger" << std::endl
            << "(5) Jamie Oliver" << std::endl
            << "    Your answer: ";

int iAnswer;
std::cin >> iAnswer;

switch(iAnswer) {
    case 1:
    case 2:
        std::cout << "wrong! (extraterrestrials don't drink alcohol)" << std::endl;
        break;
    case 3:
        std::cout << "wrong! (Nefertiti only drinks the milk of she-donkeys)" << std::endl;
        break;
    case 4:
        std::cout << "correct!" << std::endl;
        break;
    case 5:
        std::cout << "wrong! (Jamie has a sip of wine now and then, but usually spits it out
    after tasting)" << std::endl;
        break;
    default:
        std::cout << "invalid answer" << std::endl;
}
```

Module 5

Doing stuff many times

Jamie explains what's on the menu

In this module you will:

- Learn how to use iterative statements to execute pieces of code multiple times.
- Be introduced to the container class std::vector, which allows you to organise and work with large numbers of variables.

5.1 The `for`-loop

Iterative statements allow you to execute a set of instruction for a predefined number of times, or until a stop-criterion has been met. They are more commonly known as *loops*. Loops occur in most C++ programs you will encounter and, together with the conditional statement, form a basis for almost any algorithm you might imagine. The most versatile statement for creating loops is the **`for`**-statement, which is constructed as follows:

(1) First write the keyword **`for`**, followed by

(2) a pair of brackets `()` that enclose

(2a) an expression that specifies an *initial value* for a loop variable followed by a semicolon; often, the initialiser is a combined declaration/initialisation statement,

(2b) a logical expression that determines a *stop criterion* for the loop, followed by a semicolon; the loop will be executed as long as this loop condition evaluates to **`true`**,

(2c) an expression that specifies how the loop variable is adjusted after each iteration.

(3) The final element is a *single statement* that defines the *body of the loop* and that contains the instructions that need to be iterated. The body of the loop can be a compound statement.

A simple example of a **`for`**-loop is shown in code fragment 5.1. When program execution reaches this **`for`**-statement, the first thing that happens is that the integer loop variable `i` is created and initialised with the value zero. Next, the program checks if the logical condition `i < 4` is **`true`**, and if this is the case, the body of the loop will be executed once. After that, the loop variable is updated, which, in this case,

Code listing 5.1: The for-statement

```
for(int i = 0; i < 4; ++i)
   std::cout << "i = " << i << std::endl;
```

means that the value of `i` is incremented by one. The program simply repeats these steps (checking the logical condition, executing the body and updating the loop variable) for as long as the logical condition evaluates to **`true`**. If the condition is **`false`**, the loop is terminated, meaning that any variables that were created within its scope are eliminated, and the program continues with the statement following the **`for`**-loop. As you would expect based on this explanation, the output generated by the **`for`**-loop in listing 5.1 is:

```
i = 0
i = 1
i = 2
i = 3
```

`for`-statements can be nested, and the body of a **`for`**-loop can consist of multiple instructions, provided that you group them into a compound statement. Not surprisingly, therefore, Nefertiti's recommendations for using a consistent style of indentation (see page 39) apply equally to the **`for`**-loop. Moreover, we repeat Jamie's earlier warning, because the habit of writing semicolons at the end of every line can cause nasty errors with **`for`**-statement, in a similar way as you have seen for the **`if`**-statement:

Jamie's recipes for disaster: **a semicolon too many**

Novice programmers sometimes develop the habit of ending every line of code with a semicolon, and may then write something like this:
`for(...);`
`{/*some statements*/}`
The tragic result is that the **`for`**-loop will execute an empty statement a couple of times, whereas the following compound statement is always executed exactly once.

The three expressions regulating the **`for`**-loop can occur in many different forms, and none of them are obligatory, providing almost unlimited flexibility for building all kinds of loop constructs. Here are a few examples to illustrate some of the possibilities:
1. The loop variable need not be a local variable within the loop.

```
int iNumberOfOffspring;
// ...some lines of code that determine the value of iNumberOfOffspring...

for(; iNumberOfOffspring; --iNumberOfOffspring) // the initialisation instruction is empty
   produceAnOffspring();  // the number of times the body is executed is
                          // determined by the initial value of iNumberOfOffspring

// the variable iNumberOfOffspring still exists here and has value 0 when the loop terminates
```

2. The comma operator, which combines multiple expressions into one, allows you to create loops with multiple loop variables.

```
for(int i = 0, k = 1; i < 5; ++i, k *= 2)
   std::cout << "2^" << i << " = " << k << std::endl; // this will print the first five powers of 2
```

3. The loop variable can be updated within the body of the loop.

```
std::cout << "Please enter a number between 0 and 10: ";
```

```
int iInt;
std::cin >> iInt;          // loop variable is initialised here
for(; iInt < 0 || iInt > 10 ;) { // initialisation and update instructions are empty
   std::cout << "Invalid input, please try again: ";
   std::cin >> iInt;             // loop variable is updated here
}
```

4. It is also possible to have a **for**-loop without a stop criterion. In such cases, to prevent the program from entering an infinite loop, it is necessary to include a **break** statement, which provides another way to terminate the iterations. Compare the following code with the previous example. Note how the **break** statement allows you to escape from the middle of a loop, so that it is no longer necessary to have duplicate calls to std::cin

```
std::cout << "Please enter a number between 0 and 10: ";
int iInt;
for(;;) { // initialization, continuation criterion and update instruction are empty
   std::cin >> iInt;                              // iInt gets a value here
   if(!(iInt < 0 || iInt > 10)) break;            // escape from the loop if input is valid
   std::cout << "Invalid input, please try again: "; // otherwise, try again...
}
```

5.2 The container class std::vector

Iterative statements are particularly useful in combination with vectors. A vector is a collection of variables of the same type that are referred to by a common name. The variables in a vector are stored at adjacent positions in the computer's memory, allowing quick access to them.

The container class std::vector is declared in the header <vector> which is part of the standard template library (STL). To include this header, add the statement **#include** <vector> at the start of your .cpp file. Next, you can write

```
std::vector<int> vec(15);
```

to declare a vector of 15 integer variables that are referred to by the common name vec. To access one particular element of the vector, write the name of the vector, followed by square brackets surrounding the index of the element. In a vector of size n, the indices run from 0 to $n-1$, so the statement

```
vec[0] = 28;
```

assigns value 28 to the first element in the vector, and

```
std::cout << vec[14];
```

prints out the value of the last element. Note that you have both read and write access to the vector elements, depending on whether vec[i] appears on the right- or left-hand side of an assignment. Listing 5.2 illustrates both modes of access and also shows how a **for**-loop can iterate over the elements of a vector.

The recurrence relationship that is implemented in listing 5.2 represents, in fact, the earliest known population growth model. It was proposed in 1202 AD by the Italian mathematician Fibonacci as a model for the growth of an idealised rabbit population. Fibonacci's model gives rise to a famous sequence of numbers, the *Fibonacci sequence*. The first 15 numbers in the sequence are given by the console output of program 5.2:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ...
```

Code listing 5.2: Iterating over elements of a std::vector

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

int main()
{
    std::vector<int> vec(15);

    // set first two values
    vec[0] = 0;
    vec[1] = 1;

    // calculate remaining values from the recurrence
    // relationship vec[i] = vec[i - 1] + vec[i - 2]
    for(int i = 2; i < 15; ++i) {
        vec[i] = vec[i - 1] + vec[i - 2];
    }

    // print values in vec to the screen
    for(int i = 0; i < 15; ++i) {
        std::cout << vec[i] << ", ";
    }
    std::cout << "..." << std::endl

    return 0;
}
```

***Kermit the Frog* on Fibonacci's model**

Fibonacci assumed that a pair of newly born rabbits (one male and one female) can mate when they reach the age of one month, so that the female produces her first litter when she is two months of age. From that moment onwards, the pair will produce one new pair of rabbits every month. At the end of the first month, there is still only the original pair; at the end of month two, they are joined by their first offspring. At the end of the third month, the original pair has produced another litter, so that the total number of pairs amounts to three. At the fourth month, two pairs of newborns are added to the population, one produced by the original pair, and one produced by their first offspring. Continuing to reason like this, Fibonacci discovered that the number of pairs after n months, p_n, satisfies

$$p_n = p_{n-1} + p_{n-2}$$

Here, p_{n-1} is the number of pairs alive last month and p_{n-2} is the number of offspring pairs added to the population (this is given by the number of pairs two months ago, because only these will by now be at lest two months old)

***Jamie's recipes for disaster:* overrunning the end of an array**

The compiler does not check if you accidentally read or write outside the range of a `std::vector` (following the design principle that *C++ will not prevent users from shooting themselves in their feet*). While nothing prevents you from overrunning the end of an array, fatal errors are likely to occur if you do. For example, if you fail to respect the bounds of an array in an assignment operation, you will be assigning values to some other variable's data, which may cause your program to crash.

Multi-dimensional arrays

It is straightforward to create a vector of **doubles**, **bools**, or any other elementary data type, by simply adjusting the type specification in the definition of the vector variable. For example,

```
std::vector<float> vec(10)
```

declares a vector of 10 **float** variables. You can also create vectors of more complex objects, such as other vectors. In this way, it is possible to create multi-dimensional arrays. Storing such arrays can take a lot of memory space, so, in practice, large arrays rarely have more than two or three dimensions. Code listing 5.3 illustrates how you can create a matrix of integers by constructing a vector of `std::vector<int>`s. The matrix created in this example has 3 rows and 4 columns, and is given by

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \end{pmatrix}$$

Take particular notice of how elements in the matrix are accessed by means of multiple indices. For example, on line 18 of listing 5.3, the matrix element at row i and column j is accessed as `matrix[i][j]`, with one index for each array dimension, each one enclosed by its own set of square brackets.

Code listing 5.3: Creating a matrix

```
1  #include <iostream>
2  #include <vector>
3
4  int main()
5  {
6      // specify matrix dimensions
7      const int iNrRows = 3;       // number of rows
8      const int iNrCols = 4;       // number of columns
9
10     // create a vector of iNrRows row vectors
11     std::vector< std::vector<int> > matrix(iNrRows);     // insert a space between two subsequent >'s
12                                                          // to avoid confusion with the » operator
13
14     // initialise matrix elements
15     for(int i = 0; i < iNrRows; ++i) {
16         matrix[i] = std::vector<int>(iNrCols);    // create a vector of iNrCols elements for each row
17         for(int j = 0; j < iNrCols; ++j) {
18             matrix[i][j] = i + j;                 // specify intial value for each element
19         }
20     }
21
22     // show matrix
23     for(int i = 0; i < iNrRows; ++i) {
24         for(int j = 0; j < iNrCols; ++j) {
25             std::cout << matrix[i][j] << " ";      // print elements in row i
26         }
27         std::cout << std::endl;                   // newline at end of each row
28     }
29
30     //terminate program
31     return 0;
32 }
```

Different ways to create and initialise a `std::vector`

Single variables can be initialised at the time of their declaration and the same is possible for a `std::vector`. In fact, there are several different ways to initialise a vector object:

(1) `std::vector<bool> vec;` declares a vector of booleans that has no elements initially;

(2) `std::vector<bool> vec(5);` declares a vector of 5 boolean variables that are left uninitialised;

(3) `std::vector<bool> vec(5, true);` declares a vector of 5 boolean variables that all are initialised with the value `true`;

(4) `std::vector<bool> vec(vecOther);` declares a vector that is initialised as a copy of some other vector `vecOther`.

The third way of initialising a vector is applied in the following shortcut method for creating a `iNrRows` × `iNrCols` matrix object that is initialized with zeroes:

```
std::vector<std::vector<int> > matrix(iNrRows, std::vector<int>(iNrCols, 0))
```

Here, the vector object that is used as initial value for the rows of the matrix is created on the spot by the expression `std::vector<int>(iNrCols, 0)`.

Code listing 5.4: Some member functions of a std::vector

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

int main()
{
   // read series of numbers from keyboard input
   std::vector<double> vec;
   std::cout << "Please enter positive numbers, or a negative number to stop." << std::endl;
   for (;;) {
      std::cout << "Your next input: ";
      double dInput;
      std::cin >> dInput;

      if (dInput < 0.0)
         break;
      else
         vec.push_back(dInput);  // adds the value dInput at the end of vec
   }

   // write numbers to screen in ascending order
   std::cout << "The positive numbers you entered are, in ascending order: " << std::endl;
   while (!vec.empty()) {  // as long as vec still has elements...
      // ... move the smallest element to the back ...
      for (int i = 0; i < vec.size() - 1; ++i)
         if (vec[i] < vec.back()) { // (if vec[i] is smaller than the last element,...
            double tmp = vec[i]; // ...swap vec[i] and vec.back())
            vec[i] = vec.back();
            vec.back() = tmp;
         }

      // ...write its value to screen, ...
      std::cout << vec.back() << std::endl;

      // ...and remove it from the vector.
      vec.pop_back();
   }
}
```

Some member functions of a `std::vector`

Variables of type `std::vectors` are so called class objects, which have all kinds of functionality built into them (you will learn more about classes in Module 11, *Home-made objects*). Access to these built-in functions is gained via the member selection operator `.` ('dot'). A useful member function is `push_back()`, which adds an element to the end of the vector; `pop_back()` has the opposite effect: it removes the last element. The values of the first and last element can be obtained by the calling the functions `front()` and `back()`, respectively, and the function `size()` returns the current number of elements in the vector. A similar kind of information is provided by `empty()`, which returns `true` if there are no elements in the

vector, and `false` otherwise. Code listing 5.4 shows an application of these functions in a program that sorts a series of numbers entered by the user.

5.3 Exercises

5.1. Find the bugs. Below is a piece of code that is supposed to list the divisors of a number `iNumber` that is defined at some earlier position in the program. The code fragment contains two serious mistakes. Can you find and correct them?

```
for(int i = 1; i <= iNumber; ++i);
{
    int j = iNumber % i;
    if(j = 0)
        std::cout << iNumber << " is divisable by " << i << std::endl;
}
```

5.2. The Ricker model of population growth. The Ricker model is one of several classic discrete-time population growth models used in ecology to describe the change of the number of individuals in a population over time. In the Ricker model, the density of individuals N_t in generation t is fully determined by the density of individuals in the previous generation, N_{t-1}, according to the equation

$$N_t = N_{t-1}\, e^{r(1-N_{t-1})}$$

Here, r is a parameter that quantifies the intrinsic growth rate of the population, which corresponds to the maximal number of offspring produced per individual, $\exp(r)$. The term $\exp(r(1-N_{t-1}))$ is usually interpreted as the average number of offspring produced per individual. In the Ricker model, this quantity is assumed to decreases as an exponential function of the population size, to capture the negative effects of increased competition on offspring production at higher population densities. Write a C^{++} program to iterate the Ricker model, and study its behaviour for a few different values of the parameter r (e.g., try $r = -0.05$, $r = 0.5$, $r = 2.0$, $r = 2.5$ and $r = 3.0$). In all cases, set the initial population density to $N_0 = 0.001$, simulate for 100 timesteps, and store the sequence of population densities $N_0, N_1, \ldots, N_{100}$ in a `std::vector`.

5.3. Creating a histogram. Use the vector below as input for your program and write code to count the occurence of each number below 10.

```
std::vector<int> vec = {1, 1, 1, 3, 5, 7, 7, 2, 2, 8, 9, 2, 2, 3, 4};
```

5.4. Estimating rates of mutation ($\star$). A genome-wide comparison between an ancestral and a derived species counted the following totals of single base-pair mutations [1]

ancestral species base pair	**derived species base pair** AT	TA	CG	GC
AT		24	26	111
TA	41		91	33
CG	35	114		34
GC	93	33	27	

Write a C^{++} program to estimate the relative rates of different kinds of mutations that occurred from the ancestral to the derived species. Proceed as follows:

[1] Genetic information is redundant: the four nucleobases, A, T, C and G, bind complementarily (to T, A, G and C respectively) to form *base pairs*, the building blocks of double-stranded DNA. Here is an example of a base-paired DNA sequence:

A**T**CGATTGAGCTCTAGCG
T**A**GCTAACTCGAGATCGC

When a mutation occurs, the base pair changes, such that a mutation T $\rightarrow$ C leads to the example sequence becoming:

A**C**CGATTGAGCTCTAGCG
T**G**GCTAACTCGAGATCGC

1. First, define a matrix object that contains the mutation counts from the table above.
2. Next, pool the counts from the upper-diagonal and lower-diagonal half of the table to estimate six relative mutation rates: AT ↔ TA, AT ↔ CG, AT ↔ GC, TA ↔ CG, TA ↔ GC and CG ↔ GC.
3. Finally, estimate the relative rates of transitions (mutations involving the substitution of a purine base for another purine, i.e., A ⇌ G, or the substitution of a pyrimidine base for another pyrimidine, i.e., C ⇌ T) and transversions (all other substitutions).

5.5. Nasty nested for-statements (⋆⋆). Compile the following code on your computer, execute the program and examine the output.

```
1  #include <iostream>
2
3  int main()
4  {
5      for(int iPrimeCandidate = 2, iFactor; iPrimeCandidate < 1000; ++iPrimeCandidate) {
6          for(iFactor = 2; iFactor <= (iPrimeCandidate / iFactor); ++iFactor)
7              if(!(iPrimeCandidate % iFactor))
8                  break;
9          if(iFactor > (iPrimeCandidate / iFactor))
10             std::cout << iPrimeCandidate << " is prime" << std::endl;
11     }
12     return 0;
13 }
```

(a) Describe the function of the program.

Now, carefully go through the code and try to answer the following questions:

(b) What happens if you declare the loop variable `iFactor` in the inner **`for`**-loop instead of in the outer one? Compare the scope of the loop variables in the modified and the original code to explain what goes wrong.

(c) Interpret the expression `!(iPrimeCandidate % iFactor)` on line 7. Under what conditions is the **`break`** statement on line 8 executed?

(d) If the inner **`for`**-loop is not terminated by the **`break`**-statement on line 8, it will eventually be terminated by the loop condition on line 6. Why is it sufficient to only consider values of `iFactor` for which `iFactor <= (iPrimeCandidate / iFactor)`?

(e) Now consider the expression `iFactor > (iPrimeCandidate / iFactor)` on line 9. Does this condition evaluate to **`true`** if the inner **`for`**-loop has been terminated by the **`break`**-statement on line 8? How about when the loop was terminated by the loop condition on line 6?

Test your skill

5.6. Descriptive statistics. Write a program that reads a set of positive values from the keyboard input, as in listing 5.4. After collecting the values in a vector, the program should — if enough data are available — compute the mean μ and standard deviation $\sigma = \sqrt{\sigma^2}$ (i.e. square root of the variance), of the data and report these values back to the user.

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{5.1}$$

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n} (x_i - \mu)^2 \tag{5.2}$$

Additional jump statements

Besides the **break** statement, there are two other ways of interfering with the usual operation of a **for**-loop. The first is the **continue** statement, which will cause the execution of the loop to proceed immediately to the update step without first finishing the execution of the body. The following code fragment shows an example application, side-by-side with an equivalent alternative implementation that avoids the use of **continue** (focus on lines 11-20 to see the differences). You can decide for yourself which version is easier to read and understand.

```
1  double dSumInverse = 0.0;
2  int iNrValidInput = 0;
3  for(;;) {
4     std::cout << "Please enter a strictly
       positive"
5               << " number or zero to stop: ";
6     double dInput;
7     std::cin >> dInput;
8
9     if(dInput == 0.0)
10       break;
11    else if(dInput < 0.0) {
12       std::cout << "Ignoring negative
       number, "
13                 << "please try again."
14                 << std::endl;
15       continue;
16    }
17
18    dSumInverse += 1.0 / dInput;
19    ++iNrValidInput;
20 }
21
22 if(iNrValidInput)
23    std::cout << "The harmonic mean of the "
24              << "numbers you entered is: "
25              << iNrValidInput / dSumInverse
26              << std::endl;
```

```
1  double dSumInverse = 0.0;
2  int iNrValidInput = 0;
3  for(;;) {
4     std::cout << "Please enter a strictly
       positive"
5               << " number or zero to stop: ";
6     double dInput;
7     std::cin >> dInput;
8
9     if(dInput == 0.0)
10       break;
11    else if(dInput < 0.0)
12       std::cout << "Ignoring negative
       number, "
13                 << "please try again."
14                 << std::endl;
15    else {
16       dSumInverse += 1.0 / dInput;
17       ++iNrValidInput;
18    }
19 }
20
21
22 if(iNrValidInput)
23    std::cout << "The harmonic mean of the "
24              << "numbers you entered is: "
25              << iNrValidInput / dSumInverse
26              << std::endl;
```

The `do-while`-statement

Although the **`for`**-statement allows you to build any kind of loop you like, C++ has two additional loop statements, the **`do-while`**-statement and the **`while`**-statement. C++ programmers preferentially use them for loops that simply need to be iterated for as long as some condition is true. They often don't have an explicit counter variable.

The **`while`**-statement is the simplest of the two; it consists of

1 the keyword **`while`**,

2 a pair of brackets `()` that enclose a control expression, and

3 the body of the loop, which can be either a single simple statement or a compound statement.

Code listing 5.5 illustrates a clever application of the **`while`**-statement in a number-guessing game.

Code listing 5.5: The while-statement

```
#include <iostream>

int main()
{
    std::cout   << "Let's play a game:" << std::endl
                << "Think of a secret number (1 to 1000), and I will guess what it is." <<
    std::endl;

    //bracket the secret number based on a series of yes/no questions
    int iGuessLo = 0, iGuessHi = 1000;
    while(iGuessHi - iGuessLo != 1) {
        //propose guess midway between upper and lower bound
        const int iGuess = (iGuessLo + iGuessHi) / 2;

        std::cout   << std::endl << "Is the secret number larger than " << iGuess << "?"
                    << std::endl << "0 - No, it is smaller or equal"
                    << std::endl << "1 - Yes, it is larger"
                    << std::endl << "Your answer : ";
        bool isLarger;
        std::cin >> isLarger;

        if(isLarger)
            iGuessLo = iGuess;  //adjust lower bound
        else
            iGuessHi = iGuess;  //adjust upper bound

    }

    //show answer and terminate program
    std::cout << std::endl << "Your secret number is: " << iGuessHi << std::endl;
    return 0;
}
```

When execution reaches a **`while`**-statement, the program first checks the value of the control expression. The loop is terminated if the loop condition is **`false`**, and the program will then continue with the statement following the loop. If the condition is satisfied, the body of the loop is executed once and the control expression is re-evaluated. This process then continues until the loop condition is violated at some point. Note that the body of a **`while`**-loop will only ever be executed if the loop condition is satisfied initially.

The **`do-while`**-statement looks similar to the **`while`**-statement, except that the loop condition is placed at the end. Specifically, a **`do-while`**-loop:

1 starts with the keyword **do**, followed by

2 a single statement or compound statement that forms the body of the loop;

3 after that comes the keyword **while**, followed by

4 a pair of brackets () enclosing the control expression that determines whether the body will be executed another time, and

5 a semicolon to close off the whole statement.

Because the loop condition comes after the body of the loop, a **do-while**-loop will always iterate at least once before the control expression is evaluated (unlike the **while**-loop). Listing 5.6 shows an example of a **do-while**-loop that forces the user to input a number between 0 and 10. Compare this code fragment to the one on page 51 to see how the same problem is solved with a **for**-statement.

Code listing 5.6: The do-while-statement

```
1 int iInt;
2 do {
3     std::cout << "Please enter a number between 0 and 10: ";
4     std::cin >> iInt;
5 } while(iInt < 0 || iInt > 10);
```

More exercises

5.7. All loop statements lead to Rome. Copy the code from exercise 5.5 (complete the questions if you have not done so already). Next, replace the inner **for**-loop by a **while**-loop. After verifying that the program still works as intended, replace the outer **for**-loop by a **do-while**-loop.

5.4 Solutions to the exercises

Exercise 5.1. Find the bugs. The `for`-loop has an empty body, due to the semicolon at the end of line 1, and the `if`-statement on line 4 is controlled by an assignment, rather than by a test for equality. The corrected code is:

```
1 for(int i = 1; i <= iNumber; ++i)
2 {
3     int j = iNumber % i;
4     if(j == 0)
5         std::cout << iNumber << " is divisable by " << i << std::endl;
6 }
```

Exercise 5.2. The Ricker model of population growth. In the Ricker model, a small population can grow if $r > 0$. When the intrinsic growth rate is positive but small, the population reaches a stable equilibrium density $N^* = 1$. For higher values of r, the equilibrium becomes unstable and the population density starts to show regular oscillations. As r is increased further, the period of these oscillations increases through a series of period doubling bifurcations. Eventually, the dynamic becomes chaotic.

```
1  /* This program iterates the Ricker model
2     for population growth.
3     Written on 06/09/2016 by Master Yoda*/
4
5  #include <iostream>
6  #include <vector>
7  #include <cmath>
8
9  int main()
10 {
11    // set parameters
12    const double   r     = 2.0;          // intrinsic growth rate
13    const double   N0    = 0.001;    // initial population density
14    const int      tEnd  = 100;          // simulation length
15
16    // prepare for simulation
17    std::vector<double> vec(tEnd + 1);
18    vec[0] = N0;
19
20    // iterate Ricker model
21    for (int t = 0; t < tEnd; ++t)
22       vec[t + 1] = vec[t] * exp(r * (1.0 - vec[t]));
23
24    // show output on screen
25    for (int t = 0; t <= tEnd; ++t)
26       std::cout << "N_" << t << " = " << vec[t] << std::endl;
27
28    // terminate program
29    return 0;
30 }
```

Exercise 5.3. Creating a histogram.

```
1    #include <iostream>
2    #include <vector>
3
4    int main() {
5       std::vector<int> vec = {1, 1, 1, 3, 5, 7, 7, 2, 2, 8, 9, 2, 2, 3, 4};
6       std::vector<int> histogram(10, 0);
7       for (int i = 0; i < vec.size(); ++i) {
```

```
8           histogram[vec[i]]++;
9        }
10
11       for(int i = 0; i < histogram.size(); ++i) {
12          std::cout << "number " << i << " occurred " << histogram[i] << " times" << std::endl;
13        }
14    }
```

Exercise 5.4. Estimating rates of mutation.

```
1  /* This program estimates the relative rates
2     of different types of mutations from data.
3     Written on 06/09/2016 by Master Yoda*/
4
5  #include <iostream>
6  #include <vector>
7
8  int main()
9  {
10    const int n = 4;  //matrix dimension
11
12    // create 4 x 4 matrix of zeroes
13    std::vector<std::vector<double> > matrix(n, std::vector<double>(n, 0.0));
14
15    // set matrix elements
16    matrix[0][1] =  24.; matrix[0][2] =  26.; matrix[0][3] = 111.;
17    matrix[1][0] =  41.; matrix[1][2] =  91.; matrix[1][3] =  33.;
18    matrix[2][0] =  35.; matrix[2][1] = 114.; matrix[2][3] =  34.;
19    matrix[3][0] =  93.; matrix[3][1] =  33.; matrix[3][2] =  27.;
20
21    // combine upper and lower diagonal half, accumulate sum
22    double dSum = 0.0;
23    for (int i = 0; i < n - 1; ++i)
24       for (int j = i + 1; j < n; ++j) {
25          dSum += matrix[i][j] += matrix[j][i];
26          matrix[j][i] = 0.0;
27       }
28
29    // normalise rates
30    for (int i = 0; i < n - 1; ++i)
31       for (int j = i + 1; j < n; ++j)
32          matrix[i][j] /= dSum;
33
34    // compute transition rate
35    const double dTransitionRate = matrix[0][3] + matrix[1][2];
36
37    // show output
38    std::cout << "relative mutation rates:" << std::endl
39       << "AT -> TA : " << matrix[0][1] << std::endl
40       << "AT -> CG : " << matrix[0][2] << std::endl
41       << "AT -> GC : " << matrix[0][3] << std::endl
42       << "TA -> CG : " << matrix[1][2] << std::endl
43       << "TA -> GC : " << matrix[1][3] << std::endl
44       << "CG -> GC : " << matrix[2][3] << std::endl
45       << std::endl
46       << "transition rate   : " << dTransitionRate << std::endl
47       << "transversion rate : " << 1.0 - dTransitionRate << std::endl;
48
49    // terminate program
50    return 0;
51 }
```

You can also verify your own solution by comparing its output against the output of the program above:

```
relative mutation rates:
AT -> TA : 0.0981873
AT -> CG : 0.092145
AT -> GC : 0.308157
TA -> CG : 0.309668
TA -> GC : 0.0996979
CG -> GC : 0.092145

transition rate   : 0.617825
transversion rate : 0.382175
```

Exercise 5.5. Nasty nested for-statements.

(a) The program lists all prime numbers from 2 to 1000.

(b) If `iFactor` is declared in the inner loop, it will exist only within that loop. As a consequence, it is no longer possible to use `iFactor` in the statement on line 9, which lies outside the scope of the inner loop. The problem is solved by declaring `iFactor` in the outer **`for`**-loop, thus extending its scope from line 5 to 11. Now `iFactor` still exists after the inner loop has terminated.

(c) If `iFactor` is a factor of `iPrimeCandidate`, then the division `iPrimeCandidate / iFactor` has a remainder of zero, so that `iPrimeCandidate % iFactor` evaluates to zero. Given that zero is interpreted as the boolean value **`false`**, the expression `!(iPrimeCandidate % iFactor)` evaluates to **`true`**, if `Factor` is a factor of `iPrimeCandidate`. In other words, the execution of the **`break`** is triggered if `Factor` is a divisor of `iPrimeCandidate` .

(d) Any number c that is not prime can be written in the form $c = a \times b$, for at least one pair of integer factors a and b, where $a \leq b$. The condition $a \leq b$ implies that $c = a \times b \geq a \times a$, such that $a \leq c/a$. In other words, to determine if c is a prime number it is sufficient to check whether it can be divided by factors a that are no larger than c/a.

(e) The body of the inner **`for`**-loop (with the **`break`**-statement in it) is executed only if `iFactor <= (iPrimeCandidate / iFactor)`. Hence, the expression `iFactor > (iPrimeCandidate / iFactor)` on line 9 always evaluates to **`false`** if the inner loop terminated because of the **`break`** statement. By contrast, when the loop terminates as a result of the condition on line 6, this can only be because `!(iFactor <= (iPrimeCandidate / iFactor))`, necessarily implying that `iFactor > (iPrimeCandidate / iFactor)` evaluates to **`true`**.

Exercise 5.7. All loop statements lead to Rome.

```
1  #include <iostream>
2
3  int main()
4  {
5      int iPrimeCandidate = 2;
6      do {
7          int iFactor = 2;
8          while(iFactor <= (iPrimeCandidate / iFactor)) {
9              if(!(iPrimeCandidate % iFactor))
10                 break;
11             ++iFactor;
12         }
13         if(iFactor > (iPrimeCandidate / iFactor))
14             std::cout << iPrimeCandidate << " is prime" << std::endl;
15         ++iPrimeCandidate;
16     } while(iPrimeCandidate < 1000);
17
18     return 0;
19 }
```

Module 6

Functions

Jamie explains what's on the menu

In this module you will:

- Write your first C++ functions.
- Meet the weirdest C++ data type, **`void`**, which represents an empty value
- Learn to apply two different ways for passing input arguments to functions

6.1 What functions are good for

In previous modules, you have seen that compound statements subdivide program code into different blocks, each with their own set of local variables. Expressions in one block cannot access local variables in another code block, allowing you to use variables with the same name at different places. Without this possibility, you would have to invent a new name for every loop variable you use in a **`for`**-loop, or work with global loop variables (which would both make your life as a programmer miserable).

In this module, you will learn how to take program modularity to the next level by defining your own functions. A C++ function is a special type of compound statement that can take user-defined input and that returns the result of executing the statements within its body. Functions provide a way to indicate the main structure in a C++ program, much like you would use chapter and section headings to structure normal written text. They also save you from having to write repetitive pieces of code, where a set of identical instructions is applied to different input values. Last but not least, functions allow you to compact often-used sequences of general program code. As an example, consider Master Yoda's favourite home-grown function, `dot()`, which he uses to track down which pieces of code are responsible for a run-time error:

```
1 #include<iostream>
2
3 void dot()
4 {
5    std::cout << ".";
6 }
```

```
7
8  /* defininitions of other functions [ initialise(), doSimulation(), saveData() ] are not shown */
9
10 int main()
11 {
12    initialise();
13    dot();
14    doSimulation();
15    dot();
16    saveData();
17    dot();
18    return 0;
19 }
```

As you can probably see from its definition on line 3-6, the `dot()` function simply writes a dot to the console output. The function is called on line 13, 15 and 16, where you see the name of the function, followed by the function call operator. In general, this is a pair of brackets containing the input arguments. The function `dot()` does not take any arguments, so the function call brackets are empty. Now, if Master Yoda sees two dots appearing on the screen before the program stops working, he knows that the error is in the function `saveData()`. Next, he will remove the calls to `dot()` from `main()` and insert some new ones in the body of the function `saveData()` to localise where the error is exactly.

6.2 Function definitions

Let us take a closer look at the definition of the function `dot()`. Superficially, its structure looks similar to that of the function `main()`: in both cases the name of the function is followed by a pair of brackets, followed by a compound statement that contains the body of the function. Also, the name of the function is preceded in both cases by the name of a C++ type that specifies the type of the value returned by the function. The function `main()` returns a value of the type **`int`** (the instruction for returning a value is the **`return`** statement on line 18); the return type associated with the function `dot()` is **`void`**, which means that the function returns an empty value (i.e., it returns 'nothing'). A **`void`**-function will return automatically when the end of its body is reached, so it does not need to contain a **`return`**; statement, unless it is necessary to allow for a second way of exiting from the function. All other functions need to have at least one **`return`** statement, which is generally positioned at the end of the function body. You are not obliged to use the value returned by a function for anything, or to assign it to a variable. For example, it is perfectly valid (but pointless) to write: `for(double x = 0.0; x < 1.0; x += 0.05) sin(x);`

***Master Yoda* on `void` functions**

'What is the use of a function that does not return anything?', you may wonder. Well, typically **`void`**-functions are called because they *do* something useful (other than producing a return-value). This could be writing information to the screen, storing data in a file or executing a sequence of steps in a complex algorithm. **`void`**-functions are the main reason that the data type **`void`** exists. Every function in C++ must have a return type, so without **`void`** it would be impossible to create functions that return an empty value. Empty values cannot exist as actual objects, however, so it is not allowed in C++ to declare variables of the type **`void`**, or to use void functions as elements in expressions, such as in **`int`** `i = 1 + dot();`.

The definitions of `dot()` and `main()`, already indicate the general form of C++ function definitions, except that they are both functions without arguments. The following example shows how to declare functions that accept input arguments. The program relies on an ancient method for computing the greatest common divisor of two numbers to write fractions in a simplified form.

```
1 #include <iostream>
2
3 int getGCD(int iNr1, int iNr2)
4 // Euclid's algorithm for computing the
5 // greatest common divisor of two numbers iNr1 and iNr2
6 {
```

```
7       while(iNr2 != 0) {
8           int tmp = iNr2;
9           iNr2 = iNr1 % iNr2;
10          iNr1 = tmp;
11      }
12      return iNr1;
13  }
14
15  void simplify(int iNum, int iDenom)
16  // writes the fraction iNum / iDenom in a simplified form
17  {
18      std::cout << iNum << "/" << iDenom << " = ";
19
20      //divide numerator and denominator by their greatest common divisor
21      const int iGCD = getGCD(iNum, iDenom);
22      iNum /= iGCD;
23      iDenom /= iGCD;
24
25      if(iNum / iDenom)
26          std::cout << (iNum / iDenom) << " + ";
27      std::cout << (iNum % iDenom) << "/" << iDenom << std::endl;
28  }
29
30  int main()
31  {
32      simplify(6, 5);
33      simplify(57, 323);
34      simplify(52455, 34970);
35
36      //terminate program
37      return 0;
38  }
```

The output of the program is:

```
6/5 = 1 + 1/5
57/323 = 3/17
52455/34970 = 1 + 1/2
```

The key points to focus on first are the definitions of the two functions on line 3 and 15. Notice that input parameters are declared between the brackets that follow the function name. Each input is specified by a type and a name; if there are multiple input parameters, their declarations are separated by a comma. It is necessary to repeat the type name if multiple input parameters have the same type, unlike in a normal declaration statement, where you could write `int iNr1, iNr2;` to declare two integers, for example. Other than that, the function parameters behave exactly as if they were local variables declared at the start of the function body: they are created when the function is called and initialised with the values of the arguments provided at the function call. Their scope is restricted to the function body, so the function parameters are destroyed once the function returns.

Function definitions cannot be nested (you cannot define a function within the body of another function), but your functions are allowed to call other functions that you wrote (as evidenced by the function call on line 21). A function may even call itself, making it possible to easily implement recursive methods for solving certain kinds of computational problems.

The order in which the functions `getGCD()`, `simplify()` and `main()` have been defined is not arbitrary. Just as for variables, the name of a function can only be used after it has been declared. Therefore, since `getGCD()` is called in `simplify()`, its definition comes before `simplify()`. Likewise, `simplify()` is called in `main()`, so its definition comes before `main()`. One thing to keep in mind, however, is that the *declaration* of a function, need not be accompanied by a *definition* of a function body. In fact, functions can be declared as often as you like, but there has to be exactly one function definition.

To declare a function, it is sufficient to give the function prototype. This consists of the first part of the function definition, i.e., the part that comes before the body of the function. Including the argument names in the prototype is optional, but you do need to list their types. Listing 6.1 illustrates how the use of function prototypes allows you to reverse the order of the function definitions relative to what it was previously. Flexibility in where to put the function definitions might not seem that important right now, but when your programs become bigger, function declarations allow you to distribute your code over multiple files and develop your own function libraries.

The rules for the scope of variables also apply to function declarations: the function prototype of `simplify()` on line 3 of listing 6.1 serves as a global declaration, whereas the prototype on line 19 declares `getGCD()` within the scope of `simplify()`. As a result, the compiler would not recognise a call to `getGCD()` in code that is inserted between the end of `simplify()` (line 31) and the definition of `getGCD()` on line 41.

Code listing 6.1: Function prototypes

```
1  #include <iostream>
2
3  void simplify(int, int);     // global declaration: simplify() is
4                                  // recognised from this point onwards
5
6  int main()
7  {
8      simplify(6, 5);
9      simplify(57, 323);
10     simplify(52455, 34970);
11
12     // terminate program
13     return 0;
14 }
15
16 void simplify(int iNum, int iDenom)
17 // writes the fraction iNum / iDenom in a simplified form
18 {
19     int getGCD(int, int);              // declares getGCD() within the scope of simplify()
20
21     std::cout << iNum << "/" << iDenom << " = ";
22
23     // divide numerator and denominator by their greatest common divisor
24     const int iGCD = getGCD(iNum, iDenom);
25     iNum /= iGCD;
26     iDenom /= iGCD;
27
28     if(iNum / iDenom)
29         std::cout << (iNum / iDenom) << " + ";
30     std::cout << (iNum % iDenom) << "/" << iDenom << std::endl;
31 }
32
33 /* the function getGCD() is not recognised here
34 void test()
35 {
36     getGCD(1,2);                       // error: use of undeclared identifier `getGCD'
37 } */
38
39 int getGCD(int iNr1, int iNr2)          // global declaration: getGCD()
40                                        // is recognised from this point onwards
41 // Euclid's algorithm for computing the
42 // greatest common divisor of two numbers iNr1 and iNr2
43 {
44     while(iNr2 != 0) {
45         int tmp = iNr2;
46         iNr2 = iNr1 % iNr2;
47         iNr1 = tmp;
48     }
49     return iNr1;
50 }
```

6.3 Exercises

6.1. Scope of a function. Consider the following code, what is the value for variables a, b, c, d and e?

```
1
2 // the following function adds x and y, and adds one.
3 int sum(int x, int y) {
4    int b = 1;
5    int answer = b + x + y;
6    return answer;
7 }
8
9 int a = 7;
10 int b = 3;
11 int c = sum(a, b);
12 int d = sum(b, c);
13 int e = sum(d, a);
```

6.2. Defining a function. Go back to exercise 5.2 and write a function `computeNextPopulationDensity()` that calculates the population density N_t as a function of the density N_{t-1} in the previous generation. Incorporate your function in your previous implementation of the Ricker model.

6.3. Finding the largest element in a vector. Write a function that takes a vector of floating-point variables as input and that returns its largest element.

6.4. Counting the number of matches. Write a function that takes a vector of integer variables as input and counts the number of negative entries in the vector. Can you extend the function to count the number of any type of entry?

Test your skill

6.5. Create a calculator ($\star$). Write a simple calculator program that performs basic arithmetic operations. Write a program that asks the user for an operation, and the input of two numbers. Include the following operations: addition, subtraction, multiplication and division. Example output is shown in listing 6.2. To do so, follow the following steps:

- 1. Implement four separate functions for addition, subtraction, multiplication, and division.
- 2. Each function should take two parameters (operands) and return the result of the respective operation.
- 3. Display a menu to the user with options to choose the operation (addition, subtraction, multiplication, division, or exit).
- 4. Get user input for the chosen operation and operands.
- 5. Use if statements to call the appropriate function based on the user's choice.
- 6. Display the result to the user.

Code listing 6.2: Example output of the calculator program

```
1 Simple Calculator
2
3 1. Addition
4 2. Subtraction
5 3. Multiplication
6 4. Division
7 5. Exit
8
```

```
Enter your choice: 1
Enter number  1: 5
Enter number 2: 3

Result: 8
```

6.4 Solutions to the exercises

Exercise 6.1. Scope of a function. $a = 7$, $b = 3$, $c = 11$, $d = 15$, $e = 23$

Exercise 6.2. Defining a function.

```
/* This program iterates the Ricker model
   for population growth.
   Written on 13/09/2016 by Master Yoda*/

#include <iostream>
#include <vector>
#include <cmath>

double computeNextPopulationDensity(double N, const double r)
{
    return N * exp(r * (1.0 - N));
}

int main()
{
    // set parameters
    const double  r     = 2.0;          // intrinsic growth rate
    const double  N0    = 0.001;    // initial population density
    const int     tEnd  = 100;          // simulation length

    // prepare for simulation
    std::vector<double> vec(tEnd + 1);
    vec[0] = N0;

    // iterate Ricker model
    for (int t = 0; t < tEnd; ++t) {
        vec[t + 1] = computeNextPopulationDensity(vec[t], r);
    }

    // show output on screen
    for (int t = 0; t <= tEnd; ++t) {
        std::cout << "N_" << t << " = " << vec[t] << std::endl;
    }
    // terminate program
    return 0;
}
```

Exercise 6.3. Finding the largest element in a vector.

```
double findLargest(std::vector<double> vec)
{
    double dLargest = vec[0];
    for(int i = 0; i < vec.size(); ++i) {
        if(vec[i] > dLargest) {
            dLargest = vec[i];
        }
    }
    return dLargest;
```

```
}
```

Exercise 6.4. Counting the number of matches.

```
int count_entries(std::vector<int> vec, int pattern)
{
   int count = 0;
   for(int i = 0; i < vec.size(); ++i) {
      if(vec[i] == pattern) {
         count++;
      }
   }
   return count;
}

```

Module 7

Matters of style

Master Yoda
offers a preview

This module, contributed by our expert panel member Nefertiti, contains a collection of her style recommendations from different parts of the manual. It provides comprehensive guidelines for developing a consistent and professional programming style. Scan through the chapter once in its entirety, without paying too much attention to elements of the language you have not encountered yet. Then, return to it frequently at later points to consult it as a reference manual.

7.1 Naming conventions

If you stick to a consistent naming convention, readers will be able to infer immediately whether something is a type, a variable, a function, a global constant, and so on, without having to search for the declaration of that entity. Use descriptive names that make your code immediately understandable by a new reader; avoid unfamiliar abbreviations and do not abbreviate by deleting letters within a word.

Variables

There are generally two naming conventions followed, which are both clear and concise. camelCase notation uses capital letters to indicate the start of a new word when variable names consist of compound names. snake_case notation instead does explicitly not use capital characters, but uses the underscore charcater (_) to indicate seperation of words in variable (and function) names.

Use descriptive and clear names for important variables, or variables that have a large scope. Feel free to use short names for temporary variables with a restricted scope and for loop counters that are accessed many times. Describe the type of the variable in the name, or indicate this by a letter code at the start of the name. For example, use

`c` for characters, e.g., `char cLetter = 'a';`

`d` for double-precision floating-point variables, e.g., `double dWeight = 0.82;`

`f` for single-precision floating-point variables, e.g., `float fHeight = 1.96f;`

`i` for integer variables, e.g., `int iNumLegs = 6;` (OK, because num is a familiar abbreviation)

`l` for long integer variables, e.g., `long int lHugeNumber = 1l << 32;`

`ld` for a `long double`, e.g., `long double ldSum = 0.0ld;`

`vec` for a `std::vector`, e.g., `std::vector<double> vecConcentration(5, 0.0);`

`lst` for a `std::list`, e.g., `std::list<int> lstSequence;`

`is`, `has` or another verb that starts a yes/no question for a `bool`, e.g., `bool hasConverged = false;`

These can be combined with additional indicators to highlight special kinds of variables:

`k` for global constants and other constant expressions that exist for the entire duration of the program.

`m` for a member variable of a class object.

Functions

Function names are in mixed case, starting with a lowercase letter. The first word of a function name is a verb that describes what the function does. For example, `findNextPrimeNumber()`, `analyseData()` or `iterateLifeCycle()`. Boolean functions may follow the style for boolean variables.

Type names

User-defined type names are in mixed case, starting with an uppercase letter, such as in `class Individual`, `struct DataBaseItem`, `enum Colours` and `typedef std::vector<int> IntVector`.

File names

See section 7.5.

Namespaces

User-defined namespaces have short names in all lowercase, e.g., `namespace rnd;`

7.2 Formatting

Since C++ is a free-form language, the compiler does not care about how you format your code, but a human reader will. If your code is messy, no one will be able to understand it, so consider the following guidelines to keep your code tidy.

Line length

Tradition prescribes that each line of code should be no more than 80 characters long (excluding comments). Use indentation if you have to break long expressions across lines, as in

```
1 double dReactionRate = dMaxRate * dEnzymeConcentration *
2     dSubstrateConcentration / (dKm + dSubstrateConcentration);
```

An operator connecting two parts of a long expression is placed with the first part, as the last character on the line. Long output operations are also best split across different lines, using the output operators as alignment points. Long string literals can be split and will automatically be concatenated:

```
1 std::cout   << "Variable dVar has value " << dVal
2             << " in member function "
3                " myVeryLongClassName::myVeryLongFunctionName()"
4             << std::endl;
```

Indentation

Use indentation to clarify which instructions belong to a program-flow statement, such as **if**, **else**, **switch**, **for**, **do** and **while**. The style used throughout this manual is known as Stroustrup style, named after Bjarne Stroustrup, the creator of C++ who uses the same indent style in his book *The C++ Programming Language*. Alternatively, you can also follow the Google programming style guide (https://google.github.io/styleguide/cppguide.html). It does not matter which one you follow, as long as you are consistent. An example of the Stroustrup style is:

```
if(i % 2 == 0) {            // opening brace of the compound statement directly after the control expression
    std::cout << i << " is even" << std::endl;
    i /= 2;                 // body of the compound statement indented relative to control expression
}                           // closing brace on separate line, aligned with the control expression
else
    doSomethingOdd(i);      // single statements are indented and put on the next line
```

In Module 11, you will learn about C++ classes. Class definitions are formatted like this:

```
class Insect : public Animal {
public:
    Insect();                                                   // construct an Insect
    ~Insect();                                                  // kill an Insect
    int getLifeHistoryStage() {return mLifeHistoryStage;}
private:
    int mLifeHistoryStage;
};
```

i.e., without indenting the labels **public**:, **protected**:, and **private**:. The opening brace of the function body in a function definition comes on the line after the function head, which contains the return type, the name of the function and the argument list.

```
double pow(const double &x, const int &n)
// returns the value of x ^ n, overloads cmath's pow(double&, double&)
{
    if(n < 0)
        return 1.0 / pow(x, -n); // reformulate for negative exponents

    double out = 1.0;
    for(int i = 0; i < n; ++i)    // compute n-th power of x by iteration
        out *= x;

    return out;
}
```

Spaces

Binary operators typically have spaces around them, as in `i = 0;` and `double x = 1.0 / (1.0 + z);`. Unary operators are not separated from their arguments by spaces, like in `x = -3`, `++i` and `if(!isNight)`. Note that, in the previous examples, there are also no spaces before and after expressions within brackets, braces or angle brackets. In nested template types, it is important to include a space between the <> enclosing nested template arguments. Otherwise, some compilers may confuse the subsequent >> with an input or bitshift operator. For example, a disambiguation space has been inserted in:

```
std::vector<std::vector<int> > matrix;
```

7.3 Comments

Comment style

Be generous with comments. Avoid the /* ... */ syntax, except at the start of your file, where you can use it to include a large comment block with information about the author of the program, the program version, a description of the program and instructions for compiling and running the code. If you restrict yourself to the // syntax within your program, you can still comment out large stretches of code easily by including them between /* ... */. This can be extremely useful when you are still developing your program.

Function comments

Function declarations should be accompanied by a comment describing how to use the function, if this is not obvious. This often includes a description of what the inputs and outputs are, and whether there any restrictions on their range. Function definitions are accompanied by a comment that describes what the function does.

Global variables and constants

Global variables and constants almost always deserve a comment to describe what they are (the reason for them being global is probably that they are used at various points throughout the code, so it is important to clarify what they are used for).

Implementation comments

Tricky pieces of code benefit from comments that explain what happens in pseudocode, as in the following example

```
1 bool isPrime(const int &iNumber)
2 // returns true if iNumber is a prime number
3 {
4     for(int iFactor = 2; iFactor < iNumber; ++iFactor)  // consider all potential divisors
5         if(iNumber % iFactor == 0)  // if iNumber is divisable by iFactor...
6             return false;           // ...then iNumber is not prime, so return false
7     // if program flow reaches this point, iNumber is not divisable by any iFactor...
8     return true;  //...implying that iNumber is prime
9 }
```

If you use comments that span several lines, or that connect blocks of an **if-else**-statement, for example, line them up to improve readability.

7.4 Functions

Order of function arguments

Sort function arguments in their order of importance, and preferably list inputs before outputs. Arguments with default values always need to come last.

Default function arguments and overloading

Use default function arguments only if the default value really reflects a natural default behaviour for a function. For example, a function that creates a file if it received a filename as input and that deletes all files in a given directory if no argument is provided, is a sure road to disaster. Similarly, use function overloading only to create a related set of functions that perform conceptually similar operations on different kinds of variables. The thing to ask yourself is: 'Would a naive user be able to use a set of overloaded functions without realising exactly which of the versions is being called?'.

7.5 Distributing code over multiple files

File naming conventions

Filenames are in lowercase and may contain underscores to separate words, as in`parameter_file.txt`. Use the same name for a class header file (`my_class.hpp`) and the corresponding implementation file (`my_class.cpp`).

Function prototypes

If you want to move the definition of a function to another file, create a header file with the function prototype, and a separate `.cpp` file for the function definition (functions should normally not be defined inside a header file). List only the argument types, not their names, in the function prototype. For example, a header file `my_useful_functions.hpp` may contain the prototype

Code listing 7.1: my_useful_functions.hpp

```
double computeSomethingUseful(const double&);
```

The actual definition of the function is placed in a file `my_useful_functions.cpp`, and may look as follows:

Code listing 7.2: my_useful_functions.cpp

```
double computeSomethingUseful(const double &dInputVariable)
{
   double dResult = dInputVariable + 1.0;

   // some additional useful operations ...

   return dResult;
}
```

To access the function from another file, include the header:

Code listing 7.3: main.cpp

```
#include "my_useful_functions.hpp"
#include <iostream>

int main()
{
      std::cout << computeSomethingUseful(1.0) << std::endl;
      return 0;
}
```

Header files

All code in header files should be prefaced with `#pragma once` to prevent it from being included multiple times. The syntax looks like this:

Code listing 7.4: my_header.hpp

```
#pragma once

// ...put the code here...
```

Functions with default arguments

The values of default function arguments can be specified only once. For functions that are declared multiple times, specify the default argument in the declaration that is encountered first (typically, a declaration in a header file).

Module 8

Functions continued

Jamie explains what's on the menu

In this module you will:

- Be introduced to multiple ways of passing arguments to functions.
- Learn to write recursive functions
- Learn to write multiple versions of the same function

Functions are an important way of compartmentalizating your code, which can be very helpful in tracking down bugs, organizing your code structure and avoiding repetitive code. Functions are therefore an important tool to learn and master, but may take some extra getting used to. In this module we will explore functions further, and dive deeper into how you can use functions, and how you can use more advanced techniques with functions.

Two ways of passing arguments to functions

In general, there are two different ways that a computer language can pass an argument to a function. One way is to pass arguments *by value*. This means that the value of a function argument is copied into a function parameter that exists as a separate variable while the function is being executed. So far, we have only looked at functions that pass their arguments by value. In fact, passing arguments by value is the default convention in C++, because of safety reasons: when arguments are passed by value, the variables used to call a function will remain unaffected by operations on their temporary copies within the function body. That this method also has limitations is illustrated by the following failed attempt to write a function to swap the values of two variables:

```
1  /* This program is supposed to swap two values
2  but does not work as intended.
3  Written on Friday the 13th by J. Oliver */
4
5  #include <iostream>
6
7  void swap(double x, double y)
8  {
```

```
9       // show values
10      std::cout << "in swap() : x = " << x << ", y = " << y << std::endl;
11
12      // swap values
13      const double tmp = x;
14      x = y;
15      y = tmp;
16
17      // show values again
18      std::cout << "in swap() : x = " << x << ", y = " << y << std::endl;
19   }
20
21   int main()
22   {
23      double x = 1.0, y = 2.0;
24
25      // show values
26      std::cout << "in main() : x = " << x << ", y = " << y << std::endl;
27
28      // call swap() to swap values
29      swap(x, y);
30
31      // show values again
32      std::cout << "in main() : x = " << x << ", y = " << y << std::endl;
33
34      // terminate program
35      return 0;
36   }
```

The result is not what the programmer intended, as you can see from the output:

```
in main() : x = 1, y = 2
in swap() : x = 1, y = 2
in swap() : x = 2, y = 1
in main() : x = 1, y = 2
```

Apparently, the function parameters `x` and `y` are swapped inside the body of the function, but this operation has not affected the values of the `x` and `y` from the function `main()` that were used to call the function.

To understand why passing arguments by value is inadequate in the above program, it is important to realise that the `x` and `y` declared on line 23 are not the same variables as the function parameters `x` and `y` that appear on line 7. Despite having the same name, the two pairs of variables are stored at different positions in the memory. To give you a better feeling of what happens when the function `swap()` is executed, let us focus on the variable `x`, and refer to its two instances as $\texttt{x}_{\texttt{main}}$ (the `x` declared on line 23) and $\texttt{x}_{\texttt{swap}}$ (the `x` of line 7). Now let us keep track of what happens to these variables as we go through the code in the order of execution:

(1) Program execution starts at line 21. When it reaches the declaration statement on line 23, the variable $\texttt{x}_{\texttt{main}}$ is created and initialised with the value 1.0.

(2) The function `swap()` is called on line 29. The compiler now creates the variable $\texttt{x}_{\texttt{swap}}$ and allocates memory to store its value. The value of $\texttt{x}_{\texttt{main}}$ is passed to the function and used to initialise $\texttt{x}_{\texttt{swap}}$.

(3) When program execution enters the body of the function `swap()`, $\texttt{x}_{\texttt{swap}}$ has value 1.0 and $\texttt{y}_{\texttt{swap}}$ has value 2.0. The two values are exchanged on line 13-15 of the code, so $\texttt{x}_{\texttt{swap}}$ has value 2.0 by the time the output statement on line 18 is reached. None of this affects the value of $\texttt{x}_{\texttt{main}}$ in any way.

(4) When the function `swap()` returns on line 19, $\texttt{x}_{\texttt{swap}}$ is erased from the memory. The variable $\texttt{x}_{\texttt{main}}$ and its value remain unaffected. Program execution then jumps back to the function `main()` and continues with the statement on line 32.

In summary, the value of $\texttt{x}_{\texttt{main}}$ is copied into a temporary variable $\texttt{x}_{\texttt{swap}}$ that is used locally within the function, and whatever happens to the temporary variable $\texttt{x}_{\texttt{swap}}$ has no effect on $\texttt{x}_{\texttt{main}}$.

It follows then that functions whose arguments are passed by value can only influence variables outside their own scope via their return value. Moreover, given that only one value can be returned, such functions can only change the value of a single variable at a time. So, how can we implement cases like the `swap()`

routine above, where we would like a function to change multiple variables? The solution is to pass the function arguments *by reference*, which is the second general approach to argument passing.

Switching from passing arguments *by value* to passing them *by reference* requires only a small change in the first line of the function definition. Simply replace

`void swap(double x, double y)` (which passes `x` and `y` by value),

by

`void swap(double &x, double &y)`

in order to pass `x` and `y` by reference. The inserted `&` between the type name and the name of the variable will instruct the compiler to treat the function parameter as an alias (or alternative name) for the variable that is passed to the function. This prevents the creation of a temporary copy to hold the value of the function argument. Things like `&x` and `&y` that behave as alternative names for other variables are called *references* in C++. Operations on a reference do not affect the reference itself, but the variable it is referring to. In other words, when a reference parameter of a function is changed within the body of the function, that change will directly affect the variable used to call the function. To verify this by yourself, change the `swap()` from the code listing above into a function with two reference parameters and inspect the console output. You will see that the modified `swap()` function works exactly as intended.

One last thing: whenever a function argument is passed by reference, this should not only be specified in the function definition, but in all other function declarations as well. If the function prototype is written without the names of the variables, it is common to glue the `&` directly behind the type name. In particular,

`void swap(double&, double&);`

and

`void swap(double &, double &);`

are both valid declarations for the (modified) `swap()` function.

const-references

Passing function arguments by reference is popular among C++ programmers, because it prevents function arguments from being copied into a temporary variable. For functions that operate on large objects, unnecessary copying takes a lot of time, so passing by reference is an easy option to speed up the program. The downside is that passing by reference is less safe, because it opens up the possibility that a function inadvertently changes an object outside its own scope. This makes it much more difficult to find bugs: in addition to checking anything that happens to the corrupted variable itself, you also need to examine all operations inside functions that involve references to that variable. Luckily, there is a safety device: declare the function parameter as a **`const`** reference. This will prevent a function argument from being copied and prohibits the function from changing its value through the reference. Listing 8.1 illustrates the use of reference and **`const`**-reference function arguments in two functions that operate on (potentially) large vector objects. The first one only needs read access to the vector, so its argument is passed as a **`const`**-reference. The second function sorts its input vector, so here the argument is passed by reference.

***Nefertiti* provides some guidelines**

Notice that you cannot see from the way the functions are called (on line 43 and 49 of listing 8.1) that `sort()` changes the vector `v`, whereas `accumulate` does not. An experienced programmer, however, will infer this from the function prototypes (even without seeing the function bodies). For this reason, it is a good idea to stick to the following convention:

- If a function just needs read-access to its input variable, pass the argument by **`const`**-reference.
- If a function is intended to change the variable used to call the function, pass the argument by reference.
- Pass the argument by value *only* if the function needs write-access to its input parameter *and* when it is not meant to change the variable used to call the function.

Function overloading

C++ compilers recognise functions by the combination of their name and their argument list. This creates the possibility for multiple functions to share the same name, as long as the number or the type

Code listing 8.1: Calling functions by reference and const-reference

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

double accumulate(const std::vector<double> &vec)
// returns the sum of the values in vec
{
   double sum = 0.0;
   for(int i = 0; i < vec.size(); ++i)
   sum += vec[i];
   return sum;
}

void sort(std::vector<double> &vec)
// sorts vec in ascending order
{
   void swap(double&, double&);
   for(int i = 0; i < vec.size() - 1; ++i)
   for(int j = i + 1; j < vec.size(); ++j)
   if(vec[j] < vec[i])
   swap(vec[i], vec[j]);
}

void swap(double &x, double &y)
// swaps the values of x and y
{
   const double tmp = x;
   x = y;
   y = tmp;
}

int main()
{
   // initialise vector
   std::vector<double> v;
   for(int i = 0; i < 5; ++i) v.push_back(5 - i);

   // show vector elements
   std::cout << "elements in v:";
   for(int i = 0; i < v.size(); ++i) std::cout << " " <<  v[i];
   std::cout << std::endl;

   // sort and show again
   sort(v);
   std::cout << "elements in v:";
   for(int i = 0; i < v.size(); ++i) std::cout << " " <<  v[i];
   std::cout << std::endl;

   // report sum of elements
   std::cout << "the vector elements sum up to: " << accumulate(v) << std::endl;

   // terminate program
   return 0;
}
```

of their arguments differ. It is common to use such so called *overloaded* functions to create a related set of functions that execute a similar task for arguments that differ in type. The programmer can then call these functions by their common name, and leave it up to the compiler to choose the correct implementation.

Below, function overloading is applied to create a related set of functions to compute the norm of integers, floating point numbers and two kinds of vectors. For single numbers, the norm is given by the absolute value; for floating point vectors, it is calculated as the length of the vector in Euclidean space, and for integer vectors, it is computed as the Hamming (city block) distance from the zero vector.

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

int norm(const int &x)
{
   return abs(x);
}

double norm(const double &x)
{
   return fabs(x);
}

double norm(const std::vector<double> &x)
{
   double dSumSq = 0.0;
   for(int i = 0; i < x.size(); ++i)
   dSumSq += x[i] * x[i];
   return sqrt(dSumSq);
}

int norm(const std::vector<int> &x)
{
   int iSum = 0;
   for(int i = 0; i < x.size(); ++i)
   iSum += abs(x[i]);
   return iSum;
}

int main()
{
   std::cout << "norm(-1)         = " << norm(-1) << std::endl;
   std::cout << "norm(-1.5)       = " << norm(-1.5) << std::endl;

   std::vector<int> ivec(2, 1);
   std::cout << "norm({1, 1})     = " << norm(ivec) << std::endl;

   std::vector<double> dvec(2, 1.0);
   std::cout << "norm({1.0, 1.0}) = " << norm(dvec) << std::endl;

   // terminate program
   return 0;
}
```

The output generated by the program is:

```
norm(-1)         = 1
norm(-1.5)       = 1.5
norm({1, 1})     = 2
norm({1.0, 1.0}) = 1.41421
```

Default function arguments

A shortcut to function overloading is provided by the option to specify a default value for one or more function arguments. A function can then be called without explicitly specifying this argument, in which

case the default value will automatically be substituted. Alternatively, if the function is called with an explicit argument, this will override the default value. Default arguments in function declarations look similar to the initialisation of a variable. The listing below illustrates an application of a default argument to create a flexible `prCoinFlip()` function. Its default behaviour is to return the probability of observing a given sequence of coin flips (passed to the function as a `std::vector<bool>`) assuming that the coin is fair. If a second argument is provided, the probability is computed for a (biased) coin with probability of observing Heads equal to`prHeads`.

```
1  #include <iostream>
2  #include <vector>
3
4  double prCoinFlip(const std::vector<bool> &isHeads, const double &prHeads = 0.5)
5  {
6     double dProb = 1.0;
7     for(int i = 0; i < isHeads.size(); ++i)
8     dProb *= (isHeads[i] ? prHeads : 1.0 - prHeads);
9     return dProb;
10 }
11
12 int main()
13 {
14    // create sequence Tail, Heads, Heads, Heads
15    std::vector<bool> isHeads(4, true);
16    isHeads[0] = false;
17
18    // compute probability of observing sequence for fair coin
19    std::cout << "The probability of observing {";
20       for(int i = 0; i < isHeads.size(); ++i) std::cout << (isHeads[i] ? "H" : "T");
21       std::cout << "} for a fair coin is equal to " << prCoinFlip(isHeads) << std::endl;
22
23    // compute probability of observing sequence for biased coin (prHeads = 0.6)
24    std::cout << "The probability of observing {";
25       for(int i = 0; i < isHeads.size(); ++i) std::cout << (isHeads[i] ? "H" : "T");
26       std::cout << "} for a biased coin is equal to " << prCoinFlip(isHeads, 0.6) <<
    std::endl;
27
28    // terminate program
29    return 0;
30 }
```

If you do not recognise the expressions involving `?:` operations on line 8, 20 and 25, take a few minutes to read the extra material on the conditional operator (p. 42). Two calls to the function `prCoinFlip` occur on line 21 and 26. The first is called with only one argument, so that `prHeads` is initialised with its default value; the second is called with two argument, causing the function to be executed with `prHeads` equal to 0.6.

An equivalent solution is to rely on overloaded functions to achieve the same goal. This has the added benefit that the function version for the fair coin can be simplified: line 13 illustrates an application of the bitshift operator to efficiently calculate powers of 2.

```
1  double prCoinFlip(const std::vector<bool> &isHeads, const double &prHeads)
2  // function specialisation for coin with arbitrary bias prHeads
3  {
4     double dProb = 1.0;
5     for(int i = 0; i < isHeads.size(); ++i)
6     dProb *= (isHeads[i] ? prHeads : 1.0 - prHeads);
7     return dProb;
8  }
9
10 double prCoinFlip(const std::vector<bool> &isHeads)
11 // function specialisation for a fair coin
12 {
13    return 1.0 / (1 << isHeads.size()); // = (1/2) ^ isHeads.size()
14 }
```

You can verify that both versions give identical output:

```
The probability of observing {THHH} for a fair coin is equal to 0.0625
The probability of observing {THHH} for a biased coin is equal to 0.0864
```

A few restrictions apply to default arguments: in functions that are declared multiple times, the default value must be specified only once, in the declaration that comes first. To avoid ambiguity, default arguments must also come last in the list of function parameters: if a particular parameter is set to be a default parameter, all following parameters must have default values as well.

8.1 Recursive functions

Recursive functions are subroutines that call themselves. Recursion provides an alternative for an iterative implementation of the same algorithm. Sometimes, the recursive implementation is more intuitive, or may result in much clearer code. However, the iterative version will usually execute a little bit faster, because it avoids a sequence of nested function calls. As an example, consider again Euclid's algorithm for finding the greatest common divisor of two numbers. Below are the two versions, the iterative one you have seen earlier on the left; the recursive implementation on the right.

```
1   int getGCD(int iNr1, int iNr2)
2   // iterative implementation of
3   // Euclid's algorithm for computing
4   // the greatest common divisor
5   // of two numbers iNr1 and iNr2
6   {
7      while(iNr2 != 0) {
8         int tmp = iNr2;
9         iNr2 = iNr1 % iNr2;
10        iNr1 = tmp;
11     }
12     return iNr1;
13  }
14
```

```
1   int getGCD(int iNr1, int iNr2)
2   // recursive implementation of
3   // Euclid's algorithm for computing
4   // the greatest common divisor
5   // of two numbers iNr1 and iNr2
6   {
7      return iNr2 == 0 ?
8      iNr1 : getGCD(iNr2, iNr1 % iNr2);
9   }
10
```

When the recursive function calls itself, on line 8, new copies of the function parameters `iNr1` and `iNr2` are created which are initialised with the values of `iNr2` and `iNr1 % iNr2` of the previous level of recursion. The recursion continues down to the level where `iNr2 == 0`. At that point the most deeply nested function returns the value `iNr1`. From here onwards, the recursion is unstacked, and the old function parameters are removed as each recursive call returns a value.

Jamie's recipes for disaster: **infinite recursion**

Clearly, you should never forgot to include some way of terminating a recursion. For instance, in the function `int getGCD()`, the recursion ends when `iNr == 0`. If a recursive function cannot return at some point without calling itself, its invocation will initiate an infinite recursion from which your program will not recover.

8.2 Exercises

8.1. References. Take a look at listing 8.2 and attempt to predict the output. Then, implement the program and check if your prediction was correct.

Code listing 8.2: Minimal working example of a reference

```
#include <iostream>

int main() {
   int i = 0;

   int &j = i;

   std::cout << "i = " << i << "; j = " << j << std::endl;

   ++j;

   std::cout << "i = " << i << "; j = " << j << std::endl;

   ++i;

   std::cout << "i = " << i << "; j = " << j << std::endl;

   return 0;
}

```

8.2. Overloading and calling by reference. First, write a function `find_max` that accepts two `doubles` and returns the largest value. Then, write another function `find_max` that accepts two **`int`** values and returns the largest value. Lastly, write another function `find_max` that accepts two doubles and sets the first value to the maximum value and the second value to the mininum value.

8.3. Calling by reference. Consider the following alternatives for the function `computeNextPopulationDensity()` of exercise 6.2.

```
double computeNextPopulationDensity1(double &N, const double &r)
{
   return N * exp(r * (1.0 - N));
}

double computeNextPopulationDensity2(const double &N, const double &r)
{
   return N * exp(r * (1.0 - N));
}

void updatePopulationDensity(double &N, const double &r)
{
   N *= exp(r * (1.0 - N));
}

```

Compare these with the original function, and explain which of the four comply with Nefertiti's guidelines for passing arguments to functions. Also show how `updatePopulationDensity()` can be incorporated into your previous implementation of the Ricker model.

8.4. Computing the factorial of a number. Write an iterative and a recursive version of a function that computes the factorial $n!$ of an integer number $n \geq 0$. Recall that $0! = 1$, by definition, and that $n! = n \cdot (n-1)! = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$, for all $n > 0$. When testing the functions, use only small values of n to avoid integer overflow.

Test your skill

8.5. Predator Prey dynamics ($\star$). Write a program that can simulate a Predator Prey Scenario. In this

scenario, there are two species in the ecosystem: Rabbits (Prey) and foxes (Predators). Each year, the populations of rabbits and foxes change based on the following rules:

1. Rabbits reproduce: The number of rabbits increases by 30% each year.
2. Rabbits and Foxes die: every year, 10% of foxes and rabbits dies due to winter conditions.
3. Foxes hunt rabbits: Every fox has a 5% chance of capturing a rabbit, thus the total number of rabbits caught per year is: 0.05 * foxes * rabbits
4. Foxes digest rabbits: The population of foxes increases with 10% of the total number of rabbits eaten.

Your task is to simulate the populations of rabbits and foxes over time using recursion and /or references. You need to create a function that updates both populations year by year. Write a function simulateEcosystem that takes the current year and the number of rabbits and foxes as parameters. For each year, output the number of rabbits and foxes. Start the simulation with 100 rabbits and 10 foxes and simulate the ecosystem for 100 generations. Do you notice anything striking in the population dynamics of rabbits and foxes? What happens if you change the population increase of foxes with the number of rabbits eaten from 10% to 50 %? What causes this?

Code listing 8.3: Example output of Predator prey program

```
Year 0: Rabbits = 100, Foxes = 10
Year 1: Rabbits = 65, Foxes = 14.2
Year 2: Rabbits = 28.05, Foxes = 17.58
```

8.6. Advanced version ($\star$). Can you rewrite the function simulateEcosystem, such that instead of using passing by reference, it uses recursion to iterate over the generations? Alternatively, if you previously solved using recursion, can you rewrite your solution to use references?

8.3 Solutions to the exercises

Exercise 8.1. References. The console output generated by the three output statements is:

```
i = 0; j = 0
i = 1; j = 1
i = 2; j = 2
```

Exercise 8.2. Overloading and calling by reference.

```
double find_max(double A, double B) {
   if (A > B) return A;
   return B;
}

int find_max(int A, int B) {
   if (A > B) return A;
   return B;
}

void find_max2(double& A, double& B) {
   if (B > A) {
      double temp = A;
      A = B;
      B = temp;
   }
}

```

Exercise 8.3. Calling by reference. The original `double computeNextPopulationDensity(double, const double)` and `double computeNextPopulationDensity1(double &, const double &)` do not comply with Yoda's guidelines, because these functions require only read access to their function parameters. Hence, the preferred way to pass the arguments is by `const`-reference, as in `double computeNextPopulationDensity2(const double&, const double&)`. The alternative solution of an update function is fine as well, but it requires a minor change in the initialisation and iteration loop of the Ricker model:

```
// prepare for simulation
std::vector<double> vec(tEnd + 1);
double Nt = vec[0] = N0;

// iterate Ricker model
for (int t = 0; t < tEnd; ++t) {
   updatePopulationDensity(Nt, r);
   vec[t + 1] = Nt;
}

```

Exercise 8.4. Computing the factorial of a number.

```
int computeFactorialIterative(int n)
{
   int nFactorial = 1;
   while(n) {
      nFactorial *= n;
      --n;
   }
   return nFactorial;
}

```

```
11      int computeFactorialRecursive(const int &n)
12      {
13         return n == 0 ? 1 : n * computeFactorialRecursive(n - 1);
14      }
15
```

Module 9

Working with text

Jamie explains what's on the menu

In this module you will:

- Become familiar with the data type **char**.
- Learn how pieces of text are stored and manipulated in `std::string` objects.

9.1 Character constants and variables

Characters, the elementary building blocks of a piece of text, are represented in C++ by variables of the type **char**. The type **char** behaves as an integral type that occupies one byte of memory space on most platforms. Furthermore, a **char** is **unsigned** by default, which means that a **char** variable can take the values 0 to 255. Each of these values is automatically translated to a character from the standard (extended) ASCII character set by the output operator. For example, the following code generates the output `hi!`, because character 104 in the ASCII table is 'h'; character 105 is 'i'; and character 33 is the exclamation mark.

```
1  #include <iostream>
2
3  int main()
4  {
5      const char ch1 = 104;
6      const char ch2 = ch1 + 1;
7      const char ch3 = 33;
8
9      std::cout << ch1 << ch2 << ch3;
10
11     return 0;
12 }
```

***Master Yoda* on using `char` as an integer type**

As you can see on line 6 of the simple program above, **`char`** variables can be used in calculations just as any other integral data type. Using **`char`**s in this way can be good idea if you need to worry about memory usage. However, you should be extremely careful to avoid integer overflow errors, because values larger than 255 will exceed the capacity of a **`char`**. If it is necessary to work with values between -127 and 127, you can declare variables of type **`signed char`**.

It would be a bit annoying if we would have to consult the ASCII table all the time to look up the code corresponding to a specific character. Luckily, we can also assign values to characters by using character literals. These are easily recognised by their enclosing single quotes `' '`. For instance, the declaration statement **`const char`** `ch1 = 'h';` is equivalent to the declaration on line 5 of the previous code example.

In addition to the alphanumeric characters a...z, A...Z and 0...9, the punctuation marks and special keyboard characters (@,#,$) and so on, a few special characters are represented by character escape sequences that all start with backslash. They are:

Table 9.1: Character escape sequences

`'\n'`	newline (moves cursor to first position of next line)
`'\t'`	horizontal tab
`'\r'`	carriage return (returns cursor to first position of current line)
`'\b'`	back space
`'\\'`	backslash
`'\''`	single quote
`'\"'`	double quote

***Jamie's recipes for disaster:* misinterpreting `char` literals**

It is easy to confuse a character with the value used for representing it internally. For example, the expression **`char`** `ch = '0'` is not the same as **`char`** `ch = 0`. Instead, the first expression is equivalent to **`char`** `ch = 48`, because 48 is the index of the character 0 in the ASCII table.

9.2 Strings

You may recognise why the list of special characters also contains an escape sequence for the double quote character: this character has a special meaning in C++, since it is used to delimit *string literals*, such as `"This is a string literal.\n"`. String literals are internally stored as an array of characters, which can be passed to an output operator to print all the characters in the string (as you have already done many times). Working with these 'C-style' character arrays in other ways, however, is a bit cumbersome. Fortunately, the C++ library header `string` defines a string class object that implements most of the work for us.

To gain access to the `string` class, you must include the statement **`#include`** `<string>` at the start of your program. Now you can define strings in three different ways, as illustrated in the following example program:

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
```

```
6     std::string strUserName;
7     const std::string strAskInput("Please enter your name: ");
8
9     std::cout << strAskInput;
10    std::cin >> strUserName;
11
12    const std::string strChitchat("Hi, " + strUserName + ". How are you today?\n");
13
14    std::cout << strChitchat;
15
16    return 0;
17 }
```

The output of the program (including user input) is given by:

```
Please enter your name: Yoda
Hi, Yoda. How are you today?
```

The first way of defining a `std::string` is illustrated by the statement on line 6, which declares a string `strUserName` that is empty. Alternatively, a non-empty `std::string` can be created with a C-string as initialiser, as shown on line 7. A third way to create a `std::string` is to use another `std::string` object as initialiser, as shown on line 12; here the string object used for the initialisation of `strChitchat` is made by prepending `"Hi, "` at the begin and appending `". How are you today?\n"` at the end of `strUserName` by means of the + operator. (This operator can be used to concatenate two `std::string` objects, or a `std::string` object and a C-style string, but not two C-style strings.)

Besides the + operator, you can use several other operators to manipulate `std::string` objects:

Table 9.2: Available operators for `std::string` objects

`=`	assignment (copies one string into another)
`+`	concatenates two strings
`+=`	compound concatenation-assignment
`==`	checks if two strings are identical
`!=`	checks if two strings are not identical
`<` (or `<=`)	checks if one string comes before (or not later than) another in alphabetical ordering
`>` (or `>=`)	checks if one string comes after (or not before) another in alphabetical ordering
`[]`	access to character at specific position in string
`<<`	output
`>>`	input

String objects behave in many ways similar as a `std::vector<char>`: they have an operator `[]` that gives access to individual characters; and they have several member functions in common with `std::vector` that you can access via the member selection operator . ('dot'). These include, for example, `size()`, which returns the length of the string, `push_back()`, which adds a character at the end, `clear()`, which clears the contents of the string, `empty()`, which tells you whether the string is empty or not, and `front()` and `back()`, which provide access to the first or last element, respectively. In addition, string objects have specific member functions such as `substr()`, `insert()`, `replace()` and `erase()` that allow for different kinds of string manipulations. In particular, the statement

```
std::string substring = strSource.substr(pos, num);
```

copies `num` characters from `std::string strSource` (or as many as possible until the end of the string is reached), starting from position `pos`, and puts them into a new `std::string` object. The complementary operation is accomplished by the member function `insert()`. For example,

```
strTarget.insert(pos, strOther)
```

pastes the `std::string strOther` into `std::string strTarget` at position `pos`. In the same way,

```
strTarget.replace(pos, num, strOther)
```

writes `std::string strOther` into `std::string strTarget`, but it replaces `num` characters in the original string starting from position `pos`. To simply erase the same characters without replacing them by some other `std::string`, use the function `erase`, as in

```
strTarget.erase(pos, num)
```

To search a `std::string`, call the member function `find()`. For instance,

```
strTarget.find(strOther, pos)
```

searches `std::string strTarget` for the occurrence of `std::string strOther`. Starting the search from position `pos`, it returns the first position at which `std::string strOther` appears, or the value `std::string::npos` if no match is found. This constant, defined in the class `std::string`, corresponds to the length of the longest possible string (which evaluates to the signed integer value -1). The second argument of the function `find()` is optional; if no value is provided, the search will begin at the start of the string.

Listing 9.1 illustrates `erase()`, `insert()`, `find()`, `replace()` and `size()` in an otherwise utterly useless Yoda-quote-generator that produces the output:

```
A very brilliant programmer ET is!
A not so brilliant programmer Arnold is!
```

Code listing 9.1: String manipulation functions

```
#include <iostream>
#include <string>

int main()
{
    const std::string strET = "ET";
    std::string strYodaSays = "A very brilliant programmer " + strET + " is!\n";

    std::cout << strYodaSays;

    // erase substring "very " (5 characters starting from position 2)
    strYodaSays.erase(2, 5);

    // insert "not so " at position 2
    strYodaSays.insert(2, "not so ");

    // find "ET" in strYodaSays
    const int pos = strYodaSays.find(strET);

    if(pos != std::string::npos) {
        // replace "ET" by "Arnold" & illustration of size()
        strYodaSays.replace(pos, strET.size(), "Arnold");
    }

    std::cout << strYodaSays;

    return 0;
}
```

Last, but not least, it is sometimes helpful to be able to convert between a `std::string` and other data types. Table 9.3 lists a selection of the available conversion functions.

Table 9.3: Functions for converting between `std::string` objects and other data types

`str.c_str()`	Returns a C-style string version of the `std::string` object `str`.
`std::stoi(str)`	Interprets the `std::string` object `str` as a number and returns the corresponding integer value.
`std::stod(str)`	Interprets the `std::string` object `str` as a number and returns the corresponding double-precision floating-point value.
`std::to_string()`	Converts an integral or floating point value to a `std::string` object.

9.3 Exercises

9.1. Looping through the alphabet. Look up an ASCII table on the internet and verify that the letters of the alphabet are listed on consecutive positions in the table. Exploit this fact to write a loop that prints all letters of the alphabet.

9.2. Converting between upper- and lowercase letters. The functions `tolower()` and `toupper()`, respectively, which are defined in the C-library header `cctype`, convert uppercase letters to lowercase and vice-versa. Code your own version of these functions.

9.3. DNA transcription. Write a function that transcribes DNA into RNA. It takes as argument a DNA ense sequence and stored in a `std::string` object; and it returns the RNA antisense sequence transcribed from it and stored in a `std::string` object. [1] To test your program, apply the function to a short sequence, such as CGTCACAGATTAAGGTATACCATT and print the result to the screen.

Test your skill

9.4. RNA translation. Extend your solution to the previous exercise with a function that can read an RNA sequence, and that lists the separate codons [2] that it contains, starting from the start codon (AUG), until a stop codon (UAG, UGA or UAA) is encountered.

[1] Double-stranded DNA contains two strands complementary to each other. The strand that reads from 5' to 3', called the sense strand, is the coding sequence, but is not the sequence used for transcription. Its complementary sequence, called the antisense strand, reads from 3' to 5', and is used as the template for transcription. Transcription results in a complementary messenger RNA (mRNA) sense transcript, where T's become U's. Save from U's, this transcript is identical to the sense DNA strand. Similarly, the complement antisense RNA transcript is identical to the template DNA antisense strand. Below is an example DNA sequence, its complement and the resulting RNA.

DNA antisense strand	3'	CGCTATAGCGTTT	5'
DNA sense strand	5'	GCGATATCGCAAA	3'
mRNA sense transcript	5'	GCGAUAUCGCAAA	3'
mRNA antisense transcript	3'	CGCUAUAGCGUUU	5'

[2] A codon is a triplet of RNA nucleobases (A, U, C, G) that will be translated into an amino-acid. For example, the codon CGA codes for Alanine. A coding RNA sequence is a series of codons starting with the start codon AUG and ending with a stop codon (UAG, UGA or UAA), which during translation will result in a series of corresponding amino-acids, forming a protein.

9.4 Solutions to the exercises

Exercise 9.1. Looping through the alphabet.

```
#include <iostream>

int main()
{
    for(char ch = 'a'; ch <= 'z'; ++ch)
        std::cout << ch;

    return 0;
}
```

Exercise 9.2. Converting between upper- and lowercase letters.

```
#include <iostream>

char my_tolower(char chUppercase)
{
    if(chUppercase < 'A' || chUppercase > 'Z') {
        std::cout << "warning: " << chUppercase << "is not an uppercase letter\n";
        return chUppercase;
    }
    else
        return chUppercase + 'a' - 'A';
}

char my_toupper(char chLowercase)
{
    if(chLowercase < 'a' || chLowercase > 'z') {
        std::cout << "warning: " << chLowercase << "is not an lowercase letter\n";
        return chLowercase;
    }
    else
        return chLowercase + 'A' - 'a';
}

int main()
{
    std::cout << "testing my_tolower():\n";
    for(char ch = 'A'; ch <= 'Z'; ++ch)
        std::cout << ch << " -> " << my_tolower(ch) << std::endl;

    std::cout << "testing my_toupper():\n";
    for(char ch = 'a'; ch <= 'z'; ++ch)
        std::cout << ch << " -> " << my_toupper(ch) << std::endl;

    return 0;
}
```

Exercise 9.3. DNA transcription.

```
std::string transcribeDNA(const std::string &antisenseDna)
{
    std::string rna;
    // read bases from antisense DNA in the 3' -> 5' direction
    for(int i = antisenseDna.size(); i > 0; ) {
        --i;
```

```
        if(antisenseDna[i] == 'A')
            rna.push_back('U');
        else if(antisenseDna[i] == 'C')
            rna.push_back('G');
        else if(antisenseDna[i] == 'G')
            rna.push_back('C');
        else if(antisenseDna[i] == 'T')
            rna.push_back('A');
        else std::cout << "warning: skipping invalid character in DNA sequence\n";
    }
    return rna;
}
```

An elegant alternative to the **if**...**else** ladder is to use a **switch** statement (see p. 43):

```
std::string transcribeDNA(const std::string &antisenseDna)
{
    std::string rna;
    // read bases from antisense DNA in the 3' -> 5' direction
    for(int i = antisenseDna.size(); i > 0; ) {
        --i;
        switch(antisenseDna[i]) {
            case 'A' : rna.push_back('U'); break;
            case 'C' : rna.push_back('G'); break;
            case 'G' : rna.push_back('C'); break;
            case 'T' : rna.push_back('A'); break;
            default :
                std::cout << "warning: skipping invalid character in DNA sequence\n";
        }
    }
    return rna;
}
```

Module 10

"Hello disk drive!"

In this module you will:

Jamie explains what's on the menu

- Learn how your program can gain access to files on the disk drive.
- Discover stream objects other than `std::cout` and `std::cin`, and see how you can work with them
- Become familiar with input and output file stream objects from the library `fstream`
- Be introduced to I/O manipulators that allow you to customise input and output operations

10.1 Reading from and writing to a file

The library `iostream`, which we have used from the start to write output to the screen and read input from the keyboard, defines a set of functions and operators that work on so-called stream objects. In fact, `std::cin` and `std::cout` are predefined streams that are opened automatically when your program starts. Two additional predefined streams that you can write output to are `std::clog` (for writing messages to the program log) and `std:cerr` (for error messages). Normally, output written to these streams appears on the screen, just as output to `std::cout`, but if you run your program remotely (e.g., on a computer cluster), `std::clog` and `std::cerr` may be redirected by the operating system to separate files that you can download for inspection after your simulation terminated. You can also redirect the predefined streams yourself, by linking together the internal read/write buffers associated with different stream objects. For example, the following code fragment redirects output to `std::cerr` to a `.txt` file `errors.txt`:

```
1 std::cerr << "This error message will appear on the screen" << std::endl;
2
3 std::ofstream errorFileStream("errors.txt");
4 std::cerr.rdbuf(errorFileStream.rdbuf());
5
6 std::cerr << "This error message will appear in the file errors.txt" << std::endl;
```

The statement on line 3 of the code fragment introduces a new kind of stream object, `std::ofstream` (for output file stream), that is defined in the library `fstream`. Objects of type `std::ofstream` can be associated with a file on the hard disk, and used as an interface to write data into that file. The actual redirection of `std:cerr` takes place on the following line where the stream buffer of error stream `cerr.rdbuf()` is connected to the stream buffer of `errorFileStream`.

Code listing 10.1: File I/O

```
1  #include <iostream>
2  #include <fstream>
3  #include <cmath>
4  #include <cstdlib>
5
6  int main() {
7      // open file with measurement data
8      std::ifstream ifs("measurements.txt");
9      if(!ifs.is_open()) {
10         std::cerr << "error: unable to open data file\n";
11         exit(EXIT_FAILURE);
12     }
13
14     // read input data
15     double dAvg = 0.0, dStdDev = 0.0;
16     int iN;
17     for(iN = 0;;) {
18
19         // read from file
20         double x;
21         ifs >> x;
22
23         // add data point
24         ++iN;
25         dAvg += x;              // dAvg temporarily stores sum of data values
26         dStdDev += x * x;       // dStdDev temporarily stores sum of squared data values
27
28         if(ifs.eof()) {         // calculate summary statistics and exit when end of file has been reached
29             if(iN < 2) exit(EXIT_SUCCESS); // exit if unable to calculate mean and/or standard deviation
30             dAvg /= iN;
31             dStdDev = sqrt((dStdDev - iN * dAvg * dAvg) / (iN - 1));
32             break;
33         }
34     }
35     ifs.close();    // here you can see how to manually close a file
36
37     // open output file
38     std::ofstream ofs;              // no real reason for using open() here other than for illustration; can be
39     ofs.open("summary_stats.txt"); // replaced by a single statement std::ifstream ofs("summary_stats.txt");
40     if(!ofs.is_open()) {
41         std::cerr << "error: unable to open output file\n";
42         exit(EXIT_FAILURE);
43     }
44
45     // write summary statistics to file
46     ofs << "N     = " << iN      << '\n'
47         << "mean  = " << dAvg    << '\n'
48         << "stdev = " << dStdDev << '\n';
49
50     return 0;
51 }
```

Apart from gaining *write* access to a file via a `std::ofstream` object, we can also open files to read information from them. The interface for *read* access to files is provided by the stream type `std::ifstream`

(input file stream), which is also defined in the `fstream` library. The easiest way of coupling a file to a `std::ofstream` or `std::ifstream` object is to provide the name of the file at the moment of initialisation (as on line 3 of the code fragment above). Alternatively, you can create the file stream object first and couple it to a file at a later stage by calling the member function `open()` and passing it the name of the file. The member function `is_open()` allows you to verify that a file was opened successfully. It is advisable to always check this before attempting to access the file for reading or writing. All open files are closed automatically when the file stream object is destroyed, but you can also call the function `close()` to dissociate a file from a stream manually. This can be helpful, for example, if you want to re-use the stream object to access another file.

Once you have opened a file via a `std::ofstream` object, you can write information to the file stream in the same way as you would write to `std::cout`. The text will appear in the file exactly as it would otherwise appear on the screen. Likewise, reading from a file via a `std::ifstream` object occurs in the same way as reading from `std::cin`; i.e., the contents of the file will be processed in the same way as user input entered via the keyboard. As an example, listing 10.1 shows how to read in a list of measurements from a data file `measurements.txt`. After calculating the mean and standard deviation of the measurements, the summary statistics are written to an output file `summary_stats.txt`.

***Master Yoda* explains how to append data to a file**

When you open an existing file via a `std::ofstream` object, the previous contents of the file are lost. If this default behaviour is unintended, you can instead open the file in append mode. This prevents the loss of data by causing the new output to be appended to the end of the existing file. Opening a file in append mode is accomplished by providing an additional argument `ios::app` to the `std::ofstream` constructor (or to `open()` if this function is used to couple the stream to the file). For example, after declaring `std::ofstream ofs("datafile.txt", ios::app)`, output sent to stream `ofs` will be added at the end of `datafile.txt` (if the file already exists); if there is no previous copy of the file, `ios::app` has no effect and a new file will be created.

As you can see, the code of the example program consists of four sections. First, one line 9-13, the file `measurements.txt` is opened for reading via the input file stream `ifs`. The **`if`** statement on line 10 checks if any problems were encountered while attempting to open the file (such as that the file was not found or that access to it was blocked by another program). If the file was not opened successfully, the program generates an error message, (written to `std::err`) which is followed by a call to the function `exit()`. This function, defined in the C-header `cstdlib` (included on line 4), terminates the program, performing the usual cleanup tasks and passing an exit code to the operating system (`EXIT_SUCCESS` or 0 signals normal program termination; `EXIT_FAILURE` or 1 indicates that a fatal error occurred).

The second section of the code (lines 16-36) is where the data are actually read from the file. As you can see, the relevant statement on line 22 is written in the same way as an input operation from `std::cin`. Values are read from the file one after the other until the end of the file has been reached. This is indicated by the member function `eof()`, which evaluates to **`true`** as soon as an input operation encounters a special character `EOF` that signals the end of the file. The input file can be closed once all data have been processed. It is not strictly necessary to do this explicitly, as occurs on line 36. If a call to `close()` is missing, the file will be closed automatically when the file stream `ifs` is destroyed (this occurs here when control reaches the end of the function `main()`).

The third section of the code, lines 39-44, is for opening the output file `summary_stats.txt`, and checking whether this operation was successful. The final section (line 47-49) contains the actual output statement, where the summary statistics are written into the file via the output file stream `ofs`. Note that the syntax of this statement is identical to how you would write information to `std::cout`.

The program in listing 10.1 works as intended when the input file contains only values that can be interpreted as floating point numbers, separated by whitespace characters (spaces, tabs or newline characters). For example, a `measurements.txt` file with contents

```
0.852831591 0.884181555 0.507363615
0.302636383 0.286680422 0.876757749
0.078237856 0.499252192 0.319069937
0.292467082 0.093734513 0.770918102
```

is processed correctly. The resulting output file contains the following information:

```
N     = 12
mean  = 0.480344
stdev = 0.2994
```

Dealing with less neatly formatted input files can be a bit of a hassle, because attempted input operations that somehow go wrong will block the input stream. When this happens, no further data can be read. To check for these problems, input and output streams have several functions besides `eof()` that provide information about the status of the stream. These include `good()`, which returns **`true`** if no error flags have been set, and `fail()`, which returns a **`bool`** to indicate whether an error has occurred during an input operation. If an error has occurred, `clear()` must be called to reset the error flags, after which you can skip problematic characters in the input stream by calling the function `ignore()`). Except for fatal errors that caused a loss of integrity of the stream, you can then attempt to resume input operations. The following code fragment, which replaces line 15-36 in listing 10.1, illustrates how checking the status of a stream and correcting input errors can make input operations more robust. In particular, the program was tested with a `measurements.txt` file that contained text fields pasted in between the data values:

```
0.852831591 0.884181555 missing value
0.507363615 0.302636383 0.286680422
another failed measurement 0.876757749
0.078237856 0.499252192 0.319069937
0.292467082 0.093734513 0.770918102
```

```
1   // read input data
2   double dAvg = 0.0, dStdDev = 0.0;
3   int iN;
4   for(iN = 0;;) {
5
6       // attempt to read from file
7       double x;
8       ifs >> x;
9
10      if(!ifs.fail()) {        // add data point if read operation was successful...
11          ++iN;
12          dAvg += x;               // dAvg temporarily stores sum of data values
13          dStdDev += x * x;        // dStdDev temporarily stores sum of squared data values
14      }
15      else {                   // otherwise:
16          ifs.clear();             // clear error flags
17          ifs.ignore();            // ignore next character
18      }
19
20      if(ifs.eof()) {          // calculate summary statistics and exit when end of file has been reached
21          if(iN < 2) exit(0);      // exit if unable to calculate mean and/or standard deviation
22          dAvg /= iN;
23          dStdDev = sqrt((dStdDev - iN * dAvg * dAvg) / (iN - 1));
24          break;
25      }
26  }
```

Stringstream objects

Beyond filestreams, the standard C^{++} library offers several other stream objects for I/O operations. One that is useful to know is the `std::ostringstream` (or, *o*utput *string stream*), an object that processes output operations just like any other stream object but that writes the contents of its stream buffer into a `std::string` (rather than to the screen or to a file). A common application of these stringstream objects is to write the value of a variable into a `std::string` which then subsequently specifies the name of a file. For example, code listing 10.2 shows how to create a series of files `data_0.txt`, `data_100.txt`, ..., and `data_1000.txt`, to store the current state of a simulation at time point $t = 0$, $t = 100$, ..., and $t = 1000$, respectively. In order to use output string stream objects in your code, it is necessary to include the header `sstream`, as shown on line 4 of listing 10.2. Access to the `std::string` object associated with the string stream is provided by the member function `str()` (called on line 23). On line 26, you can see how the `std::string filename` is converted to a C-style string, which is then passed as an argument to the constructor of an output file stream. If you like, the statements on line 23 and 26 can be combined into a single statement to open the file:

```
std::ofstream ofs(oss.str().c_str());
```

Code listing 10.2: Applying stringstreams to generating filenames during a simulation

```
#include <string>
#include <iostream>
#include <fstream>
#include <sstream>

int main() {
   const int tFinal = 1000;       // simulate for tFinal steps
   const int tSaveSimState = 100;    // store data every tSaveSimState steps

   for (int t = 0; t <= tFinal; ++t) {

      // store simulation state in file
      if (t % tSaveSimState == 0) {

         // create output stringstream
         std::ostringstream oss;

         // write to stringstream
         oss << "data_" << t << ".txt";

         // access std::string by calling member function str()
         std::string filename = oss.str();

         // open file and store information
         std::ofstream ofs(filename.c_str());
         if (ofs.is_open()) {
            // insert output statements here to write data to file...
         }
         ofs.close();
      }

      // enter code here to update simulation state variables...
   }
   return 0;
}
```

10.2 Exercises

10.1. Practising file I/O - part 1. Prepare a file `measurements.txt` containing a list of 20 arbitrary floating point values (e.g. paste some values from a spreadsheet into a simple text editor and save the result as a `.txt` file). Next, write a C++ program that opens the file, reads in the values and displays them on the screen.

10.2. Practising file I/O - part 2. Extend your solution of the previous exercise in two steps. First replace a few numbers in your `.txt` input file by the character code `NA` to represent missing values, and adjust the `.cpp` code to correctly deal with these elements in the input file. Next, adjust the program to allow the user to input the name of the input file. Make sure the program is terminated correctly if no file name is provided.

Test your skill

10.3. Practising file I/O - part 3 (⋆). Building on your solution to the previous exercises, write a program that reads in a list of floating point values x_i from a file and that replaces them by values y_i that are scaled between 0 and 1 as follows:

$$y_i = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$

Here, $\min(x)$ is equal to the smallest of the x_i and $\max(x)$ is equal to the largest.

Customising and formatting input and output

The default rules for formatting output and interpreting input work well in many cases, but if needed, there are several ways to precisely control the format of data processed via streams. We here introduce one method, which relies on so-called I/O manipulators. These are special functions that can be directly included in input and output expressions. In fact, you already know how to use one of these manipulators: `std::endl`, which outputs a newline character and then flushes the stream.

Most I/O manipulators take no arguments and they are included in the chain of I/O operations without parentheses (obscuring the fact that they involve a call to a manipulator function). There are also a few manipulators that take an argument. To use these, you need to include the library `iomanip`. Table 10.1 list a selection of the available I/O manipulators; more can be found in the documentation of the `iostream` library. A great online resource for finding information on the C^{++} libraries and example code is available at **cppreference.com**.

To see an application of I/O manipulators in the context of a simple simulation program, consider the code in listing 10.3 that tracks the densities of two populations that grow according to the Ricker model

$$N_{t+1} = R\, N_t\, \exp(-\alpha\, N_t)$$

The growth parameters $R = 20.0$ and $\alpha = 0.01$ are identical for both populations, but they start at slightly different initial densities. This is done to show that there is a sensitive dependence on initial conditions in the Ricker model for this parameter set.

Code listing 10.3: I/O manipulators

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cmath>
#include <cstdlib>

const double    dX0             = 1.0;      // initial density population 1
const double    dY0             = 1.000001; // initial density population 2
const double    dR              = 20.0;     // maximal per-capita reproduction factor
const double    dAlpha          = 0.01;     // strength of density dependence
const int       iGenerationEnd  = 60;       // simulation length
const char      chFieldSeparator= ',';      // to separate between number fields in .csv file

int main()
{
    // open data file
    std::ofstream ofs("data.csv");
    if(!ofs.is_open()) {
        std::cerr << "error: unable to open data file\n";
        exit(1);
    }

    // set permanent output format flags
    std::cout << std::fixed << std::setprecision(1);
    ofs << std::scientific << std::setprecision(8);


    // write column headers
```

```
29      std::cout << "time   X_t   Y_t\n";
30      ofs << "time" << chFieldSeparator
31          << "            X_t" << chFieldSeparator
32          << "            Y_t" << chFieldSeparator
33          << "       X_t - Y_t\n";
34
35      // set initial conditions and start iteration
36      double dX = dX0, dY = dY0;
37      for(int iGeneration = 0; iGeneration < iGenerationEnd; ++iGeneration) {
38
39          // update population states
40          dX = dR * dX * exp(-dAlpha * dX);
41          dY = dR * dY * exp(-dAlpha * dY);
42
43          // write formatted output to screen
44          std::cout   << std::left << std::setw(4) << iGeneration << ' '
45                      << std::right << std::setw(5) << dX << ' '
46                      << std::setw(5) << dY << '\n';
47
48          // write formatted output to file
49          ofs << std::setw(4) << std::noshowpos
50              << iGeneration << chFieldSeparator
51              << std::setw(15) << dX << chFieldSeparator
52              << std::setw(15) << dY << chFieldSeparator
53              << std::setw(16) << std::showpos << dX - dY << std::endl;
54      }
55      return 0;
56  }
```

Indeed, the output data generated by the simulation program shows that the population densities gradually diverge over time up to the point that the initial correlation between the time series is lost (Figure 10.1). The program produces neatly formatted output to the screen, which is shown here for the first 5 time steps:

```
time   X_t   Y_t
0     19.8  19.8
1    324.9 324.9
2    252.2 252.2
3    404.9 404.9
4    141.2 141.2
...
```

It also generates a neatly formatted `.csv` file that can be inspected in a text editor or loaded directly into a spreadsheet program like MS-Excel for plotting the results (Figure 10.1). The first lines of the file look like this:

```
time,            X_t,            Y_t,       X_t - Y_t
   0, 1.98009967e+01, 1.98010163e+01, -1.96029865e-05
   1, 3.24879577e+02, 3.24879835e+02, -2.57944621e-04
   2, 2.52242541e+02, 2.52242090e+02, +4.50372746e-04
   3, 4.04923417e+02, 4.04924518e+02, -1.10068493e-03
   4, 1.41202645e+02, 1.41201475e+02, +1.17036717e-03
...
```

Jamie Oliver
on `.csv` files

When MS-Excel does not load `.csv` files correctly, this is likely because MS-Windows regional settings instruct it to expect a semicolon as a separator between number fields rather than a comma. To deal with this weird way of interpreting 'comma-separated' files, it is helpful to declare a (global) field separator character that you can change with minimal editing to your code (as on line 12 of listing 10.3), or you can adjust the way Windows deals with `.csv` files by resetting the number format under *Regional settings* in the *Control panel*.

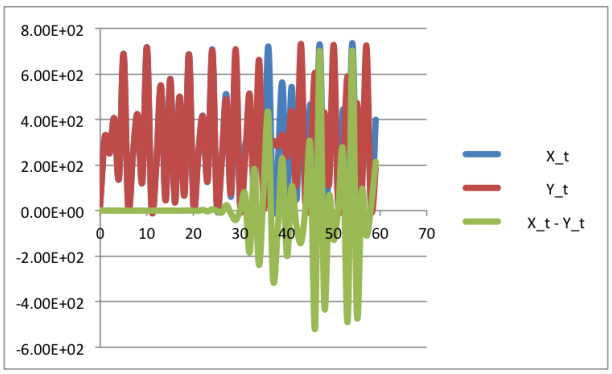


Fig. 10.1: Quick and dirty MS-Excel plot of the data in the file `data.csv` produced by the program in listing 10.3.

10.3 Solutions to the exercises

Exercise 10.1. Practising file I/O - part 1.

```
#include <iostream>
#include <fstream>
#include <cstdlib>

int main()
{
    const std::string filename("measurements.txt");

    // open file
    std::ifstream ifs(filename.c_str());
    if(!ifs.is_open()) {
        std::cerr << "error: unable to open datafile " << filename << '\n';
        exit(EXIT_FAILURE);
    }

    // read data from file and display on screen
    for(int n = 0;; ++n) {
        double dMeasurement;
        ifs >> dMeasurement;
        std::cout << "measurement " << n << " : " << dMeasurement << '\n';
        if(ifs.eof()) break;
    }
    return 0;
}
```

Exercise 10.2. Practising file I/O - part 2.

```
#include <iostream>
#include <fstream>
#include <cstdlib>

int main() {

    std::string filename;
    std::cin >> filename;

    // open file
    std::ifstream ifs(filename.c_str());
    if(!ifs.is_open()) {
        std::cerr << "error: unable to open datafile " << filename << '\n';
        exit(EXIT_FAILURE);
```

```
15      }
16
17      // read data from file and display on screen
18      for(int n = 0;;) {
19          double dMeasurement;
20          ifs >> dMeasurement;
21
22          if(!ifs.fail()) {
23              std::cout << "measurement " << n << " : " << dMeasurement << '\n';
24              ++n;
25          }
26          else {
27              ifs.clear();
28              ifs.ignore();
29          }
30
31          if(ifs.eof()) break;
32      }
33      return 0;
34  }
```

Table 10.1: A selection of I/O manipulators.

Manipulator	Context[1]	Example	Effect
Padding of numeric values			
`left`	O	`std::cout << std::left << i;`	Pads fill characters to the right so that output is left justified.
`right` (default)	O	`std::cout << std::right << i;`	Pads fill characters to the left so that output is right justified.
`internal`	O	`std::cout << std::internal << i;`	Inserts fill characters behind sign (or base) character.
Formatting of numeric fields			
`setw()`	O	`std::cout << std::setw(5) << x;`	Sets numeric field width.
`showpos`	O	`std::cout << std::showpos << x;`	Shows the sign of a number also for positive values.
Notation of floating point numbers			
`fixed`	O	`std::cout << std::fixed << x;`	Forces floating point numbers to be displayed in fixed-point notation.
`scientific`	O	`std::cout << std::scientific << x;`	Forces floating point numbers to be displayed in scientific notation.
`setprecision()`	O	`std::cout << std::setprecision(3) << x;`	Sets the number of digits displayed after a decimal point.
`showpoint`	O	`std::cout << std::showpoint << x;`	Shows floating point and trailing zeros for all floating point output.
`noshowpoint`	O	`std::cout << std::noshowpoint << x;`	Cancels `showpoint`.
Other manipulators			
`boolalpha`	I/O	`std::cin >> std::boolalpha >> isTrue;`	Uses `true` and `false` instead of `0` and `1` to represent boolean values.
`noboolalpha`	I/O	`std::cin >> std::noboolalpha >> isTrue;`	Cancels `boolalpha`.
`setfill()`	O	`std::cout >> std::setfill(',') >> x;`	Set fill character.
`flush`	O	`std::cout << std::flush;`	Flushes the output stream.
`endl`	O	`std::cout << std::endl;`	Inserts a newline character and flushes the stream.
`skipws`	I	`std::cin >> std::skipws >> str;`	Skips leading whitespace characters.
`noskipws`	I	`std::cin >> std::noskipws >> str;`	Cancels `skipws`.

1 Indicates whether a manipulator is used only in input operations (I), only in output operations (O), or in both (I/O)

Module 11

Home-made objects

Jamie explains what's on the menu

In this module you will:

- Learn how to define compound data types and create instances of them that contain variables of different kinds.
- Become familiar with `std::pair`, **`struct`** and **`class`** objects.
- Apply the principle of data encapsulation and learn how to access **`private`** data members of objects via **`public`** member functions.

11.1 Pair objects

In module 5, *Doing stuff many times*, you learned that a `std::vector` is great for creating a collection of variables that can be referred to by a common name. It is important to realise, however, that all of the values stored in a `std::vector` are of the same type. There are situations where this restriction is problematic. For example, suppose we need to process a set of characteristics for a group of patients participating in an experimental study. For each patient, we have an anonymous patient ID (encoded by a `std::string`) and a blood pressure measurement (encoded by a **`double`**). How can we now best store the two pieces of data that belong to a single patient? They cannot be put together in a `std::vector`, because the patient ID and the blood pressure measurement are of different type. The solution to the problem is to store them inside a `std::pair`, a ready-made data type that is defined in the Standard Template Library header `utility`.

To define a `std::pair` object, first specify the types of the first and second data element that will be stored in the pair object. For a patient record, this would be `std::pair <std::string, double>`. This first part of the definition functions as a type name, and it has to be followed by the name of a variable to create an actual object. For example: `std::pair <std::string, double> patientRecord;`, declares a variable `patientRecord` of the type `std::pair <std::string, double>`. To initialise the object at the moment of declaration, simply pass the initial values of the data members to the constructor of the `std::pair` object, as in `std::pair <std::string, double> patientRecord("pm00032", 124.0);`.

Once the `std::pair` object `patientRecord` has been created, you can access the data stored inside it by referring to its first member (the patient ID) as `patientRecord.first` and to its second member (the blood pressure) as `patientRecord.second` (here you can see another example of the use of the member access operator 'dot' (`.`)). The following code fragment shows a slightly more complicated example, where a list

of patient IDs and corresponding blood pressure measurements is read from a data file `data.txt` with contents

```
pf00001        118.4
pm00002        129.2
pm00003        132.8
pm00032        124.0
pf00033        115.6
pf00034        141.4
pm00126        123.1
pf00127        127.2
```

For each patient, the ID and the blood pressure measurement are put in a `std::pair` object called `record`, which is then added to the end of a `std::vector`, thus building up an array of patient records:

Code listing 11.1: Processing patient data using a `std::pair`

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <utility>
#include <vector>
#include <string>

int main()
{
    // open file
    std::ifstream ifs("patientData.txt");
    if(!ifs.is_open()) {
        std::cerr << "error: unable to open datafile patientData.txt\n";
        exit(EXIT_FAILURE);
    }

    // read data from file
    std::vector<std::pair<std::string, double> > patientRecords;
    for(;;) {
        std::string patientID;
        ifs >> patientID;

        if(ifs.fail()) break;

        double dBloodPressure;
        ifs >> dBloodPressure;

        std::pair<std::string, double> record(patientID, dBloodPressure);
        patientRecords.push_back(record);
    }

    // show data on screen
    for(int i = 0; i < patientRecords.size(); ++i)
        std::cout << "patient " << patientRecords[i].first
                  << " has blood pressure " << patientRecords[i].second << '\n';

    return 0;
}
```

The output generated by the program is:

```
patient pf00001 has blood pressure 118.4
patient pm00002 has blood pressure 129.2
patient pm00003 has blood pressure 132.8
patient pm00032 has blood pressure 124
patient pf00033 has blood pressure 115.6
patient pf00034 has blood pressure 141.4
patient pm00126 has blood pressure 123.1
patient pf00127 has blood pressure 127.2
```

Take particular notice of the declaration statement on line 18 of the code, where you can see that `std::pair<std::string, double>` is used as a type specification for the type of elements that are put in a `std::vector`. Also carefully examine the output statement on line 34-35. Here, `patientRecords[i]` accesses the i-th element in the array of patient records via the subscript operator `[]`. This element is an object of type `std::pair<std::string, double>`. The members `first` and `second` of the object are subsequently accessed via the 'dot' member access operator.

11.2 Structures

The type `std::pair` is a predefined datatype, but you can easily build your own version of it from scratch yourself. The following program illustrates how this can be accomplished by defining a so-called *structure*. C++ structures provide a means of storing related pieces of information as a collection of variables referenced under one name. As such, they form the basis for developing *compound*, or *composite* data types, as do their close relatives, C++ class objects (see section 11.3).

Code listing 11.2: Processing patient data using a `struct`

```
1  #include <iostream>
2  #include <fstream>
3  #include <cstdlib>
4  #include <vector>
5  #include <string>
6
7  int main()
8  {
9      // open file
10     std::ifstream ifs("patientData.txt");
11     if(!ifs.is_open()) {
12         std::cerr << "error: unable to open datafile patientData.txt\n";
13         exit(EXIT_FAILURE);
14     }
15
16     struct PatientRecord {
17         std::string first;
18         double second;
19     };
20
21     // read data from file
22     std::vector<PatientRecord> patientRecords;
23     for(;;) {
24         std::string strID;
25         ifs >> strID;
26
27         if(ifs.fail()) break;
28
29         double dPressure;
30         ifs >> dPressure;
31
32         PatientRecord record = {strID, dPressure};
33         patientRecords.push_back(record);
34     }
35
36     // show data on screen
37     for(int i = 0; i < patientRecords.size(); ++i)
38         std::cout << "patient " << patientRecords[i].first
39                   << " has blood pressure " << patientRecords[i].second << '\n';
40
41     return 0;
42 }
```

The key difference between this implementation and listing 11.1 occurs on line 16-19 of listing 11.2 where the structure `PatientRecord` is defined. A structure is declared by typing the keyword **`struct`**, followed by the name of the new structure.

Code listing 11.3: Defining a structure

```
1 struct PatientRecord {
2    std::string first;
3    double second;
4 };
```

From that point onwards, the name of the structure, `PatientRecord`, will be recognised as a user-defined type name, that can be used to declare variables, just like any other type name. On line 22, for instance, `PatientRecord` appears as a template argument for a `std::vector`, and in the declaration statement on line 32, you can see how a variable of type `PatienRecord` is defined. Notice also on line 32 that the object `record` is initialised at declaration. For a structure, this is done by listing the initial values of the member variables enclosed by a pair of curly braces.

The part between the curly braces on line 16-19, which follows the **`struct`** declaration, is the definition of the structure. Here you specify the exact contents of the structure. In this case, it contains two data members, or *member variables* a `std::string` and a **`double`**. It is important to realise that the definition of a structure is nothing more than a logical outline for building a particular kind of object. No physical representation of the structure will be present in the memory until the point that an actual object, i.e., an *instance* of the structure, is created.

***Nefertiti* on type names**

It is helpful to use a consistent notation for user-defined type names, such as the names of `structs`, that will allow you to distinguish them from variable names. The convention I use is to write user-defined type names in mixed case, starting with capital letter. Some examples are `PatientRecord`, `MyStructName` and `BitString`. By also strictly adhering to the convention that variable names start with a lowercase letter (e.g., `aPatientRecord`, `myStructureObject` and `bitString`) the risk of confusing variable and type names is eliminated.

It is not necessary to name the data members first and second, as in the example. This was done primarily to highlight the similarity between this implementation and the one using `std::pair` (listing refls:std::pair). Prefer to use descriptive names, as for other variables, such as `strPatientID` and `dBloodPressure`. Some programmers prefer to start all names of member variables with an '`m`' or other character so that they can easily distinguish them from other variables. It is also not necessary to have exactly two data members: a `std::pair` is a structure with just two member variables, but in general, structures can contain as many variables as you like.

To Go Further
Why must we end structures with a semicolon?

To extend **`struct`** `PatientRecord` with additional members, just add their specifications to the definition of the **`struct`**. For example, the code below defines a structure with two additional member variables, one **`double`** to store a temperature measurement and a **`bool`** to keep track of whether the patient was in a treatment or a control group in an experimental study. In this extended example, the definition of the **`struct`** is immediately followed by a declaration of two variables, `aPatientRecord` and `anotherPatientRecord` (which are two separate instances of `PatientRecord`). The possibility to declare **`struct`** variables in this way explains why the closing brace of the struct specification on line 19 of listing 11.2 is followed by a semicolon. Put differently, it is unnecessary to write a semicolon after a compound statement or function body, but the compiler expects a semicolon after the closing brace of a **`struct`** or **`class`** definition to mark the end of a (potentially empty) declaration list.

```
struct PatientRecord {
     std::string mstrPatientId;
     double mdBloodPressure, mdTemperature;
     bool mInTreatmentGroup;
} aPatientRecord, anotherPatientRecord;
```

11.3 A first look at classes

Structures are essentially an *alter ego* of C++ class objects, which form the basis for object-oriented programming. Actually, in the current version of C++, the definition of a structure is internally translated by the compiler to the definition of a class. Structures and classes are so similar that you might wonder why the language allows for two redundant ways of defining objects. One reason is that the maintenance of both structures and classes ensures compatibility with the predecessor language C; another (potential) benefit of separating structures and classes is that their specifications are able to evolve in different directions in future versions of C++. In practice, programmers use structures for relatively simple objects that merely serve to keep together collections of related variables, whereas classes are used for more sophisticated objects with member functions and data members that are shielded off from external access. Probably without realising it, you have already worked with such sophisticated objects: `std::vector`, `std::string`, and stream objects like `std::cout` are all defined as class objects with member functions and internal data.

Class definitions look very similar to the definitions of structures, except that they start with the keyword **`class`**. For example,

```
1    class PatientRecord {
2    public:
3        std::string mstrPatientId;
4        double mdBloodPressure, mdTemperature;
5        bool mInTreatmentGroup;
6    } aPatientRecord, anotherPatientRecord;
```

is the exact equivalent of the definition of the extended `PatientRecord` structure introduced earlier. You will probably notice one additional difference on line 3. On this line the keyword **`public`** is included to indicate to the compiler that the following member variables can be accessed by code outside of **`class`** `PatientRecord`. The member variables of a **`struct`** are **`public`** by default, so it is not necessary to use **`public`** in the definitions of structures.

The counterpart of **`public`** is **`private`**, which is the default access specifier for the members of a class object. Access to **`private`** member variables of a class is restricted to code that belongs to the class. This allows for *data encapsulation*, i.e., a program design strategy that precisely controls the way data are manipulated in order to ensure that they are kept safe from outside interference and misuse. Data encapsulation is accomplished by designing classes with **`private`** data members and **`public`** member functions that allow access to the data from the outside in a controlled way. An illustration of this principle is provided in listing 11.4 which emulates a caricature of the interaction between a farmer (the external program) and a cow (a **`class`** object). The `Cow` object has an internal fat reserve that increases when the cow is fed, and that decreases when the cow produces milk. The external program (the 'farmer') cannot control the fat reserve of the cow directly (`mdFatReserve` is **`private`**) but it can interact with the `Cow` via its **`public`** member functions `feed()` and `produceMilk()`, and in this way influence the fat reserve indirectly. In particular, the more `food` is provided to the `Cow`, the more `milk` it produces, as evidenced by the program output:

```
feeding one unit of food per day:
day 10 : milk = 0.401263
day 20 : milk = 0.641514
day 30 : milk = 0.785361
day 40 : milk = 0.871488
day 50 : milk = 0.923055
day 60 : milk = 0.95393
day 70 : milk = 0.972416
day 80 : milk = 0.983485

feeding two units of food per day:
day 90 : milk = 1.39137
day 100 : milk = 1.63559
day 110 : milk = 1.78182
day 120 : milk = 1.86937
day 130 : milk = 1.92178
day 140 : milk = 1.95317
day 150 : milk = 1.97196
day 160 : milk = 1.98321
```

The program in listing 11.4 consists of three parts, representing, respectively, the interface, implemen-

tation and application of the **class** Cow. The first part (line 4-11) contains the definition of the **class** Cow. This gives you an overview of the ways the program can interact with the class (i.e., its interface, as determined by the list of **public** member functions) and the kind of data that are contained within each class object (the **private** member variables). The implementation (line 15-26) lists the definitions of the class member functions, where you can look up exactly how the member functions interact with the data contained in the class. The definition of a member function has the same syntax as a normal function definition, except that the name of the function is preceded by the name of the class and the scope resolution operator :: to indicate that the function belongs to a specific class. The member functions therefore have access to the **private** variables, as you can see on line 17 and 23-24. The third part of the listing (line 29-54) contains the code for the application that uses the class object. This piece of the program has access to the public class members only. In larger programs, the class definition, its implementation and the user's application of the class are typically kept separate in different files.

Code listing 11.4: Data encapsulation

```
1  #include <iostream>
2
3  //*** class definition **********************
4
5  class Cow {
6  public:
7      void feed(double);
8      double produceMilk();
9  private:
10     double mdFatReserve;
11 };
12
13 //*** definitions of the member functions ***
14
15 void Cow::feed(double food)
16 {
17     mdFatReserve += food;
18 }
19
20 double Cow::produceMilk()
21 {
22     // 5 % of the fat reserve is used for producing milk
23     const double milk = 0.05 * mdFatReserve;
24     mdFatReserve -= milk;
25     return milk;
26 }
27
28 //*** main() program ************************
29 int main()
30 {
31     // declare a Cow
32     Cow myCow;
33
34     // feed myCow one unit of food per day
35     std::cout << "feeding one unit of food per day:\n";
36     for(int day = 1; day <= 80; ++day) {
37         myCow.feed(1.0);
38         const double milk = myCow.produceMilk();
39
40         if(day % 10 == 0)
41             std::cout << "day " << day << " : milk = " << milk << '\n';
42     }
43
44     // start feeding myCow two units of food per day
45     std::cout << "\nfeeding two units of food per day:\n";
46     for(int day = 81; day <= 160; ++day) {
47         myCow.feed(2.0);
48         const double milk = myCow.produceMilk();
49
```

```
50          if(day % 10 == 0)
51              std::cout << "day " << day << " : milk = " << milk << '\n';
52      }
53      return 0;
54  }
```

11.4 Operating on objects

Besides encapsulation, a huge advantage of working with objects is that they almost automatically induce structure in your coding and program design. With some experience, this will help you to write reusable and concise code that highlights the logical structure of your program. In a nutshell: *aim to delegate all the nitty-gritty details of how to deal with the member variables of a class to the class implementation; once you have completed this part of code, switch your mind to thinking about the objects as logical units on their own and describe the operations that occur on them and they ways they interact.* We will cover this technique for working with objects in depth in Module 12, *Data encapsulation and object design*. Here, we focus on the necessary basis of operations involving objects: passing and returning objects to and from functions, and copying one object into another.

We first show how to use objects as parameters and return values of functions. To start simple, the following example introduces a structure to represent a colour in RGB format and a function `blend()` that mixes together two RGB colour objects; the colour of the resulting blend is returned by the function as another RGB colour object.

```
1  #include <iostream>
2
3  struct RGBColour {
4      unsigned char r, g, b;
5  };
6
7  RGBColour blend(RGBColour first, RGBColour second)
8  {
9      RGBColour mix;
10     mix.r = (first.r + second.r) / 2u;
11     mix.g = (first.g + second.g) / 2u;
12     mix.b = (first.b + second.b) / 2u;
13     return mix;
14 }
15
16 int main()
17 {
18     RGBColour red = {255u, 0u, 0u}, yellow = {255u, 255u, 0u};
19
20     RGBColour orange = blend(red, yellow);
21
22     return 0;
23 }
```

The definition of the function `blend()` illustrates that you can declare objects as function parameters in exactly the same way as you would normally deal with arguments of basic data types. Moreover, the normal syntax rules apply for using a **`struct`** or **`class`** name as the return type for a function. All of this implies that the compiler automatically knows how to make copies of **`struct`** or **`class`** objects. It does so by using a so-called default copy constructor, which simply copies the data members one by one. In addition, the compiler will rely on a default interpretation of the assignment operator, which will be to assign the value of each member variable of the object on the right-hand side of the assignment to the corresponding member variable of the object on the left-hand side. In other words, an assignment `left = right;` for two `RGBColour` objects (as on line 20 of the listing above) is shorthand for:

```
1 left.r = right.r;
2 left.g = right.g;
3 left.b = right.b;
```

Objects can also be passed to member functions of their own class, as demonstrated by the following variant of the algorithm for blending RGB colours

```
1  #include <iostream>
2
3  class RGBColour {
4  public:
5      void define(unsigned char, unsigned char, unsigned char);
6      RGBColour blend(const RGBColour&);
7  private:
8      unsigned char mR, mG, mB;
9  };
10
11 void RGBColour::define(unsigned char r, unsigned char g, unsigned char b)
12 {
13     mR = r;
14     mG = g;
15     mB = b;
16 }
17
18 RGBColour RGBColour::blend(const RGBColour &other)
19 {
20     RGBColour mix;
21     mix.mR = (mR + other.mR) / 2u;
22     mix.mG = (mG + other.mG) / 2u;
23     mix.mB = (mB + other.mB) / 2u;
24     return mix;
25 }
26
27 int main()
28 {
29     RGBColour red, yellow, orange;
30
31     red.define(255u, 0u, 0u);
32     yellow.define(255u, 255u, 0u);
33
34     orange = red.blend(yellow);
35
36     return 0;
37 }
```

In this variant, `RGBColour` is defined as a class with **`private`** data members. Therefore, a **`public`** member function `RGBColour::define()` had to be added, in order to be able to set the initial RGB values (in Module **??** you will learn about constructors, which deal with the initialisation of **`class`** objects in a more natural way). The function `blend()` was also added to the **`class`** `RGBColour`, so that it has access to the private data members. As a result, the function is called in a different way on line 34 of the program. This statement calls the member function `blend()` of the object `red`, and passes object `yellow` to the function as an argument. In this case, the argument is passed by **`const`** reference, which prevents the compiler from making a copy of the object at the time of the function call.

11.5 Exercises

11.1. Elementary operations on a std::pair and a user-defined structure.

(a) Declare a `std::pair` that contains an integer and a character variable, and create an instance of it. Allow the user to set the values of the member variables `first` and `second` via keyboard input. Finally pass the `std::pair` object to a function that prints the values of the member variables to the screen.

(b) Replace the `std::pair` by a home-made structure.

11.2. Elementary operations on a class object. Replace the **struct** of the previous exercise by a **class** in which the member variables are **private**. Add appropriate **public** member functions to take care of input and output operations on the member variables, exactly as in Exercise 11.1.

11.3. The PGCPPPHD phone directory (⋆). Use listing 11.2 as an inspiration for writing a C++ program that serves as a search engine for the phone directory of the Pan-Galactic C++ Programming Help Desk (PGCPPPHD). Your program should import data from a file `phonedirectory.txt` with the following contents:

```
ET          6133
Arnold      8093
Jamie       8094
Yoda        8097
Nefertiti   8780
```

The data from the file should be loaded into a `std::vector` of **struct** objects, each containing a name and an extension. After loading the database, the program should allow the user to enter a name, and it should then show the corresponding extension on the screen (or an appropriate message if the requested name is not listed in the PGCPPPHD).

Test your skill

11.4. Complex number arithmetic (⋆). Complex numbers can be implemented in C++ as structures that contain two floating point values to represent the real and imaginary parts of a complex number $z = a + b\,i$ (with $z \in \mathbb{C}$, $a = \mathrm{Re}(z) \in \mathbb{R}$ and $b = \mathrm{Im}(z) \in \mathbb{R}$). Declare a **struct** `ComplexNumber` and write functions that operate on it to

- print the value of a complex number $z = a + b\,i$ to the screen in the ordinary notation $a + b\,i$.
- compute the norm of a complex number $||z|| = \sqrt{a^2 + b^2}$.
- compute the conjugate of a complex number $\bar{z} = a - b\,i$.
- calculate the ratio between a complex number and a real number, $z/x = a/x + (b/x)\,i$.
- calculate the product of two complex numbers $(a + b\,i)(c + d\,i) = (ac - bd) + (ad + bc)\,i$.
- calculate the ratio of two complex numbers $y/z = (y\bar{z})/||z||^2$.

Test your code by verifying that it correctly evaluates

$$\frac{5 + 5\,i}{3 - 4\,i} = -\frac{1}{5} + \frac{7}{5}\,i$$

Building up complex data types

The variables contained in classes can be class objects themselves, and classes may define their own data types. In this way, C++ offers a nearly unlimited flexibility to design complex data structures. One example that you have already seen is the definition of a matrix that is built by creating a `std::vector<std::vector<double> >` object. For instance,

```
std::vector<std::vector<double> > matrix(2, std::vector<double>(3, 0.0)
```

defines a 2×3 matrix of **double**s whose elements are initialised with the value `0.0`. Listing 11.5 shows another example of a complex data type. Here, the programmer developed a **class** `PatientDatabase` that contains a `std::vector` of patient records. The **struct** `PatientRecord` introduced earlier is used for the individual entries in the database.

Code listing 11.5: Coding a patient database

```
1 #include <iostream>
2 #include <fstream>
3 #include <cstdlib>
4 #include <vector>
5 #include <string>
```

```
6
7  //*** definition of the class PatientDatabase *******
8
9  class PatientDatabase {
10 public:
11     struct PatientRecord {
12         std::string mstrPatientID;
13         double mdBloodPressure;
14     };
15     void addRecord(const PatientRecord&);
16     bool retrievePatientData(PatientRecord&);
17 private:
18     std::vector<PatientRecord> mDatabase;
19 };
20
21 //*** implementation of the class PatientDatabase ***
22
23 void PatientDatabase::addRecord(const PatientRecord &record)
24 // adds a PatientRecord to the end of the database
25 {
26     mDatabase.push_back(record);
27 }
28
29 bool PatientDatabase::retrievePatientData(PatientRecord &record)
30 // returns true and stores patient data in record if a matching database item is present;
31 // otherwise returns false
32 {
33     for(int i = 0; i < mDatabase.size(); ++i)
34         if(mDatabase[i].mstrPatientID == record.mstrPatientID) {
35             record.mdBloodPressure = mDatabase[i].mdBloodPressure;
36             return true;
37         }
38
39     // if patient was not found...
40     return false;
41 }
42
43 //*** application *********************************
44
45 void loadDatabase(PatientDatabase &database, const std::string &filename)
46 {
47     // open file
48     std::ifstream ifs(filename.c_str());
49     if(!ifs.is_open()) {
50         std::cerr << "error: unable to open datafile patientData.txt\n";
51         exit(EXIT_FAILURE);
52     }
53
54     // read data from file to fill database
55     for(;;) {
56         // read record from file
57         PatientDatabase::PatientRecord record;
58         ifs >> record.mstrPatientID;
59         if(ifs.fail()) break;
60         ifs >> record.mdBloodPressure;
61
62         // add record to myPatientDataBase
63         database.addRecord(record);
64     }
65 }
66
67 int main()
68 {
69     // create database and load data from file into it
70     PatientDatabase myPatientDatabase;
71     loadDatabase(myPatientDatabase, "patientData.txt");
```

```
72
73      // allow user to retrieve patient data from database
74      for(;;) {
75          // prompt for user input
76          std::cout << "please enter a patientID (or type stop to quit).\n> ";
77          PatientDatabase::PatientRecord record;
78          std::cin >> record.mstrPatientID;
79
80          // terminate loop, or retrieve patient data
81          if(record.mstrPatientID == "stop")
82              break;
83          else {
84              const bool isKnownPatient = myPatientDatabase.retrievePatientData(record);
85
86              if(isKnownPatient)
87                  std::cout << "patient " << record.mstrPatientID
88                            << " has blood pressure " << record.mdBloodPressure << "\n\n";
89              else
90                  std::cout << "patient " << record.mstrPatientID
91                            << " was not found in database.\n\n";
92          }
93      }
94      return 0;
95  }
```

Some output of the program (with input from a user) is listed below:

```
please enter a patientID (or type stop to quit).
> pf00001
patient pf00001 has blood pressure 118.4

please enter a patientID (or type stop to quit).
> pf00500
patient pf00500 was not found in database.

please enter a patientID (or type stop to quit).
> stop
```

The program in listing 11.5 introduces a number of novel features, so let's walk through the three parts of the code (interface, implementation and user application) step by step. First take a look at the **public** part of the definition of **class** PatientDatabase (line 10-16). As you can see, the class contains two public member functions, one for adding records to the database, another for retrieving patient data. Also part of the **public** part of the class is the definition of the **struct** PatientRecord. As a consequence, it is possible to declare instances of **struct** PatientRecord outside the class **class** PatientDatabase, as is done for the local variables record on line 57 and 77. Note that, since the definition of type PatientRecord belongs to **class** PatientDatabase, the type name has to be specified as PatientDatabase::PatientRecord in such cases.

The implementation of the **class** PatientDatabase illustrates how structures are passed to functions. In the function PatientDatabase::addRecord(), the function argument record is a **const** reference, so that the patient record is not copied when it is passed to the member function but also not changed by the function. A copy will eventually be created by passing record to std::vector::push_back() which adds a copy of the original PatientRecord to the end of mDataBase. The member function PatientDatabase::retrievePatientData() receives a patient ID as part of the PatientRecord record that is passed to the function. It then attempts to find the patient in the database, and adds the patient's blood pressure to the PatientRecord that was originally passed to the function. For this to work correctly, the function argument *must* be passed by reference.

The third part of the program shows how intuitive it is to work with objects of the **class** PatientDatabase once the class implementation has been written. An instance of the class called myPatientDatabase is created on line 70, which is passed to the function loadDatabase() on line 71, together with the name of a file with patient data. This function opens the file and adds records to myPatientDatabase, since this object is passed to the function by reference. Once the database has been loaded, the program enters a loop (line 74-93) in which the user is prompted to enter patient IDs. The user input is read straight into a PatientDatabase::PatientRecord object (line 78), which is subsequently passed to the member function retrievePatientData() of myPatientDatabase.

More exercises

11.5. Simulating a farm (⋆⋆). Use the `class Cow` from listing 11.4 as a basis for simulating a farm. To do this, develop a `class Farm` that contains a `private herd` of `Cows` and a variable `money` of type `double`. Implement the following behaviours by means of `public` member functions:

(a) The farmer can invest money into the `Farm` to realise a positive initial value of `money`, sufficient to buy at least one `Cow`.

(b) The farmer can sell a `Cow` from the `herd` at the weekly livestock market, in order to increase the value of `money`;

(c) The farmer can buy a `Cow` at the livestock market and add it to the `herd`, which reduces the value of `money`;

(d) The farmer can use `money` to buy feed for the animals, which is distributed every day among the `Cows` in the `herd`;

(e) The farmer milks the `Cows` everyday, and sells the milk to make `money`.

Use your rural wit (if any) to implement clever rules for buying and selling cows, depending on the current market prices of livestock, animal feed and milk. For simplicity, you can initially assume these to be given and simulate a single `Farm` over the course of a year (for example, keep track of the development of the size of the herd). *Optional Challenge:* simulate multiple `Farms` and allow the law of supply and demand to set the market price for milk.

11.6 Solutions to the exercises

Exercise 11.1. Elementary operations on a std::pair and a user-defined structure. Here is the implementation using `std::pair`:

```
#include <iostream>
#include <utility>

void print(const std::pair<int, char> &myPair)
{
    std::cout << "value of the int  member: " << myPair.first << '\n';
    std::cout << "value of the char member: " << myPair.second << '\n';
}

int main()
{
    // declare std::pair object
    std::pair<int, char> myObject;

    // set values via keyboard input
    std::cout << "please enter a value for the int member.\n> ";
    std::cin >> myObject.first;
    std::cout << "please enter a value for the char member.\n> ";
    std::cin >> myObject.second;

    // print values
    print(myObject);
    return 0;
}
```

The equivalent implementation using a structure is:

```
#include <iostream>

struct MyStructure {
    int first;
    char second;
};

void print(const MyStructure  &myStruct)
{
    std::cout << "value of the int  member: " << myStruct.first << '\n';
    std::cout << "value of the char member: " << myStruct.second << '\n';
}

int main()
{
    // declare struct object
    MyStructure myObject;

    // set values via keyboard input
    std::cout << "please enter a value for the int member.\n> ";
    std::cin >> myObject.first;
    std::cout << "please enter a value for the char member.\n> ";
    std::cin >> myObject.second;

    // print values
    print(myObject);
    return 0;
}
```

Exercise 11.2. Elementary operations on a class object.

```
#include <iostream>

class MyClass {
public:
    void setValues();
    void print();
private:
    int first;
    char second;
};

void MyClass::setValues()
{
    std::cout << "please enter a value for the int member.\n> ";
    std::cin >> first;
    std::cout << "please enter a value for the char member.\n> ";
    std::cin >> second;
}

void MyClass::print()
{
    std::cout << "value of the int  member: " << first << '\n';
    std::cout << "value of the char member: " << second << '\n';
}

int main()
{
    // declare struct object
    MyClass myObject;

    // set values via keyboard input
    myObject.setValues();

    // print values
    myObject.print();
    return 0;
}
```

Exercise 11.3. The PGCPPPHD phone directory.

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <vector>
#include <string>

int main()
{
    // open file
    std::ifstream ifs("phonedirectory.txt");
    if(!ifs.is_open()) {
        std::cerr << "error: unable to open phonedirectory.txt\n";
        exit(EXIT_FAILURE);
    }

    struct PhoneRecord {
        std::string mstrName;
        int miExtension;
    };

    // read data from file
```

```
    std::vector<PhoneRecord> phoneDirectory;
    for(;;) {
        PhoneRecord tmp;
        ifs >> tmp.mstrName;
        if(ifs.fail()) break;
        ifs >> tmp.miExtension;
        phoneDirectory.push_back(tmp);
    }

    // allow user to search directory
    for(;;) {
        std::cout << "whom would you like to call? (or type stop to quit).\n> ";
        std::string strInput;
        std::cin >> strInput;

        // terminate loop, or look up phone number
        if(strInput == "stop")
            break;
        else {
            int i;
            for(i = 0; i < phoneDirectory.size(); ++i)
                if(phoneDirectory[i].mstrName == strInput) {
                    std::cout << phoneDirectory[i].mstrName << " can be reached on ext. "
                              << phoneDirectory[i].miExtension << "\n\n";
                    break;
                }
            if(i == phoneDirectory.size())
                std::cout << "sorry, " << strInput << " is not listed in the PGCPPPHD\n\n";
        }
    }
    return 0;
}
```

Exercise 11.5. Simulating a farm.

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

double kdMarketValueCow = 600.0, kdMarketValueFeed = 1.0;
double dMarketValueMilk = 2.0;

//*** definition of the class Farm ****************

class Farm{
public:
    void invest(const double);
    void sellCow(const int);
    void buyCow(const int);
    void buyFeed();
    void sellMilk(double&);
    int getHerdSize() {return mHerd.size();}
private:
    class Cow {
    public:
        void feed(double);
        double produceMilk();
    private:
        double mdFatReserve;
    };
    std::vector<Cow> mHerd;
    double mdMoney;
};

```

```
29 //*** definitions of Farm member functions ********
30
31 void Farm::invest(const double money)
32 {
33     mdMoney = money;
34 }
35
36 void Farm::sellCow(const int t)
37 {
38     // sell a cow if it is market day and feed is more expensive than milk
39     if(t % 7 == 0 && kdMarketValueFeed > dMarketValueMilk && !mHerd.empty()) {
40         mHerd.pop_back();
41         mdMoney += kdMarketValueCow;
42     }
43 }
44
45 void Farm::buyCow(const int t)
46 {
47     // buy a cow if it is market day and milk is more expensive than feed
48     if(t % 7 == 0 && kdMarketValueFeed < dMarketValueMilk && mdMoney >= kdMarketValueCow) {
49         mdMoney -= kdMarketValueCow;
50         mHerd.push_back(Cow());
51     }
52 }
53
54 void Farm::buyFeed()
55 {
56     if(mHerd.empty()) return;
57
58     // buy one unit of feed per cow...
59     double feed = mHerd.size();
60     // ...or less if there's not enough money
61     if(mdMoney < feed * kdMarketValueFeed)
62         feed = mdMoney / kdMarketValueFeed;
63
64     mdMoney -= feed * kdMarketValueFeed;
65
66     // feed cows
67     double feedRation = feed / mHerd.size();
68     for(int i = 0; i < mHerd.size(); ++i)
69         mHerd[i].feed(feedRation);
70 }
71
72 void Farm::sellMilk(double &dMilkTotal)
73 {
74     // milk cows
75     double dMilk = 0.0;
76     for(int i = 0; i < mHerd.size(); ++i)
77         dMilk += mHerd[i].produceMilk();
78
79     // sell milk on market
80     mdMoney += dMilk * dMarketValueMilk;
81
82     dMilkTotal += dMilk;
83 }
84
85
86 //*** definitions of Farm::Cow member functions ***
87
88 void Farm::Cow::feed(double food)
89 {
90     mdFatReserve += food;
91 }
92
93 double Farm::Cow::produceMilk()
94 {
```

```
95      // 5 % of the fat reserve is used for producing milk
96      const double milk = 0.05 * mdFatReserve;
97      mdFatReserve -= milk;
98      return milk;
99  }
100
101 //*** main() program *****************************
102 int main()
103 {
104     const int nFarms = 100;
105     const int nDays = 365;
106
107     // create Farms and endow each of them with a start-up capital
108     std::vector<Farm> vecFarms(nFarms);
109     for(int i = 0; i < nFarms; ++i)
110         vecFarms[i].invest(10 * kdMarketValueCow);
111
112     for(int t = 0; t < nDays; ++t) {
113         // run farm business
114         int iCowsTotal = 0;
115         double dMilkTotal = 0.0;
116         for(int i = 0; i < nFarms; ++i) {
117             vecFarms[i].buyCow(t);
118             vecFarms[i].buyFeed();
119             vecFarms[i].sellMilk(dMilkTotal);
120             vecFarms[i].sellCow(t);
121             iCowsTotal += vecFarms[i].getHerdSize();
122         }
123         // set market price for milk
124         dMarketValueMilk = 0.05 + 1000.0 / (dMilkTotal + 0.05);
125
126         std::cout << t << ' ' << dMarketValueMilk << ' ' << iCowsTotal << '\n';
127     }
128
129     return 0;
130 }
```

Module 12

Data encapsulation and object design

Jamie explains what's on the menu

In this module you will:

- Become familiar with various ways to control the access rights to class data.
- See examples of how to design versatile classes without compromising the benefits of data encapsulation.
- Learn how to overload operators in such a way that they can applied to home-made class objects.

12.1 Introduction

An advantage of object-oriented-programming is that it enforces a particular organisational structure on your program. When you are used to procedural programming, it may take some time to appreciate the advantages of this structure, but once you get the point, object-oriented programming will help you to develop complex simulations in an efficient way. A critical first step is that you carefully design the objects that form the basis of your simulation algorithm. You want the objects to deal with the necessary computational work automatically as far as possible, so that you can focus your attention on a higher level of organisation – without worrying about the details of how the objects handle their task internally. This is accomplished by equipping your objects with internal functionality and by controlling how the user can interact with the object.

In order to make these abstract ideas a bit more concrete, we will illustrate in this module how to build up a complex `class` object with advanced functionality. The different steps of the object building process will be illustrated for a concrete, worked-out example problem: simulating the interaction between players in an Iterated Prisoner's Dilemma (IPD) game. If you are not familiar with this game or the biological question of the evolution of cooperation, please first read the introduction offered by Kermit on p.126.

Our first step is to declare a `class` to represent a player in the IPD. By the end of the module, we will use objects of this `class` to simulate a tournament between competing strategies, in order to find out which strategy outperforms the others. But first, we focus on the objects themselves, and define a `class PlayerIPD` with two `private` member variables, `bool coopInit`, and `std::array<bool, 4> coopResp` (see listing 12.1). The first variable determines the action of the player in the initial round of the IPD (with cooperation represented by `true` and defection by `false`), the second variable is an array with four

boolean variables that specifies the player's actions contingent on the outcome of the previous round.

***Kermit the Frog* explains the iterated prisoner's dilemma**

The problem of the emergence and maintenance of cooperation in groups of self-interested individuals is relevant for understanding human and animal societies, as well as major evolutionary transitions, such as the evolution of multicellularity. Explaining why individuals cooperate is particularly challenging when group-interests are opposite to the individual interests of group members. As an example of such a social dilemma, consider a group of fishermen who all benefit from moderating the total harvest from a fish stock in order to prevent the stock population from collapsing; yet, each individual fisherman has an incentive to harvest more than his colleagues, in order to maximise his profit.
In economics and evolutionary game theory, social dilemmas are frequently studied by analysing a simple caricature model, known as the prisoner's dilemma game. The game considers two members of a criminal gang who are held in custody by the police. Each prisoner is locked up in a private cell and has no means of communicating with the other. The prosecutors do not have sufficient evidence to convict the pair on the principal charge (i.e., murdering several members of a rival gang), but they can get both convicted on a lesser charge (armed robbery). Therefore, the prosecutors decide to offer each prisoner a bargain. In return for testifying against their accomplice, the prosecutor promises each prisoner an acquittal of the minor charges. Each prisoner is now faced with the following dilemma:

- If I testify against my accomplice, but my partner does the same, I will be convicted for the murder charges and sent to jail for a sentence of P years.
- If I testify against my accomplice, but my partner remains silent, I will be released immediately.
- If I cooperate with my accomplice by remaining silent, but my partner testifies against me, I will be convicted for murder and armed robbery. The jail sentence will then amount to $P + p$ years (with p being the sentence for armed robbery).
- If neither of us testify against the other, we will only be convicted for the minor charges. Each of us will then only spend p years in jail (with $p < P$).

Assuming that the prisoners have no opportunity to reward or punish their partner in some other way or at another time in the future, the rational decision for each prisoner is to testify against his accomplice. This is because betraying a partner offers a greater reward than cooperating with him, irrespective of what the partner decides to do (i.e., the first option is preferable over the third, and the second is preferable over the fourth). However, the consequence is that both criminals spend P years in jail, whereas they could have have gotten away with a much lower sentence of p years if they had cooperated with each other.
Whereas defection (i.e., testifying against your partner) is the rational choice in a one-shot prisoner's dilemma game, the situation changes markedly if the same pair of players has to play the game repeatedly. Then, defection becomes much less attractive, because betraying your partner may induce him to take revenge in future interactions. Indeed, simulations of the *iterated* prisoner's dilemma (IPD) confirm that cooperative strategies can gain the upper hand if players have the opportunity to make their choices dependent on past interactions. A simple but particularly successful strategy in the IPD is tit-for-tat (TFT), i.e., cooperate in the first interaction, but afterwards do whatever your partner did in the previous round of the game. This result supports the idea that reciprocity is fundamental to the maintenance of cooperation in long-term relationships.

For simplicity, a player's behaviour is assumed to be deterministic (players are restricted to play a pure strategy), and each player can only remember the outcome of the previous round. The four

potential outcomes of the last interaction will be ordered in the same way as in Kermit's explanation of the prisoner's dilemma. That is, a player's actions are specified as follows:

`coopInit`	action in the first round of the game;
`coopResp[0]`	response after mutual defection in the previous round;
`coopResp[1]`	response when a player defected in the previous round while the partner cooperated;
`coopResp[2]`	response when a player cooperated in the previous round while the partner defected;
`coopResp[3]`	response after mutual cooperation in the previous round;

To simplify the type notations in the class definition, we introduce two **`typedef`** statements

`typedef bool` `Action;` and

`typedef` `std::array<Action, 4> Response;`

in the **`public`** part of the class (see line 3-4 of listing 12.1). The first allows us to reuse the same **`class`** definition, if we would like to model stochastic response strategies in the future (actions could then be represented by a **`double`** to specify the probability of cooperating). The second primarily helps to simplify the type notations; it allows us to use `Response` instead of `std::array<bool, 4>` as a typename inside the class definition and `PlayerIPD::Response` outside of it. The initial class definition also contains two member functions (also referred to as *methods*) that allow a user to set the strategy of a player and to show it on the screen.

Code listing 12.1: Initial specification of the class PlayerIPD

```
1  class PlayerIPD {
2  public:
3      typedef bool Action;
4      typedef std::array<Action, 4> Response;
5
6      void setStrategy(Action, const Response&);
7      void showStrategy();
8  private:
9      Action coopInit;
10     Response coopResp;
11 };
12
13 void PlayerIPD::setStrategy(Action naiveAction, const Response& responseAction)
14 {
15     coopInit = naiveAction;
16     coopResp = responseAction;
17 }
18
19 void PlayerIPD::showStrategy()
20 {
21     std::cout << "0 -> " << (coopInit ? 'C' : 'D');
22     for(int i = 0; i < coopResp.size(); ++i)
23         std::cout << ", " << (i / 2 ? 'C' : 'D') << (i % 2 ? 'C' : 'D')
24                   << " -> " << (coopResp[i] ? 'C' : 'D');
25     std::cout << '\n';
26 }
```

Already at this early stage of development, it is important to test the **`class`**. The following is a simple program that defines a player with the strategy tit-for-tat. (make sure to also to include the headers `iostream` and `array` when you piece together the code for the complete program)

```
1  int main()
2  {
3      PlayerIPD titForTat;
4      titForTat.setStrategy(true, {false, true, false, true});
5      titForTat.showStrategy();
6      return 0;
7  }
```

Its output is given by:

```
0 -> C, DD -> D, DC -> C, CD -> D, CC -> C
```

12.2 Constructors

If a **class** object is declared as it is on line 3 of the test program above, memory is allocated for all of its member variables, but the variables themselves are not initialised. In fact, the data members are assigned a value for the first time when the body of the function `PlayerIPD::setStrategy()` is executed. This situation is equivalent to when a simple variable is first declared but only later assigned a first value, such as here:

```
1 int i;
2 // some lines of code
3 i = 41;
```

As you may recall, it is normal practice in C++ to combine the declaration of a variable with the assignment of its initial value. This also creates the opportunity to declare the variable as **const**. For example, if the value of **int** i does not need to change after its initial value has been set, we would typically prefer to declare the variable as

```
1 const int i = 41;
```

Can something similar be done for **class** objects as well? Yes it can, but we must first specify how an instance of the class should be initialised exactly. This is accomplished by defining a so-called *constructor*. A constructor is a function that is called automatically when an object is created. Classes can have multiple constructors, each handling a specific way by which an object is created. This explains why a `std::vector` object, for example, can be created in several different ways:

```
1 std::vector<int> vec1;              // calls a default constructor; produces empty vector
2 std::vector<int> vec2(3, 0);        // calls a fill constructor;
3 std::vector<int> vec3 {1, 2, 3};    // calls an initialiser list constructor
4 std::vector<int> vec4(vec3);        // calls a copy constructor
```

each situation is handled by a different constructor that implements a specific method for initialising the object.

Several constructors are generated automatically by the compiler for each **class**. These include a default constructor, enabling the creation of **class** instances in simple declaration statements, and constructors that allow objects to be moved or copied (these are needed when objects are passed to functions or returned from functions as return value). The implicitly generated constructors can be overloaded or disabled; this is necessary in certain advanced applications when the **class** stores the memory addresses of other variables. For **class** objects that contain elementary data types or elementary types stored in STL containers, however, we do not have to worry about these complications, and the implicitly defined constructors will work fine.

Further methods for initialising **class** instances (in addition to those already defined implicitly) can be created by defining one or multiple constructors explicitly. In order to do so, add the definition of the constructor to the **class** in the same way as you would otherwise add a member function. Two things are a bit special for constructors: they always have the same name as the **class** to which they belong and they never return a value; if there are multiple constructors, they must differ from each other in their argument list.

The initialisation performed by `PlayerIPD::setStrategy()` would be an obvious task to delegate to a constructor. Therefore, let us replace this function by a constructor `PlayerIPD::PlayerIPD()`. The code below shows the revised **class** definition with a prototype of the constructor on line 6 and its definition on line 13-17:

```
1  class PlayerIPD {
2  public:
3      typedef bool Action;
4      typedef std::array<Action, 4> Response;
5
6      PlayerIPD(Action, const Response&);
7      void showStrategy();
8  private:
9      Action coopInit;
10     Response coopResp;
11 };
12
13 PlayerIPD::PlayerIPD(Action naiveAction, const Response& responseAction)
14 {
15     coopInit = naiveAction;
16     coopResp = responseAction;
17 }
18
19 // The definition of void PlayerIPD::showStrategy() was omitted here...
20
21 int main()
22 {
23     PlayerIPD titForTat(true, {false, true, false, true});
24     titForTat.showStrategy();
25     return 0;
26 }
```

If you carefully compare the definition of the constructor, with the previous definition of `PlayerIPD::setStrategy()` on line 13-17 of listing 12.1, you can see that constructors, in their simplest form, follow exactly the same syntax as member functions, except that they do not have a return type.

While the above constructor function allows `PlayerIPD` objects to be initialised at declaration (as you can see on line 23), it does not yet combine the declaration and initialisation of the member variables of the `PlayerIPD` object. Memory is allocated for these variables when the constructor is called, but the variables only get their first values inside the body of the constructor. As a consequence, it is not possible to define the variables `coopInit` and `coopResp` as **`const`** members (verify for yourself that the compiler produces an error message if you try).

In order to initialise member variables of a **`class`** object at the moment that memory is allocated for them, we can make use of a special notation for defining the constructor. This notation is similar to a standard function declaration, except that a list of member variables and their initial values is provided immediately following the parameter list (i.e., just before the function body). The list starts with a colon (`:`), followed by one or more items separated by commas (`,`). Each item specifies an initial value for a member variable and consists of the identifier of the member variable followed by its initial value inside a pair of parentheses, as in `identifier(initial_value)`. The code fragment below shows the `PlayerIPD` constructor with a member initialisation list

```
1 PlayerIPD::PlayerIPD(Action naiveAction, const Response& responseAction) :
2     coopInit(naiveAction), coopResp(responseAction)
3 {}
```

Note that, in this case, the body of the constructor is empty, because all of the member variables are initialised in the member initialisation list and the constructor performs no other tasks. In cases like this, when the constructor is short, you can opt to include the constructor definition in the definition of the **`class`**, but this is not compulsory.

Listing 12.2 shows the updated definition of the **`class`** `PlayerIPD` after moving the definition of the constructor into the **`class`** definition; the constructor with initialisation list and empty function body appears on line 6. A second explicitly defined constructor has been added (on line 5), which initialises all response variables with the same value. This additional constructor is called on line 18 to initialise the objects `allC` (representing the player who cooperates unconditionally) and `allD` (representing an unconditional defector). The output of the test program generated for the strategies `titForTat`, `allC` and `allD` on line 20-22 is:

```
0 -> C, DD -> D, DC -> C, CD -> D, CC -> C
```

```
0 -> C, DD -> C, DC -> C, CD -> C, CC -> C
0 -> D, DD -> D, DC -> D, CD -> D, CC -> D
```

Now that the variables `coopInit` and `coopResp` are initialised in the constructor initialisation lists, they can be defined as **`const`** members of the class, as has been done on line 9 and 10 of the **`class`** definition in Code listing 12.2. This will prevent the variables from changing their value after they have been created and initialised. As mentioned earlier, initialisation in a constructor initialisation list is compulsory for all **`const`** member variables. For non-**`const`** members, you can choose whether you want to initialise them in a constructor initialisation list, assign a value to them in the body of the constructor; or leave them uninitialised (but see Nefertiti's advice below).

***ET* on `inline` member functions**

The option of specifying the body of a member function inside a **`class`** definition exists not only for constructors but for other member functions of a **`class`** as well. In fact, defining a function inside the definition of a **`class`** is equivalent to declaring the function as **`inline`** (see p. **??**). This will be interpreted by the compiler as a request to expand the code of the function at all points where the function is invoked, effectively replacing each function call by the body of the function. Inline definitions help to speed up program execution, especially for small member functions that are called frequently. Inline expansion of large functions will increase the size of your program a lot, and is therefore best avoided. The compiler will anyway ignore an **`inline`** request if the function contains loops or other complex program flow, if it is a recursive function, or if it contains **`static`** variables.

***Nefertiti* on defining multiple constructors**

When you define multiple constructors for a class, make an effort to ensure that all of them deliver the object in the same state of initialisation. In other words, if one constructor initialises a member variable, the other constructors should assign an initial value to that variable as well. Otherwise, member variables may or may not be initialised depending on which constructor was called to create an object, which is a sure recipe for initialisation errors.

Code listing 12.2: Definition of the class PlayerIPD with two inline constructors

```
1  class PlayerIPD {
2  public:
3      typedef bool Action;
4      typedef std::array<Action, 4> Response;
5      PlayerIPD(Action c) : coopInit(c), coopResp({c, c, c, c}) {}
6      PlayerIPD(Action ci, const Response& cr) : coopInit(ci), coopResp(cr) {}
7      void showStrategy();
8  private:
9      const Action coopInit;
10     const Response coopResp;
11 };
12
13 // The definition of void PlayerIPD::showStrategy() was omitted here...
14
15 int main()
16 {
17     PlayerIPD titForTat(true, {false, true, false, true});
18     PlayerIPD allC(true), allD(false);
19
20     titForTat.showStrategy();
21     allC.showStrategy();
22     allD.showStrategy();
23
24     return 0;
25 }
```

12.3 Sharing information between class instances: `static` members

In their simplest manifestation, C++ **`class`** objects are intended to group together a bunch of variables and to be able to manipulate them as a single object. Therefore, it makes sense for every instance of a **`class`** to have its own instance of the member variables so that, when the values of the member variables are changed for one object, this will have no effect on the member variables of another object. There can, however, also be parameters or variables that have the same value for all instances of a **`class`**. Including such common data as a member variable in a **`class`** would not be a very efficient use of memory space, because every instance of the **`class`** would contain a redundant copy of the shared data. The alternative of making the shared data available by means of global variables is also not optimal, because global variables can be accessed from everywhere, which makes them hard to deal with when debugging. For shared parameters, this problem can be mitigated by declaring the global data as **`const`**, but this will not be possible for shared variables (i.e., things that actually change during program execution).

A hybrid solution that offers the benefits of data encapsulation without requiring the redundant storage of shared values, is to declare the shared data as **`static`**. Data members that are **`static`** will be created only once, and all instances of the class will have shared access to them. Therefore, when one class object modifies a **`static`** data member, its value will be changed for all other instances as well. **`static`** data members can be both **`public`** (i.e., accessible from outside the **`class`**) or **`private`** (i.e., hidden away); **`public static`** member functions can be added to the class to provide controlled access to **`private static`** data, in the same way as we have seen for normal data members.

***Master Yoda* on `static class` members**

One technical issue to be aware of when working with **`static class`** members is that their declaration inside the class does not count as a definition of the variable (unlike what is usual for other variable declarations). This means that listing the variable in the **`class`** definition only informs the compiler that the **`static`** variable exists somewhere; it does not instruct the program to allocate memory resources to the variable. In fact, a separate declaration-definition outside the body of the **`class`** is necessary to request memory for the variable.

By thinking carefully about whether variables should be declared as **`public`** or **`private`**, whether their values should be modifiable or **`const`**, and whether they should be shared (**`static`**) or separately stored by each individual object, you can carefully tailor the access rights and storage specifications of **`class`** data members. The modified `PlayerIPD` class definition in code listing 12.3 illustrates several of the different options: the strategic variables `coopInit` and `coopResp` are defined as **`const`** (private) data members, because they are fixed for a given strategy; new member variables `frequency` and `avgPenalty` have been added (on line 13) to keep track of the frequency and average penalty (jail time) for each strategy - these member variable will be updated during the simulation of the tournament and are specific for each strategy, so the variables are non-**`const`** and non-**`static`** (i.e., not shared between class instances; note also that `frequency` and `avgPenalty` are initialised in the body of both constructors on line 5 and 7). Finally, two **`const static`** data members, `penaltyMajorCrime` and `penaltyMinorCrime`, have been added to store the values of the payoff parameters P and p, respectively. Because these variables are **`static`** class members, they are defined and initialised outside of the class definition, on line 17 and 18 of code listing 12.3. As you can see, the names of the two variables on these lines are preceded by the name of the class and the scope operator `::`. This is necessary to indicate that the two variables belong to **`class`** `PlayerIPD`.

In summary then, there are several ways by which you can tailor access to **`class`** data members and the way they are stored:

- Data members can be made **`public`** in order to allow direct access to them from outside of the **`class`**; the default recommendation is to offer only **`private`** access to data members.
- Data members can be declared **`const`**, in order to fix their value at initialisation; A **`const`** data member must always be initialised in a constructor initialisation list.
- Data members that are shared by all instances of a **`class`** can be declared as **`static class`** members. Variables of a **`class`** that are **`static`** always have to be defined separately outside of the **`class`**

Code listing 12.3: Different ways to fine-tune access to data members

```
class PlayerIPD {
public:
    typedef bool Action;
    typedef std::array<Action, 4> Response;
    PlayerIPD(Action c) : coopInit(c), coopResp({c, c, c, c}) { frequency = avgPenalty = 0.0; }
    PlayerIPD(Action ci, const Response& cr) : coopInit(ci), coopResp(cr)
       { frequency = avgPenalty = 0.0;}
    void showStrategy();
private:
    const Action coopInit;
    const Response coopResp;
    static const double penaltyMajorCrime, penaltyMinorCrime;
    double frequency, avgPenalty;
};

// definition of static data members
const double PlayerIPD::penaltyMajorCrime = 0.5;
const double PlayerIPD::penaltyMinorCrime = 0.1;
```

12.4 Managing access to a class via methods and `friend` functions

Read and write access to private data members

Similar to data members, member functions of classes can either be **`public`** or **`private`**. In most cases, data members are **`private`** in order to protect them against uncontrolled access (this is the idea of encapsulation), whereas the methods are **`public`**, because they provide an interface for the interaction between the class and the outside world. For instance, besides creating `PlayerIPD` objects (by means of the constructors) and displaying their strategy on the screen (by means of `showStrategy()`), users will probably need to be able to retrieve information on the average payoff of a strategy from the class object. Therefore, the **`public`** interface of the class must (at the very least) be equipped with a simple method `getAvgPenalty()` to provide read access to the `avgPenalty` variable:

```
class PlayerIPD {
public:
    ...
    double getAvgPenalty() { return avgPenalty; }
private:
    ...
    double avgPenalty;
};
```

Read-access to the member variable `frequency` is useful to have as well, but in addition, the user will probably need some way to set the initial frequency of each strategy. There are different ways to provide this functionality. If frequencies have to be initialised only once, the best solution is to provide read-access by means of a method **`double`** `getFrequency()`) and handle the initialisation via an extra argument to the constructor. Alternatively, we can add separate member functions to handle read and write access to the variable `frequency`:

```
1 class PlayerIPD {
2 public:
3     ...
4     double getFrequency()         { return frequency; } // provides read access
5     void setFrequency(double frq) { frequency = frq;  } // provides write access
6 private:
7     ...
8     double frequency;
9 };
```

The above solution is equivalent to a third option, which is to provide read and write access through a single method that returns a reference to the member variable (see line 4 in the following code listing). The two ways of using the method are illustrated on line 14 and 15. Note that on line 14, the call to the member function appears on the left-hand side of an assignment; this is possible because the method returns a reference, i.e., the function call serves as an alias for the member variable `frequency`. As a result, the assignment has an effect equivalent to `titForTat.frequency = 0.5;`, as if `frequency` were a **`public`** data member.

```
1  class PlayerIPD {
2  public:
3      ...
4      double& frq() { return frequency; } // provides read and write access
5  private:
6      ...
7      double frequency;
8  };
9
10 int main()
11 {
12     PlayerIPD titForTat(true, {false, true, false, true});
13
14     titForTat.frq() = 0.5; // write access
15     std::cout <<  "initial frequency = " << titForTat.frq() << '\n'; // read access
16
17     return 0;
18 }
```

***Nefertiti* on the class user interface**

The **`public`** section of a class definition serves as the *user interface* to the class. When an experienced programmers works with classes that were designed by someone else, she will first look at the items in this user interface to find out in what different ways her program can interact with the class. Supporting the code with comments and using self-explanatory function names is therefore particularly important in the **`public`** section of the class definition.

A key design objective in object-oriented-programming is to keep the user interface as simple as it can be, without compromising the functionality and flexibility required by the user. All aspects that have to do with the inner workings of the class (about which the user can remain more or less ignorant) can therefore be best delegated to the **`private`** section of the class definition. This also applies to member functions. Imagine, for example, that you would like to prevent the repetition of a block of code that is executed by multiple class members. Delegating this block of code to a new member function can then be helpful, but such a member function need not be directly accessible to the user. The code below illustrates this idea by introducing a new **`private`** method **`void`** `initialise()` (on line 10), which takes care of the initialisation of the member variable `avgPenalty`. Since this function is called from within the function bodies of the two constructors (on line 5 and 6), it is not necessary for the user to have access

to `initialise()` directly. Therefore, in order to keep the **`public`** section of the class definition as clean as possible, `initialise()` is included as a **`private`** member function.

`const` methods

It is important to realise that all member functions, regardless of whether they are **`public`** or **`private`**, have access to all members in the **`public`** and **`private`** section of a class. Some members will make use of these access rights to manipulate the information stored in a class object, such as the constructors or the function `initialise()` in code listing 12.4, which assign initial values to some of the member variables. Other member functions that we have seen, such as `getAvgPenalty()` or `showStrategy()` (see listing 12.1), merely employ their access rights to read data from a class object but do not change the information it contains. It is good practice to designate to such functions the special status of a **`const`** method. **`const`** methods can be executed by all instances of a class, even if those instances themselves are declared to be **`const`** objects. By contrast, normal, non-**`const`** methods have read and write access to an object, and can therefore only be executed by non-**`const`** instances of a class.

Code listing 12.5 illustrates how the definition of **`const`** methods allows for the passing of an object of type `PlayerIPD` to a function by **`const`** reference, our preferred way of passing variables to functions. Passing by reference is particularly useful for large class objects, because it prevents the object data from being copied upon initialisation of the function argument. As usual, if we want to rule out that the original object is modified by the function call, we should pass the function argument by **`const`** reference. However, when we attempt to do this for a class object, the function is not longer allowed invoke its member functions, unless they are declared as **`const`**.

To see how this works in practice, first have a look at the function `report()` on line 29-34 of code listing 12.5. This function is called on line 40. Note that the function accepts an argument that is a **`const`** reference to an object of class `PlayerID`. Two member functions of this object are called in the output statements on line 32 and 33; first the function `showStrategy()`, and then the function `getAvgPenalty()`, which returns the value of the `avgPenalty` variable. Executing the program yields the following output:

```
strategy     : 0 -> C, DD -> D, DC -> C, CD -> D, CC -> C
avg. penalty : 0
```

Since the argument `obj` is treated by the compiler as **`const`** object within the body of the function `report`, it is obligatory that `showStrategy()` on line 32 and `getAvgPenalty()` on line 33 do not alter the information contained in the object `obj`. The compiler knows that this requirement is met, because both functions are defined as **`const`** methods on line 10 and 11 of the class definition. This guarantees that they merely rely on read access to the class. As you can see, a function is declared as a **`const`** method by adding the keyword **`const`** after the list of arguments in the function declaration. For functions that have a definition separate from their declaration, it is necessary to repeat the keyword **`const`** on the first line of the function definition, as you can see for the definition of `showStrategy` on line 20.

***ET* on `mutable`**

C^{++} style guidelines encourage the use of **`const`** methods as a tool to enforce **`const`**-correctness. In fact, a special loop-hole is provided in the language that will allow you to declare a member function as **`const`** even if it does change the data stored in an object. In order for this exception to be recognised, the data members that are affected by the method must be declared as **`mutable`** variables. For example, if `MyClass::f()` is a function that changes the value of an integer `i` that belongs to `MyClass`, then `MyClass::f()` can be made a **`const`** method if `MyClass::i` is declared as **`mutable int`** `i` in the class definition. In practice, this special loop-hole is exploited in special cases such as for auxiliary variables like counters, output file streams or derived data that do not reflect the logical state of the object.

`friend` functions

Sometimes, the method of controlling access to individual data members by `get()` and `set()` functions is too clumsy or inefficient, so that we would like to have a more direct way of accessing the class in a specific part of the code. Rather than making member variables**`public`**, the preferred solution in such cases is to arrange for privileged access for particular outside functions by declaring them as **`friend`**s of the class.

Code listing 12.4: A `private` method

```
class PlayerIPD {
public:
    ...
    PlayerIPD(Action c) : coopInit(c), coopResp({c, c, c, c}) { initialise(); }
    PlayerIPD(Action ci, const Response& cr) : coopInit(ci), coopResp(cr) { initialise();}
    ...
private:
    void initialise() { frequency = avgPenalty = 0.0; }
    ...
    double frequency, avgPenalty;
};
```

Code listing 12.5: `const` member functions

```
#include <array>
#include <iostream>

class PlayerIPD {
public:
    typedef bool Action;
    typedef std::array<Action, 4> Response;
    PlayerIPD(Action c) : coopInit(c), coopResp({c, c, c, c}) { initialise(); }
    PlayerIPD(Action ci, const Response& cr) : coopInit(ci), coopResp(cr) { initialise(); }
    void showStrategy() const;
    double getAvgPenalty() const { return avgPenalty; }
    double& frq() { return frequency; }
private:
    void initialise() { frequency = avgPenalty = 0.0; }
    const Action coopInit;
    const Response coopResp;
    double frequency, avgPenalty;
};

void PlayerIPD::showStrategy() const
{
    std::cout << "0 -> " << (coopInit ? 'C' : 'D');
    for(int i = 0; i < coopResp.size(); ++i)
        std::cout << ", " << (i / 2 ? 'C' : 'D') << (i % 2 ? 'C' : 'D')
        << " -> " << (coopResp[i] ? 'C' : 'D');
    std::cout << '\n';
}

void report(const PlayerIPD &obj)
{
    std::cout << "strategy     : ";
    obj.showStrategy();
    std::cout << "avg. penalty : " << obj.getPenalty() << '\n';
}

int main()
{
    PlayerIPD titForTat(true, {false, true, false, true});

    report(titForTat);

    return 0;
}
```

Table 12.1: Access rights[1] to class data members from within and outside of the class

		class data member	
		`public`	`private`
Outside functions	default	RW[2]	-
	`friend`	RW[2]	RW[2]
Member functions	default	RW[2]	RW[2]
	`const`	R[3]	R[3]

[1] None (-); read access (R); read and write access (RW).
[2] Write access can be prohibited by declaring the data member as a **`const`** variable.
[3] By exception, **`const`** member functions have write access to data members that are declared as a **`mutable`** variable.

Friend functions are not part of the class, yet they do have the same (unconstrained) access rights as member functions. To indicate that a function is a friend, simply list the function prototype in the class definition, preceded by the keyword **`friend`**. The following (simplified) program code illustrates how this can be done for a function `interact` that simulates an iterated prisoner's dilemma between two strategies and updates the average penalties after the interaction.

The **`friend`** function is called three times in the body of `main()` to simulate the pairwise interactions between three strategies: tit-for-tat, always-cooperate and always-defect. The output of the simulation indicates that tit-for-tat does well against an unconditional cooperator and, at the same time, avoids being exploited by an unconditional defector:

```
TFT against allC:
player 1 mean penalty = -0.1
player 2 mean penalty = -0.1
TFT against allD:
player 1 mean penalty = -0.501
player 2 mean penalty = -0.495
allC against allD:
player 1 mean penalty = -0.6
player 2 mean penalty = 0
```

Code listing 12.6: A `friend` function

```
class PlayerIPD {
public:
    typedef bool Action;
    typedef std::array<Action, 4> Response;
    PlayerIPD(Action c) : coopInit(c), coopResp({c, c, c, c}) { initialise(); }
    PlayerIPD(Action ci, const Response& cr) : coopInit(ci), coopResp(cr) { initialise();}
    ...
    friend void interact(PlayerIPD&, PlayerIPD&);
private:
    ...
    const Action coopInit;
    const Response coopResp;
    double frequency, avgPenalty;
    static const int nRounds;
    static const double penaltyMajorCrime, penaltyMinorCrime;
};

const int PlayerIPD::nRounds = 100;
const double PlayerIPD::penaltyMajorCrime = 0.5;
const double PlayerIPD::penaltyMinorCrime = 0.1;

void interact(PlayerIPD &player1, PlayerIPD &player2)
{
    PlayerIPD::Action x1 = player1.coopInit;
    PlayerIPD::Action x2 = player2.coopInit;

    const std::array<double, 4> penalty = {
```

```
28          PlayerIPD::penaltyMajorCrime,
29          0.0,
30          PlayerIPD::penaltyMajorCrime + PlayerIPD::penaltyMinorCrime,
31          PlayerIPD::penaltyMinorCrime};
32
33      double sum1 = 0.0, sum2 = 0.0;
34      for(int t = 0; t < PlayerIPD::nRounds; ++t) {
35
36          // determine outcome of current round
37          const int j1 = 2 * static_cast<int>(x1) + static_cast<int>(x2);
38          const int j2 = 2 * static_cast<int>(x2) + static_cast<int>(x1);
39
40          // accumulate penalties for both players
41          sum1 += penalty[j1];
42          sum2 += penalty[j2];
43
44          // determine action in next round
45          x1 = player1.coopResp[j1];
46          x2 = player2.coopResp[j2];
47      }
48      sum1 /= PlayerIPD::nRounds;
49      sum2 /= PlayerIPD::nRounds;
50
51      std::cout << "player 1 mean penalty = " << sum1 << '\n'
52                << "player 2 mean penalty = " << sum2 << '\n';
53
54      // update average penalty for the two strategies
55      player1.avgPenalty += sum1 * player2.frequency;
56      player2.avgPenalty += sum2 * player1.frequency;
57  }
58
59  int main()
60  {
61      PlayerIPD titForTat(true, {false, true, false, true});
62      PlayerIPD allC(true), allD(false);
63
64      std::cout << "TFT against allC:\n";
65      interact(titForTat, allC);
66
67      std::cout << "TFT against allD:\n";
68      interact(titForTat, allD);
69
70      std::cout << "allC against allD:\n";
71      interact(allC, allD);
72
73      return 0;
74  }
```

Now have a closer look at the following lines in the code listing:

line **22** The absence of `PlayerIPD::` in front of the function name indicates that the function `interact` is not a member of the class `PlayerIPD`. In fact, nothing in the function prototype reveals that `interact()` is a **`friend`** function.

line **8** This is the point were the privileged access of the function `interact` to the class `PlayerID` is made explicit.

line **24** Here you can see an example of where the **`friend`** function relies on direct access to one of the **`private`** variables (`coopInit` in this case). See also lines 45, 46, 55 and 56.

line **28** A friend function also has access to **`private static`** variables. However, since the **`friend`** function is defined outside of the scope of the class, it is necessary to access these variables through the scope operator `PlayerIPD::`. The same applies to type names that were defined within the scope of the class, as you can see for `PlayerIPD::Action` on lines 24 and 25.

Friend functions provide an easy way to gain access to **`private`** class members but their extensive use is discouraged, because they rely on an exception to the default access restrictions. In fact, in the

example above, there is a more elegant solution to the programming problem that requires only minor changes. The alternative avoids the use of a **friend** function by implementing `interact` as a class method. The following line-by-line code comparison shows the two implementations next to each other (most code sections that are identical in both implementations have been left out).

```
class PlayerIPD {
public:
    ...
    friend void interact(PlayerIPD&,
                         PlayerIPD&);
private:
    ...
};
...
void interact(PlayerIPD &player1, PlayerIPD
    &player2)
{
    PlayerIPD::Action x1 = player1.coopInit;
    PlayerIPD::Action x2 = player2.coopInit;

    const std::array<double, 4> penalty =
      { PlayerIPD::penaltyMajorCrime,
        0.0,
        PlayerIPD::penaltyMajorCrime +
    PlayerIPD::penaltyMinorCrime,
        PlayerIPD::penaltyMinorCrime};

    double sum1 = 0.0, sum2 = 0.0;
    for(int t = 0; t < PlayerIPD::nRounds; ++t)
    {
        // determine outcome of current round
        ...
        // accumulate penalties for both players
        ...

        // determine action in next round
        x1 = player1.coopResp[j1];
        x2 = player2.coopResp[j2];
    }
    sum1 /= PlayerIPD::nRounds;
    sum2 /= PlayerIPD::nRounds;
    ...

    // update average penalty
  player1.avgPenalty += sum1 *
    player2.frequency;
  player2.avgPenalty += sum2 *
    player1.frequency;
}

int main()
{
    ...
    std::cout << "TFT against allC:\n";
    interact(titForTat, allC);

    std::cout << "TFT against allD:\n";
    interact(titForTat, allD);

    std::cout << "allC against allD:\n";
    interact(allC, allD);

    return 0;
}
```

```
class PlayerIPD {
public:
    ...
    void interact(PlayerIPD&);

private:
    ...
};
...
void PlayerIPD::interact(PlayerIPD
    &partner)
{
    Action x1 = coopInit;
    Action x2 = partner.coopInit;

    const std::array<double, 4> penalty =
      { penaltyMajorCrime,
        0.0,
        penaltyMajorCrime +
    penaltyMinorCrime,
        penaltyMinorCrime};

    double sum1 = 0.0, sum2 = 0.0;
    for(int t = 0; t < nRounds; ++t)
    {
        // determine outcome of current round
       ...
        // accumulate penalties for both players
        ...

        // determine action in next round
        x1 = coopResp[j1];
        x2 = partner.coopResp[j2];
    }
    sum1 /= nRounds;
    sum2 /= nRounds;
    ...

    // update average penalty
    avgPenalty += sum1 *
    partner.frequency;
    partner.avgPenalty += sum2 *
    frequency;
}

int main()
{
    ...
    std::cout << "TFT against allC:\n";
    titForTat.interact(allC);

    std::cout << "TFT against allD:\n";
    titForTat.interact(allD);

    std::cout << "allC against allD:\n";
    allC.interact(allD);

    return 0;
}
```

The key difference between the two implementations is that the original solution with the **friend** function (left) treats the two players symmetrically, whereas the member function implementation is written from the perspective of a focal player (the class instance for which the member function is called) who interacts with its partner (which is passed as an argument to the function). This asymmetry is apparent on lines 12-13, 30-31 and 38-39, but very clearly visible in the way `interact()` is called on lines 46, 49 and 52. Another difference is that the version of `interact()` on the right belongs to the class `PlayerIPD` (as reflected by the function prototype on line 10), so it is not necessary anymore to access the static class members and data types via a scope operator `PlayerIPD::`.

Summary

This and the preceding section introduced different tools for customising access to class data members. To help you keep an overview, Table 12.1 shows the different options side-by-side. The key points to remember are:

- Functions that do not belong to a class can only access public data members. To create an exception to this rule, you can designate the outside function as a **friend** of the class, which gives it the same access rights as a member function.
- Member functions, regardless of whether they are **public** of **private**, have unrestricted access to data members of their class. To prohibit a member function from modifying the class data, you can declare the member function as a **const** method. In this way, it will have read-access only.
- There are various ways to tailor the access rights to individual variables:
  1. If unrestricted access is desirable (which usually is a bad idea), you can opt to include a variable in the **public** instead of the **private** section of the class definition.
  2. If the variable does not need to change after initialisation, you can protect it from write access by declaring the variable as **const**.
  3. In exceptional cases, a variable can be declared as **mutable**. This will open up the possibility for **const** methods to modify the value of the variable, even though they normally have read access only. The use of this loophole is intended for auxiliary variables that do not reflect the fundamental state of the class object.

12.5 Overloading operators

In Module 9, *Working with text*, you were introduced to objects of the class `std::string` and learned to work with them to handle words, sentences and other text or text-like sequences. You may recall that there are a number of operators that enable the efficient manipulation of `std::string` objects (they are listed in Table 9.2). For example, if we have two string objects, `std::string str1 = "hello "` and `std::string str2 = "world"`, then we can simply concatenate them by 'adding them together', as in:

```
std::cout << str1 + str2 << 'n';
```

Perhaps you realise that, in this expression, the + operator has an effect that is specific and meaningful for `std::string` objects (concatenation), but different from its usual effect on numerical values (addition).

How does the compiler know what the behaviour of the + operator should be for `std::string` objects? And why, in fact, does it interpret the + operator in a meaningful way, rather than doing something stupid that you would perhaps expect of a computer (such as adding up the character values of the letters in the two strings). The reason is that the `string` library has overloaded (i.e., (re)defined the meaning of) the + operator by including a special function with the name **operator**+ that implements the concatenation of two `std::string` objects. The code below gives you an idea of what this function looks like (the actual implementation in the `string` library is a bit more complicated to make it more flexible and efficient):

```
1 std::string operator+(const std::string &lhs, const std::string &rhs)
2 {
3     std::string result(lhs);     // copy left operand into result string
4     result.append(rhs);          // add right operand at the end
5     return result;               // return concatenated strings
6 }
```

As suggested by this example, it is relatively simple to overload operators for one of your home-made classes. All you need to do is define a function that defines the behaviour of the operator on your class

objects. Let us see how this works for the class `PlayerIPD`, if we decide to replace the function `interact` by an operator function **`operator*`**. That way, we will be able to replace the expression

```
interact(titForTat, allC);
```

on line 65 of code listing 12.6 by the shorthand notation:

```
titForTat * allC;
```
(see line 50 in the listing below)

As before, there are two ways of accomplishing this task: one solution relies on a **`friend`** operator function, the other implements the operator as a class member function. The first approach is outlined in the listing below. The only modification that is required, relative to code listing 12.6, is that the function name `interact` is replaced by **`operator*`**:

```
1  class PlayerIPD {
2  public:
3      ...
4      friend void operator*(PlayerIPD&, PlayerIPD&);
5  private:
6      ...
7  };
8  ...
9  void operator*(PlayerIPD &player1, PlayerIPD &player2)
10 {
11     PlayerIPD::Action x1 = player1.coopInit;
12     PlayerIPD::Action x2 = player2.coopInit;
13
14     const std::array<double, 4> penalty = {
15         PlayerIPD::penaltyMajorCrime,
16         0.0,
17         PlayerIPD::penaltyMajorCrime + PlayerIPD::penaltyMinorCrime,
18         PlayerIPD::penaltyMinorCrime};
19
20     double sum1 = 0.0, sum2 = 0.0;
21     for(int round = 0; round < PlayerIPD::nRounds; ++round) {
22
23         // determine outcome of current round
24         const int j1 = 2 * static_cast<int>(x1) + static_cast<int>(x2);
25         const int j2 = 2 * static_cast<int>(x2) + static_cast<int>(x1);
26
27         // accumulate penalties for both players
28         sum1 += penalty[j1];
29         sum2 += penalty[j2];
30
31         // determine action in next round
32         x1 = player1.coopResp[j1];
33         x2 = player2.coopResp[j2];
34     }
35     sum1 /= PlayerIPD::nRounds;
36     sum2 /= PlayerIPD::nRounds;
37
38     std::cout << "player 1 mean penalty = " << sum1 << '\n'
39               << "player 2 mean penalty = " << sum2 << '\n';
40
41     // update average penalty for the two strategies
42     player1.avgPenalty += sum1 * player2.frequency;
43     player2.avgPenalty += sum2 * player1.frequency;
44 }
45
46 int main()
47 {
48     ...
49     std::cout << "TFT against allC:\n";
50     titForTat * allC;  // this is where the operator function is called
51     ...
52     return 0;
53 }
```

Code listing 12.7, at the end of this module, illustrates the alternative solution, in which **operator*** is included as a member function of the class. In this case, the function prototype of **operator*** is given by

```
void PlayerIPD::operator*(PlayerIPD &partner)
```

and its function body (see line 59-94 in listing 12.7) is identical to that of the previous member-function implementation of `interact`. For this implementation, it is perhaps a bit less obvious to see the connection between the definition of the operator function and its use in the expression `titForTat * allC`. For example, which of the two strategies will now take the role of the `partner` in the function body of **operator***, and which one will be the focal strategy? To answer this question it is helpful to know that the compiler will interpret operator expressions in the following way:

`titForTat * allC` is equivalent to: `titForTat.operator*(allC)`

Accordingly, the operand on the left-hand side of the multiplication sign will take the role of the focal individual; the one on the right-hand side will be the partner.

***Master Yoda* on operator overloading**

You can overload many of the C++ operators, but - for the sake of clarity - it is best to reserve operator overloading to cases where the effect of the operator on your class objects bears some resemblance to the operator's original use. There are a few formal restrictions as well: you cannot change the precedence of an operator or the number of operands that it requires. For this reason, operator functions typically also cannot have default arguments. ET will later discuss a few issues related to overloading the assignment operator (or any of the compound assignment operators); make sure to have a look at Module **??**, **??**, before you attempt to overload an assignment operator.

Overloading the output operator

Many of the standard C++ operators can be overloaded for your home-made classes, including the standard arithmetic, (compound) assignment, logical and relational operators (but see Yoda's comment above). Overloading the output operator << is possible as well and will allow you to pass home-made class objects directly to a `std::ostream` object. To illustrate how this works for our example class `PlayerIPD`, let us consider how the original implementation (shown on the left, on page 142) with the output function `showStrategy()` can be converted into an alternative implementation (shown on the same page, on the right) with an overloaded output operator.

First, have a brief look at the statements on line 39 of both listings. On the left, you see how member function `showStrategy()` is called for the class instance `titForTat`. The equivalent statement on the right has the format of an ordinary output statement, in which the object `titForTat` is sent towards the standard output stream `std::cout` in the same way as you would print the value of any variable to the screen. Both statements produce exactly the same output:

```
0 -> C, DD -> D, DC -> C, CD -> D, CC -> C
```

Next, consider the class definitions on the left and on the right. You will see that output operator has been listed as a **friend** function of the class `PlayerIPD`. In this case, defining the operator as a member function is not an option, since we do not have access to the class definition of its left-hand side operand (the `std::ostream` object). The second operand (the one on the right-hand side of the output operator) is an instance of our class `PlayerIPD`. This operand is passed by **const** reference, consistent with the fact that the output operation does not change the contents of the `PlayerIPD` object (you may recall that this was why `PlayerIPD::showStrategy()` was defined as a **const** method earlier).

Note, finally, that the implementations of `showStrategy()` and **operator**<< on lines 20-33 are almost identical, except for a few subtle differences:

1. `showStrategy()` writes its output specifically to the standard output stream `std::cout`, whereas the definition of **operator**<< is more general. It will allow output to be sent to any arbitrary `std::ostream`. This could, for example, be one of the standard output streams `std::cout`, `std::clog` or `std::cout`, or a stream that is connected to an output file.

2. `showStrategy()` has immediate access to the private class data members `coopInit` and `coopResp`, because it is a class member function. By contrast, **operator**<< is not a class member, so `coopInit` and `coopResp` must always be accessed via the object that they are contained in (i.e., as `obj.coopInit` and `obj.coopResp`, respectively). This explains also why **operator**<< must be made a **friend** of class `PlayerIPD`. Without this status, the output operator would not be allowed to retrieve the values of the private data members.

3. `showStrategy()` is a **`void`** function, but the overloaded output operator returns the `std::ostream` to which it has been sending data (this is done by means of a reference return value to avoid that a copy is being made). Thanks to this clever trick, output operations can be concatenated, as you have been used to. For example, the expression `std::cout << titForTat << std::endl` on line 39 will initially be interpreted by the compiler as **`operator`**`<<(std::cout, titForTat) << std::endl`. After producing the first part of the output, the function **`operator`**`<<` returns its first operand, so that the expression will evaluate to `std::cout << std::endl`. From that point, the program continues to write the second part of the output sequence to the screen.

```
1  #include <array>
2  #include <iostream>
3
4  class PlayerIPD {
5  public:
6      typedef bool Action;
7      typedef std::array<Action, 4> Response;
8      ...
9      void showStrategy() const;
10
11
12     ...
13 private:
14     ...
15     const Action coopInit;
16     const Response coopResp;
17     ...
18 };
19
20 void PlayerIPD::showStrategy() const
21
22 {
23     std::cout << "0 -> "
24               << (coopInit ? 'C' : 'D');
25     const int n = coopResp.size();
26     for(int i = 0; i < n; ++i)
27         std::cout  << ", "
28             << (i / 2 ? 'C' : 'D')
29             << (i % 2 ? 'C' : 'D')
30             << " -> "
31             << (coopResp[i] ? 'C' : 'D');
32     return;
33 }
34
35 int main()
36 {
37     PlayerIPD titForTat(true, {false, true,
       false, true});
38
39     titForTat.showStrategy();
40     std::cout << std::endl;
41     return 0;
42 }
```

```
1  #include <array>
2  #include <iostream>
3
4  class PlayerIPD {
5  public:
6      typedef bool Action;
7      typedef std::array<Action, 4> Response;
8      ...
9      friend std::ostream&
10         operator<<(std::ostream&,
11                    const PlayerIPD&);
12     ...
13 private:
14     ...
15     const Action coopInit;
16     const Response coopResp;
17     ...
18 };
19
20 std::ostream& operator<<(std::ostream &os,
21     const PlayerIPD &obj)
22 {
23     os  << "0 -> "
24         << (obj.coopInit ? 'C' : 'D');
25     const int n = obj.coopResp.size();
26     for(int i = 0; i < n; ++i)
27         os  << ", "
28             << (i / 2 ? 'C' : 'D')
29             << (i % 2 ? 'C' : 'D')
30             << " -> "
31             << (obj.coopResp[i] ? 'C' : 'D');
32     return os;
33 }
34
35 int main()
36 {
37     PlayerIPD titForTat(true, {false, true,
       false, true});
38
39     std::cout << titForTat << std::endl;
40
41     return 0;
42 }
```

12.6 Simulating an iterated prisoner's dilemma tournament

As promised, we conclude this chapter by providing the completed program code for simulating an iterated prisoner's dilemma tournament. The following implementation structures the tournament in a discrete number of (100) rounds; in each round, the competing strategies interact in an iterated prisoner's dilemma (which, in itself, consists of 100 repeated pairwise interactions) and their frequencies are adjusted according to their average payoff values (see `simulateTournament()` on line 98-147 of code listing 12.7 below). In this way, the successful strategies become more abundant over time, whereas the unsuccessful ones decrease in frequency. To be precise, the frequency of strategy i in round t of the tournament, $f_i(t)$,

is calculated as

$$f_i(t) = f_i(t-1)\frac{1-\pi_i(t-1)}{1-\bar{\pi}(t-1)}$$

where, $1-\pi_i(t-1)$ measures the performance of strategy i in the previous round of the tournament (calculated as a baseline performance value of 1 minus the expected jail-time for the strategy, $\pi_i(t-1)$). To ensure that the sum of all frequencies $\sum f_i(t)$ remains equal to 1, the frequencies are normalised by a factor $1-\bar{\pi}(t-1)$. This term represents the average performance in round $t-1$ (weighted with respect to the frequencies of the different strategies).

The code below simulates a tournament between six strategies: an unconditional cooperator (allC), an unconditional defector (allD), two versions of the strategy tit-for-tat (one starts an interaction by playing C; the other starts by playing D), and the same two versions of a strategy known as win-stay, lose-shift (). In fact, the simulation of the tournament shows that the latter strategy dominates eventually. After 100 rounds in the tournament, the two versions of win-stay, lose-shift have reached frequencies of 57.3% and 40.8%, respectively, as demonstrated by the final last five lines of output from the simulation:

```
t = 95 3.90e-05 0.0254 1.08e-11 0.581 0.394 2.35e-07
t = 96 3.26e-05 0.0239 8.21e-12 0.579 0.397 2.07e-07
t = 97 2.72e-05 0.0224 6.21e-12 0.577 0.401 1.83e-07
t = 98 2.26e-05 0.0210 4.69e-12 0.575 0.404 1.62e-07
t = 99 1.87e-05 0.0197 3.55e-12 0.573 0.408 1.43e-07
```

The only other strategy that still remains at an appreciable frequency is strategy #2, which is labeled as 'nice tit-for-tat' in the code on line 109.

While preparing the final version of the code, it became clear that an additional method `resetAvgPenalty()` was needed to reset the value of `avgPenalty` to zero after each round of the tournament. This addition, in turn, eliminated the need to initialise `avgPenalty` in the constructors, allowing the class definitions to be simplified (by eliminating the **`private`** method `initialise()`). In practice, a programmer will often switch between working on the simulation code and extending or modifying the class implementation in an object-oriented-programming project: sometimes, s/he may discover that certain parts of the class definition can be simplified or streamlined; at other times, new functionality may need to be added. When working in this mode, it remains useful to think of the class definition and implementation as a piece of code that should be able to stand on its own. This will make it easier to reuse the class in a future application, saving a lot of development time.

Code listing 12.7: Completed simulation code for the iterated prisoner's dilemma tournament

```
1  #include <array>
2  #include <iostream>
3  #include <iomanip>
4
5  //*** definition of class PlayerIPD ****************************************************
6
7  class PlayerIPD {
8  public:
9      // type definitions
10     typedef bool Action;
11     typedef std::array<Action, 4> Response;
12
13     // constructors
14     PlayerIPD(Action c) : coopInit(c), coopResp({c, c, c, c}) {}
15     PlayerIPD(Action ci, const Response& cr) : coopInit(ci), coopResp(cr) {}
16
17     // other class methods
18     friend std::ostream& operator<<(std::ostream&, const PlayerIPD&);
19     double getAvgPenalty() const { return avgPenalty; }
20     void resetAvgPenalty() { avgPenalty = 0.0; }
21     double& frq() { return frequency; }
22     void operator*(PlayerIPD&);
23 private:
24     // strategic variables
25     const Action coopInit;          // action in first round of IPD
26     const Response coopResp;        // actions in subsequent rounds,
27                                     // conditional on the outcome of the previous round
28     // state variables
29     double frequency, avgPenalty;
```

```
30
31      // shared parameters
32      static const int nRounds;
33      static const double penaltyMajorCrime, penaltyMinorCrime;
34  };
35
36  //*** definitions of static class members *********************************************
37
38  const int       PlayerIPD::nRounds = 100;               // number of rounds in the IPD
39  const double    PlayerIPD::penaltyMajorCrime = 0.5;     // payoff parameter P
40  const double    PlayerIPD::penaltyMinorCrime = 0.1;     // payoff parameter p
41
42  //*** implementation of PlayerIPD methods *************************************************
43
44  std::ostream& operator<<(std::ostream &os, const PlayerIPD &obj)
45  // overloads the output operator « for PlayerIPD objects
46  {
47      os  << "0 -> "
48          << (obj.coopInit ? 'C' : 'D');
49      for(int i = 0; i < obj.coopResp.size(); ++i)
50          os  << ", "
51              << (i / 2 ? 'C' : 'D')
52              << (i % 2 ? 'C' : 'D')
53              << " -> "
54              << (obj.coopResp[i] ? 'C' : 'D');
55      std::cout << '\n';
56      return os;
57  }
58
59  void PlayerIPD::operator*(PlayerIPD &partner)
60  // simulates the interaction between the focal strategy and a partner,
61  // and updates the accumulates the expected penalty (jail-time), weighted
62  // with respect to the frequency of the interaction
63  {
64      Action x1 = coopInit;
65      Action x2 = partner.coopInit;
66
67      const std::array<double, 4> penalty = {
68          penaltyMajorCrime,
69          0.0,
70          penaltyMajorCrime + penaltyMinorCrime,
71          penaltyMinorCrime};
72
73      double sum1 = 0.0, sum2 = 0.0;
74      for(int round = 0; round < nRounds; ++round) {
75
76          // determine outcome of current round
77          const int j1 = 2 * static_cast<int>(x1) + static_cast<int>(x2);
78          const int j2 = 2 * static_cast<int>(x2) + static_cast<int>(x1);
79
80          // accumulate penalties for both players
81          sum1 += penalty[j1];
82          sum2 += penalty[j2];
83
84          // determine action in next round
85          x1 = coopResp[j1];
86          x2 = partner.coopResp[j2];
87      }
88      sum1 /= nRounds;
89      sum2 /= nRounds;
90
91      // update average penalty for the two strategies
92      avgPenalty += sum1 * partner.frequency;
93      partner.avgPenalty += sum2 * frequency;
94  }
95
```

```cpp
//*** implementation of the iterated prisoner's dilemma tournament **********************

void simulateTournament()
// simulates a tournament between a predefined set of strategies that
// interact in an iterated prisoner's dilemma
{
    // define simulation parameters
    const int nStrategies = 6;          // number of competing strategies
    const int tEnd = 100;               // number of rounds in the tournament

    // initialise strategies
    std::array<PlayerIPD, nStrategies> strategies {
        PlayerIPD( true),                                   // allC
        PlayerIPD( true, {false, true, false, true}),   // nice tit-for-tat
        PlayerIPD(false, {false, true, false, true}),   // nasty tit-for-tat
        PlayerIPD( true, {true, false, false, true}),   // nice win-stay, lose-shift
        PlayerIPD(false, {true, false, false, true}),   // nasty win-stay, lose-shift
        PlayerIPD(false)                                    // allD
    };

    // set initial frequencies
    for(int i = 0; i < nStrategies; ++i)
        strategies[i].frq() = 1.0 / nStrategies;

    for(int t = 0; t < tEnd; ++t) {

        // reset penalty values
        for(int i = 0; i < nStrategies; ++i)
            strategies[i].resetAvgPenalty();

        // simulate pairwise interactions and determine frequencies after selection
        double sum = 0.0;
        std::array<double, nStrategies> tmpFreq;
        for(int i = 0; i < nStrategies; ++i) {
            for(int j = i; j < nStrategies; ++j)
                strategies[i] * strategies[j];

            sum += tmpFreq[i] =
                strategies[i].frq() * (1.0 - strategies[i].getAvgPenalty());
        }

        // normalise frequencies and update objects
        for(int i = 0; i < nStrategies; ++i)
            strategies[i].frq() = tmpFreq[i] / sum;

        // produce output
        std::cout << "t = " << t;
        for(int i = 0; i < nStrategies; ++i)
            std::cout << ' ' << std::showpoint << std::setprecision(3) << strategies[i].frq();
        std::cout << '\n';
    }
}

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

12.7 Exercises

12.1. The sequential assessment game. Many animals display aggressive signals (threat displays) before they engage in a costly escalated fight with an opponent. Often, these displays occur in a graded sequence.

When the initial display does not settle the dispute, the opponents move on to another level of display that is more challenging and costly to produce. By sequentially assessing each other's strength, animals may gain important information about their ability to win an escalated fight against their opponent. If the chance of winning the fight is perceived to be small, the best decision may be to give up, so as to avoid the cost of injury in an escalated fight.

The following code implements a simple version of a sequential assessment game, in which two opponents engage in repeated ritualised fighting behaviours. In each round of the game, the two opponents can decide to continue the interaction or to give up. If both continue, the two individuals will engage in a ritualised fight, which will be won by one of the two opponents. Which one will win, depends on their relative fighting ability (or `rhp`, i.e., resource holding potential).

The two contestants are assumed to have no information initially about their relative fighting ability. Yet, if there is an asymmetry in resource holding potential, this would likely be reflected in the outcome of the ritualised fights. Therefore, the rational behaviour for both opponents is to base their decision to give up or continue on their performance during the past sequence of ritualised fights. In particular, a focal individual, who won k times out of a total of n rounds in a sequential assessment game, can expect to win future rounds with probability $p = k/n$. It makes sense to give up whenever $p < \frac{1}{2}$, because the opponent (who has exactly the same information) will conclude that he has a probability of winning that exceeds 50%.

An issue to consider is that an individual's estimate of p is very unreliable if the number of rounds is small; even a relatively strong individual can be unlucky and lose a number of rounds just by chance. Therefore, we will assume that individuals are initially biased towards believing that they have a fair chance of winning, and that they give up when they are *almost sure* that they are weaker than their opponent. For example, a simple rule that they might use, is to give up whenever

$$p + 2\,\sigma_p < \tfrac{1}{2}$$

Here, $\sigma_p = \sqrt{p\,(1-p)}/\sqrt{n}$ is the standard error of the estimated probability of winning. Moreover, to reflect the initial belief bias, we assume that individuals start their sequential assessment with $n = 4$ and $k = 2$, as if they had already interacted four times with their opponent and won half of the time.

Take a detailed look at the code below, and then attempt to answer the following questions

(a) What are the member variables of the class `Individual`? List all of them and explain what they are used for. Do the same for the member functions.

(b) Which of the class member functions affect the values of the variables `k` and `n`? Explain how these variables change over time.

(c) How precisely does the outcome of the ritualised fight depend on the asymmetry in resource holding potential between the two opponents? In particular, give a mathematical interpretation of the statements on line 32-33. (see code listing **??** and the accompanying text on p. **??** for information about the function `rand()`).

(d) What are the different possible outcomes of a round of the sequential assessment game in the program and what do they represent biologically?

Code listing 12.8: Implementation of a simple sequential assessment game

```
1  #include <cstdlib>
2  #include <cmath>
3  #include <iostream>
4
5  class Individual {
6  public:
7      Individual(double RHP) { rhp = RHP; k = 2; n = 4; }
8      bool giveUp();
9      int fight1 (Individual&);
10 private:
11     int k, n;
12     double rhp;
13 };
14
15 bool Individual::giveUp() {
16     if(n == 0)
```

```
17          return false;
18      else {
19          double p = static_cast<double> (k) / n;
20          return p + 2.0 * sqrt(p * (1.0 - p) / n) < 0.5;
21      }
22  }
23
24  int Individual::fight1(Individual &opponent) {
25      if(giveUp()) return opponent.giveUp() ? 0 : -1;
26      else {
27          if(opponent.giveUp()) return +1;
28          else {
29              ++n;
30              ++opponent.n;
31              if(std::rand() < RAND_MAX * rhp / (rhp + opponent.rhp)) ++k;
32              else ++opponent.k;
33              return 0;
34          }
35      }
36  }
37
38  int main()
39  {
40      std::srand(5); // seed random number generator
41
42      Individual A(1.0), B(0.95);
43
44      int outcome;
45      do {
46          outcome = A.fight1(B);
47          std::cout << outcome << '\n';
48      } while(outcome == 0);
49
50      return 0;
51  }
```

Once you are confident that you understand the code, implement the following adjustments

(e) Replace the `inline` definition of the constructor by a function declaration plus a definition of the constructor outside of the class definition.

(f) Turn the member variable `rhp` into a `const double`, and adjust the constructor accordingly.

(g) Turn as many member functions into `const` methods as you can (without any other changes to the class definition).

(h) Write a function `int fight2(Individual&, Individual&)` that has an effect equivalent `int Individual::fight1(Ind` but that does not belong to the class `Individual`. Adjust the statement on line 47 to test if your function works as it should.

(i) Replace the function `Individual::fight1()` by an equivalent operator function, such that you can replace the statement on line 47 by `outcome = A * B;`

12.2. The sequential assessment game with imperfect information ($\star$). Implement a variant of the sequential assessment game, where individuals, instead of being able to remember the outcome of all previous rounds, can only remember the outcomes of their twenty-five last interactions. In addition, assume that individuals cannot individually recognise their opponent, i.e., they merely keep track of the number of interactions they won across all of their previous interactions with members of their group, irrespective of the identity of their interaction partners. Starting from the implementation in code listing 12.8, first adjust the class implementation to incorporate the constraint on the number of interactions that can be remembered by an `Individual`. Then, implement 1000 rounds of repeated interactions between group members in a group consisting of five individuals. In each round, simulate all possible pairwise interactions within the group and keep track, for each pair, how often one individual dominates the other. Introduce small difference in resource holding potential between the group members (e.g., in the range 0.8-1.0),

and investigate how these are reflected in the observed dominance relationships within the group. [**Hint:** consider using a `std::bitset` to store the outcomes of the last 25 interactions.]

12.3. Complex number arithmetic - continued. In exercise 11.4, you implemented several functions to perform basic operations on complex numbers, represented by a **`struct`** `ComplexNumber`. Return to your original solution to this problem and define operator functions to replace your functions for computing the conjugate, the product and the ratio. Also add an output operator. Note: taking the conjugate is a unary operation, so you need to overload an operator that takes only a single argument (such as `tilde`). Make two different versions of the program, one that relies on operator functions that are external to the **`struct`** `ComplexNumber` and another where the data members of `ComplexNumber` are **`private`** and the operators are member functions (except for the output operator). For this second version, you will also need to define a constructor `ComplexNumber(`**`double, double`**`)`.

Test your skill

12.4. Implementing mathematical set objects. A collection of distinct objects can be represented mathematically by a set. For example, the collection $\{1, 3, 5\}$ is a set that contains three integer numbers. All of the elements in a mathematical set are unique, and there is no particular meaning to the order in which the elements of a set are listed (therefore, they are often shown in ascending order). Mathematical sets can be manipulated by a restricted set of operations, which all rely on the concept of set membership. Set membership is a binary relationship; for each object x and set A, we can say either that x is an element of A or that it is not. The former case is denoted as $x \in A$, the latter as $x \notin A$. Based on this definition, we can construct the following relationships between sets:

1. A set A is a subset of another set B, denoted as $A \subseteq B$, if all of the objects in A are elements of B.
2. The union of two sets A and B, denoted as $A \cup B$, is a set that contains all objects that are elements of either A, B or both.
3. The intersection of two sets A and B, denoted as $A \cap B$, is a set that contains all objects that are elements of both A and B.
4. The set difference of two sets A and B, denoted as $A \setminus B$, is a set that contains all objects that are elements of A but not of B.

Design a C^{++} class to implement mathematical set objects. Write a program that relies on your class to perform elementary calculations for sets that contain integer objects. For example, use the program to illustrate that, for arbitrarily chosen sets A and B,

$$(A \cup B) \setminus (A \cap B) = (A \setminus B) \cup (B \setminus A)$$

Note that you can check whether two sets U and V are equal by verifying that $U \subset V$ and $V \subset U$. Apart from implementing the basic set operations, make sure to also equip your class with an output operator, so that you can easily verify the result of the different operations. If you need an extra challenge, set up the class in such a way that set elements are maintained internally in ascending order. [**Optional:** if you prefer not to write this class entirely from scratch, you will surely find some helpful functions in the STL library.]

12.8 Solutions to the exercises

Exercise 12.1. The sequential assessment game.

(a) The class contains three data members, one constructor and two additional member functions:

member variables	
`n`	keeps track of the total number of rounds
`k`	counts the number of rounds that were won
`rhp`	reflects the individual's resource holding potential or fighting ability
member functions	
`Individual(double)`	constructor that initialises instances of `Individual`
`bool giveUp()`	implements the decision rule (continue fighting or give up)
`int fight(Individual&)`	implements a single round of the sequential assessment game

(b) The values of the variables `k` and `n` are initialised in the constructor, and updated depended on the outcome of each round in the function `int fight(Individual&)`. The variable `n` increases for both players in each round, so it keeps track of the number of sequential assessment rounds. The variable `k` is incremented only for the opponent that wins the ritualised fight, so it keeps track of the number of rounds that were won.

(c) If one player has RHP value x, and the other has RHP value y, then the first player wins the interaction with probability $x/(x+y)$.

(d) The possible outcomes are 0 (the fight is undecided); +1 (the focal individual dominates the opponent) or -1 (the focal individual submits to the opponent).

(e) See line 8 and 16-19 of the code below.

(f) See line 13 and 16 of the code below.

(g) Member function `giveUp()` can be declared as a **`const`** method. Member function `giveUp()` cannot be made **`const`** (except when `k` and `n` are made **`mutable`**), because this function alters the values of class data members `k` and `n`.

(h) See lines 7 and 44-58 and 69 of the code below.

(i) See lines 10, 30 and 68 of the code below.

```
1  #include <cstdlib>
2  #include <cmath>
3  #include <iostream>
4
5  class Individual {
6  public:
7      friend int fight2(Individual&, Individual&);
8      Individual(double);
9      bool giveUp() const;
10     int operator* (Individual&);
11 private:
12     int k, n;
13     const double rhp;
14 };
15
16 Individual::Individual(double RHP) : rhp (RHP)
17 {
18     k = 2; n = 4;
19 }
20
21 bool Individual::giveUp() const {
22     if(n == 0)
23         return false;
24     else {
```

```
25          double p = static_cast<double> (k) / n;
26          return p + 2.0 * sqrt(p * (1.0 - p) / n) < 0.5;
27      }
28  }
29
30  int Individual::operator*(Individual &opponent) {
31      if(giveUp()) return opponent.giveUp() ? 0 : -1;
32      else {
33          if(opponent.giveUp()) return +1;
34          else {
35              ++n;
36              ++opponent.n;
37              if(std::rand() < RAND_MAX * rhp / (rhp + opponent.rhp)) ++k;
38              else ++opponent.k;
39              return 0;
40          }
41      }
42  }
43
44  int fight2(Individual &player1, Individual &player2) {
45      if(player1.giveUp()) return player2.giveUp() ? 0 : -1;
46      else {
47          if(player2.giveUp()) return +1;
48          else {
49              ++player1.n;
50              ++player2.n;
51              if(std::rand() < RAND_MAX * player1.rhp / (player1.rhp + player2.rhp))
52                  ++player1.k;
53              else
54                  ++player2.k;
55              return 0;
56          }
57      }
58  }
59
60  int main()
61  {
62      std::srand(5); // seed random number generator
63
64      Individual A(1.0), B(0.95);
65
66      int outcome;
67      do {
68          outcome = A * B;         // calls the member operator function
69          outcome = fight2(A, B); // calls the friend function
70          std::cout << outcome << '\n';
71      } while(outcome == 0);
72
73      return 0;
74  }
```

Exercise 12.2. The sequential assessment game with imperfect information.

```
1  #include <cstdlib>
2  #include <cmath>
3  #include <iostream>
4  #include <vector>
5  #include <bitset>
6
7  const int N = 25;                // memory size
8  const int nRounds = 1000;        // number of interaction rounds
9
10 class Individual {
```

```
public:
    Individual(double);
    bool giveUp();
    int fight1 (Individual&);
private:
    std::bitset<N> k, n;
    const double rhp;
};

Individual::Individual(double RHP) : rhp (RHP)
{
    // set initial observations (equivalent to n = 4, k = 2)
    n.reset();
    n[0] = n[1] = n[2] = n[3] = true;
    k.reset();
    k[0] = k[1] = true;
}

bool Individual::giveUp() {
    const int nn = n.count();
    if(nn == 0)
        return false;
    else {
        double p = static_cast<double> (k.count()) / nn;
        return p + 2.0 * sqrt(p * (1.0 - p) / nn) < 0.5;
    }
}

int Individual::fight1(Individual &opponent) {

    // erase memory of old interaction
    n <<= 1;     // the most significant bit, corresponding to the oldest memory,
    k <<= 1;     // is lost when the bitset is shifted to the left
    opponent.n <<= 1;
    opponent.k <<= 1;

    // implementation of ritualised fight
    if(giveUp()) return opponent.giveUp() ? 0 : -1;
    else {
        if(opponent.giveUp()) return +1;
        else {
            n.set(0);            // add memory of current interaction
            opponent.n.set(0);   // add memory of current interaction

            if(std::rand() < RAND_MAX * rhp / (rhp + opponent.rhp))
                k.set(0);
            else
                opponent.k.set(0);
            return 0;
        }
    }
}

int main()
{
    std::srand(5); // seed random number generator

    // create group of 5 individuals with slightly different RHPs
    std::vector<Individual> group
        {Individual(0.8), Individual(0.85), Individual(0.9), Individual(0.95), Individual(1.0)};

    // declare a matrix to keep track of the dominance relationships
    const int K = group.size();
    std::vector< std::vector<int> > dominance(K, std::vector<int>(K, 0));

    for(int t = 0; t < nRounds; ++t) {
```

```
77          // simulate all pairwise interactions within the group
78          for(int i = 0; i < K; ++i)
79              for(int j = 0; j < K; ++j) {
80                  if(i != j) { // individuals don't interact with themselves
81                      int outcome = group[i].fight1(group[j]);
82                      if(outcome == 1) ++dominance[i][j];          // i dominates j
83                      else if(outcome == -1) ++dominance[j][i];    // j dominates i
84                  }
85              }
86      }
87
88      // report results
89      std::cout << "observed dominance relationships:\n";
90      for(int i = 0; i < K; ++i)
91          for(int j = i + 1; j < K; ++j) {
92              double tmp = dominance[i][j] + dominance[j][i];
93              double q = tmp == 0.0 ? 0.5 : dominance[i][j] / tmp;
94              std::cout << "individual " << i << " dominates " << j << ' ' << 100 * q << "% of
    the time\n";
95          }
96
97      return 0;
98  }
```

Exercise 12.3. Complex number arithmetic - continued. Version 1 of the program (publicly accessible data and external operator functions):

```
1  #include <iostream>
2  #include <cmath>
3
4  struct ComplexNumber {
5      double mdRe, mdIm;
6  };
7
8  std::ostream& operator<<(std::ostream &os, const ComplexNumber &z)
9  {
10     os << z.mdRe << " + " << z.mdIm << " i";
11     return os;
12 }
13
14 double norm(const ComplexNumber &z)
15 {
16     return sqrt(z.mdRe * z.mdRe + z.mdIm * z.mdIm);
17 }
18
19 ComplexNumber operator~(const ComplexNumber &z)
20 {
21     return {z.mdRe, -z.mdIm};
22 }
23
24 ComplexNumber operator*(const ComplexNumber &y, const ComplexNumber &z)
25 {
26     return {y.mdRe * z.mdRe - y.mdIm * z.mdIm, y.mdRe * z.mdIm + y.mdIm * z.mdRe};
27 }
28
29 ComplexNumber operator/(const ComplexNumber &y, const ComplexNumber &z)
30 {
31     const double tmp = norm(z);
32     const double sqrNormZ = tmp * tmp;
33     ComplexNumber x = y * (~z);
34     return {x.mdRe / sqrNormZ, x.mdIm / sqrNormZ};
35 }
36
```

```
int main()
{
    ComplexNumber y  {5.0, 5.0}, z {3.0, -4.0};

    std::cout << y / z << '\n';

    return 0;
}
```

Version 2 of the program (private data and operator functions as class members):

```
#include <iostream>
#include <cmath>

class ComplexNumber {
public:
    friend std::ostream& operator<<(std::ostream&, const ComplexNumber&);
    ComplexNumber(double a, double b) : mdRe(a), mdIm(b) {} // constructor
    double norm() const;
    ComplexNumber operator~() const;
    ComplexNumber operator*(const ComplexNumber&) const;
    ComplexNumber operator/(const ComplexNumber&) const;
private:
    double mdRe, mdIm;
};

std::ostream& operator<<(std::ostream &os, const ComplexNumber &z)
{
    os << z.mdRe << " + " << z.mdIm << " i";
    return os;
}

double ComplexNumber::norm() const
{
    return sqrt(mdRe * mdRe + mdIm * mdIm);
}

ComplexNumber ComplexNumber::operator~() const
{
    return {mdRe, -mdIm};
}

ComplexNumber ComplexNumber::operator*(const ComplexNumber &z) const
{
    return {mdRe * z.mdRe - mdIm * z.mdIm, mdRe * z.mdIm + mdIm * z.mdRe};
}

ComplexNumber ComplexNumber::operator/(const ComplexNumber &z) const
{
    const double tmp = z.norm();
    const double sqrNormZ = tmp * tmp;
    ComplexNumber x = operator*(~z);
    return {x.mdRe / sqrNormZ, x.mdIm / sqrNormZ};
}

int main()
{
    ComplexNumber y {5.0, 5.0}, z {3.0, -4.0};

    std::cout << y / z << '\n';

    return 0;
}
```

Module 13

Procedural programming review

***Jamie Oliver* explains what's on the menu**

In this module you will find a summary of the elementary features of the C++ programming language that you will need as a basis for procedural programming. The module is intended as a reference manual that you can use to rehearse what you have learned so far, or to refresh your memory when you have not programmed for a while. Also, we recommend that you review this material before starting the advanced track *Object-oriented programming*. Various sections of this module were adapted from the excellent C++ tutorial on http://www.cplusplus.com/. The online tutorial contains much additional information and is highly recommended.

13.1 Variables and data types

Identifiers

Variables in a C++ program refer to a portion of the memory where the value of the variable is stored. Each variable needs a name that identifies it and distinguishes it from the others. In C++, a valid identifier is a sequence of one or more letters, digits, or underscore characters (_). Spaces, punctuation marks, and symbols are not allowed as part of an identifier. In addition, identifiers cannot begin with a digit, and typically begin with a letter. Identifiers that begin with an underline character (_), are often reserved for compiler-specific keywords or external identifiers, as well as identifiers containing two successive underscore characters anywhere. C++ is a case sensitive language.

C++ has a number of reserved keywords, such as `auto`, `char`, `double`, `extern`, `this`, `try`, `while` and several more, that cannot be used as identifiers by a programmer. Most of the time you can easily recognise these reserved keyword thanks to the automatic syntax highlighting available in most programming environments.

Fundamental data types

All variables are internally represented using zeros and ones, but these are interpreted in different ways according to the type of the variable. Moreover, different kinds of variables may not occupy the same amount of memory space; e.g., a simple integer is not stored in the same way as a letter or a large floating-point number. For this reason, the program needs to be aware of the kind of data stored in

the variable. The basic storage units supported natively by most systems are represented in C++ by the following fundamental data types

Character Single characters, such as 'A' or '!' are represented by the C++ data type `char`, which is a one-byte character.

Integer Numerical integer types can store a whole number value, such as 7 or 1024. The associated C++ data types exist in a variety of sizes, `short`, `int` or `long`, and can either be `signed` or `unsigned`, depending on whether they support negative values or not.

Floating-point The C++ data types `float`, `double` and `long double` can represent real values, such as 3.14 or 0.01, with different levels of precision, depending on which of the three floating-point types is used.

Logical The boolean type, known in C++ as `bool`, can only represent one of two states, `true` or `false`.

In addition to the fundamental data types listed above, C++ features a type `void` that is used to represent an empty value.

Variable declarations

C++ is a strongly-typed language, which means that every variable needs to be declared with its type before its first use. A declaration informs the compiler how much space needs to be reserved in memory for the variable and how to interpret its value. The syntax to declare a new variable in C++is straightforward: write the type followed by the variable name (i.e., its identifier), as in `double x;` Multiple variables can be declared in a single statement by separating their identifiers by a comma (e.g., `double x, y;`). Declaration statements are closed off with a semicolon.

Declarations can be modified by type modifiers, qualifiers and storage class specifiers, such as `signed`, `unsigned` and `short` (for integral types), `long` (for integers and `double`s) and `const`, which is used to prevent changes to a variable after its declaration.

Variable initialisation

Variables have an undetermined value until they are assigned a value for the first time. This means that you cannot count on a compiler to set the value of a variable to some specific default value when the variable is declared. It is possible to specificy an initial value for a variable at the moment it is declared. This is called the initialization of the variable. In C++, there are three ways to initialize variables. They are all equivalent and are reminiscent of the evolution of the language over the years:

1. The first one, known as C-like initialization (because it is inherited from the C language), consists of appending an equal sign followed by the value to which the variable is initialized:

   ```
   type identifier = initial_value;
   ```

   For example, to declare a variable of type `int` called `x` and initialize it to a value of zero from the same moment it is declared, we can write: `int x = 0;`

2. A second method, known as constructor initialization (introduced by the C++ language), encloses the initial value between parentheses (`()`):

   ```
   type identifier (initial_value);
   ```

   For example: `int x (0);`

3. Finally, a third method, known as uniform initialization, is similar to the above, but using curly braces (`{}`) instead of parentheses (this was introduced by the revision of the C++ standard, in 2011):

   ```
   type identifier {initial_value};
   ```

   For example: `int x {0};`

All three ways of initializing variables are valid and equivalent in C++.

Literals and constants

Initialisation statements frequently require the use of literals or constant expressions to specify the initial value of a variable. It is good practice to match the type of the literal to the type of the variable, in order to avoid unnecessary implicit type conversion. Literals and constants are specified as follows for the elementary data types.

`bool` The two possible for values for a boolean variable are denoted as **`true`** and **`false`**.

`int` Plain integer constants are written as whole numbers in decimal notation, e.g., `12` or `-1`; when you write a number followed by the letter `u` (e.g., `0u`) it will be interpreted as an **`unsigned`** number; writing `0x` in front of the number, as in `0x16A9F` allows you to specify integer constants in hexadecimal notation.

`char` Character constants are written between single quotes, e.g., `'a'` or `'\t'` for the tab-character.

`double` Double-precision floating point values are written with a decimal point, in normal or scientific notation, e.g., `3.14`, `1.` or `1.0e-6`.

`float` Single-precision floating point values are written exactly the same as double-precision floating point literals, except that they are followed by the letter `f`, e.g., `3.14f`, `1.f` or `1.0e-6f`.

Additionally you can specify literals consisting of a sequence of characters enclosed in double quotes, e.g., `"This message will self-destruct in 10 seconds."`. These are interpreted as C-style strings, which can be conveniently stored in a `std::string` object.

Variable scope

When program flow leaves a block of code, such as the body of a function, all variables that were declared inside that block of code will (by default) be erased from the memory. As a consequence, local function variables can only be used from within the function that they were defined in. Global variables, by contrast, can be accessed from all functions defined in the same file as the variable, and are even available in other files of a project if they are first declared there as **`extern`** variables. Programmers use the concept 'scope' to refer to the stretch of code from where a variable can be accessed. Variables are defined by the combination of their name and their scope. No two variables within the same scope can have the same name, but it is possible for two variables to have the same name if they differ in scope (i.e., a local variable may have the same name as a global variable; these will be stored at separate memory locations and will generally have different values). Thanks to this rule, it will not be a problem if you accidentally define a local variable with the same name as a variable in some global library that you are not aware of. However, the deliberate use of identical names for different variables is a likely source of confusion and should therefore be avoided.

13.2 Statements and program flow

A simple C++ statement corresponds to an individual instruction in a program, such as a variable declaration or an expression statement like `x = a + b;`. Simple statements always end with a semicolon (`;`), and are executed in the same order in which they appear in a program. Multiple statements can be grouped together to form a compound statement, which is a block of instructions enclosed in curly braces:

```
{ statement1; statement2; statement3; }
```

Compound statements occur frequently as an element of flow control constructs, which force the program to deviate from the default linear execution of the code. C++ provides flow control statements that serve to either repeat segments of code (iterative statements), or take decisions and bifurcate (conditional statements). The flow control statements explained in this section require a generic (sub)statement as part of their syntax. This statement may either be a simple C++ statement, – such as a single instruction, terminated with a semicolon (`;`) – or a compound statement.

Expression statements and operators

Expression statements typically contain one or several variables that are manipulated by one or several operators. C++ has a large number of operators, including the standard arithmetic operators `+`, `-`, `*`, `/` and `%` (plus versions that combine arithmetic operations with an assignment, such as `+=`, `-=`, `*=`, `/=` and `%=`); the increment (`++`) and decrement (`--`) operators; logical operators `!` (logical NOT), `&&` (logical AND) and `||` (logical OR); relational operators `==` (comparison), `!=` (not equal), `>`, `>=`, `<` and `<=`; bitwise

operators ~ (bitwise complement), & (bitwise AND), | (bitwise OR), ^ (bitwise XOR), << (shift bits left) and >> (shift bits right); as well as various other operators for retrieving memory addresses, accessing elements of compound data types, type conversions, and combining expressions (,).

Beware that the behaviour of some operators may depend on the type of the operands (e.g., the division operator may implement integer of floating point division) and that values in expressions involving operands of different types may be altered by implicit type conversion. The precendence rules for operators are explained in Module 3. When in doubt, or when the default rules are inadequate, use parentheses to control the order of evaluation of complex expressions.

The conditional `if`...`else` statement

The **`if`** keyword is used to execute a statement or block, if, and only if, a condition is fulfilled. Its syntax is:

```
if (condition) statement
```

Here, `condition` is the expression that is being evaluated. If this condition is true, `statement` is executed. If it is false, `statement` is not executed (it is simply ignored), and the program continues right after the entire selection statement. For example, the following code fragment prints the message (`x is positive`), only if the value stored in the `x` variable is larger than 0:

```
1 if (x > 0)
2   std::cout << "x is positive";
```

If `x` is zero or negative, this statement is ignored, and nothing is printed.

If you want to include more than a single statement to be executed when the condition is fulfilled, these statements shall be enclosed in braces (`{}`), forming a block:

```
1 if (x > 0)
2 {
3    std::cout << "the number " << x;
4    std::cout << " is positive";
5 }
```

Selection statements with **`if`** can also specify what happens when the condition is not fulfilled, by using the **`else`** keyword to introduce an alternative statement. Its syntax is:

```
if (condition) statement1 else statement2
```

where `statement1` is executed in case `condition` is true, and in case it is not, `statement2` is executed. For example:

```
1 if (x > 0)
2   std::cout << "x is positive";
3 else
4   std::cout << "x is non-positive";
```

This prints `x is positive`, if indeed x is larger than 0, but if it does not, and only if it does not, it prints `x is non-positive` instead. Several **`if-else`** structures can be concatenated with the intention of checking a range of values. For example:

```
1 if (x > 0)
2   std::cout << "x is positive";
3 else if (x == 0)
4   std::cout << "x is zero";
5 else
6    std::cout << "x is negative";
```

This prints whether `x` is positive, negative, or zero by concatenating two **`if-else`** structures. Again, it would have also been possible to execute more than a single statement per case by grouping them into blocks enclosed in braces (`{}`).

The `for`-loop

Loops repeat a statement a certain number of times, or while a condition is fulfilled. There are several loop constructs in C++. The most verstaile one is the **`for`**-loop, which is designed to iterate a number of times. Its syntax is:

```
for (initialization; condition; increase) statement
```

This loop repeatedly executes `statement` while `condition` is true. In addition, the **`for`**-loop provides specific locations to contain an `initialization` and an `increase` expression, executed before the loop begins the first time, and after each iteration, respectively. These expressions make it possible to use counter variables as `condition`.

The **`for`**-loop works in the following way:

1. `initialization` is executed. Generally, this declares a counter variable, and sets it to some initial value. This is executed a single time, at the beginning of the loop.
2. `condition` is checked. If it is true, the loop continues; otherwise, the loop ends, and `statement` is skipped, going directly to step 5.
3. `statement` is executed. As usual, it can be either a single statement or a block enclosed in curly braces (`{}`).
4. `increase` is executed, and the loop gets back to step 2.
5. the loop ends: execution continues by the next statement after it.

The following example illustrates how to count down from ten to zero by means of a **`for`**-loop:

```
1  // countdown using a for loop
2  #include <iostream>
3
4  int main ()
5  {
6    for (int n=10; n>0; n--) {
7      std::cout << n << ", ";
8    }
9    std::cout << "liftoff!\n";
10   return 0;
11 }
```

which produces the output:

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, liftoff!
```

The three fields in a **`for`**-loop are optional. They can be left empty, but in all cases the semicolon signs between them are required. For example, **`for`** `(;n<10;)` is a loop without initialization or increase (equivalent to a **`while`**-loop; see below); and **`for`** `(;n<10;++n)` is a loop with increase, but no initialization (maybe because the variable was already initialized before the loop). A loop with no condition is equivalent to a loop with true as condition (i.e., an infinite loop).

Because each of the fields is executed in a particular time in the life cycle of a loop, it may be useful to execute more than a single expression as any of initialization, condition, or statement. Unfortunately, these are not statements, but rather, simple expressions, and thus cannot be replaced by a block. As expressions, they can, however, make use of the comma operator (`,`): This operator is an expression separator, and can separate multiple expressions where only one is generally expected. For example, using it, it would be possible for a for loop to handle two counter variables, initializing and increasing both:

```
1 for (n=0, i=100 ; n != i ; ++n, --i)
2 {
3    // whatever here...
4 }
```

Here `n` starts with a value of 0, and `i` with 100, the condition is `n != i` (i.e., that `n` is not equal to `i`). Because `n` is increased by one, and `i` decreased by one on each iteration, the loop's condition will become false after the 50th iteration, when both `n` and `i` are equal to 50 (unless `n` or `i` are modified within the loop).

Jump statements

Jump statements allow altering the flow of a program by performing jumps to specific locations. Jump statements are avoided, if possible, by most programmers, because they are notorious for turning streamlined code into an incomprehensible mess. Still, in some cases, jump statements are necessary or more elegant than alternative solutions. The jump statement you will encounter most often is the **`break`** statement. This statement forces program flow to leave a loop, even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force a loop to terminate before its natural end. For example,

```
#include <iostream>

int main ()
{
  for (int n=10; n>0; n--) {
    std::cout << n << ", ";
    if (n==3)
    {
      std::cout << "countdown aborted!";
      break;
    }
  }
  return 0;
}
```

produces the output:

```
10, 9, 8, 7, 6, 5, 4, 3, countdown aborted!
```

The **`continue`** statement is another jump statement that is used to modify the behaviour of loops. Its effect is to cause the program to skip the rest of the loop in the current iteration, as if the end of the statement block had been reached. However, the loop is not terminated completely (unlike what happens with the **`break`** statement), but execution jumps to the start of the following iteration. For example, let's skip number 5 in our countdown:

```
#include <iostream>

int main ()
{
  for (int n=10; n>0; n--) {
    if (n==5) continue;
    std::cout << n << ", ";
  }
  std::cout << "liftoff!\n";
  return 0;
}
```

Now, the ouput is:

```
10, 9, 8, 7, 6, 4, 3, 2, 1, liftoff!
```

Additional flow-control statements

C++ has several additional flow-control statements, such as the conditional expression, the **`switch`**-statement for multibranch conditions, the **`while`** and **`do`**...**`while`** alternatives for the **`for`**-loop. For more information about these alternatives, please consult the advanced material sections in Modules 4 and 5.

13.3 Functions

Functions enable a program to be structured in segments of code that perform individual tasks. In C++, a function is a group of statements that is given a name, and which can be called from some point of the program. The most common syntax to define a function is:

```
return_type function_name( parameter1, parameter2, ...) { function_body }
```

Where:

- `return_type` is the type of the value returned by the function.
- `function_name` is the identifier by which the function can be called.
- each `parameter` (there can be as many as needed) consists of a type followed by an identifier, and is separated from the next parameter by a comma. Each parameter looks very much like a regular variable declaration (for example: **`int`** `x`), and in fact acts within the function as a regular variable which is local to the function. The purpose of parameters is to allow passing arguments to the function from the location where it is called from.
- `function_body` is a block of statements surrounded by braces `{}` that specify what the function actually does.

Here is an example:

```
1  #include <iostream>
2
3  int addition(int a, int b)
4  {
5    int r;
6    r = a + b;
7    return r;
8  }
9
10 int main ()
11 {
12   std::cout << "Adding 5 to 3 yields " << addition(5,3);
13   return 0;
14 }
```

This program is divided in two functions: `addition` and `main`. No matter the order in which they are defined, a C++ program always starts by calling `main`. In fact, `main` is the only function called automatically, and the code in any other function is only executed if its function is called from `main` (directly or indirectly).

In the example above, `main` begins by writing a string to the output stream object `std::cout` and right after that, it performs the first function call: it calls `addition`. Note that the call to the function follows a structure very similar to its declaration. In particular, the parameters in the function declaration have a clear correspondence to the arguments passed in the function call. The call passes two values, 5 and 3, to the function; these will be used to initialise the local variables **`int`** `a` and **`int`** `b` declared within the function `addition`.

At the point at which the function is called from within `main`, the control is passed to function `addition`: here, execution of `main` is stopped, and will only resume once the `addition` function ends. At the moment of the function call, the value of both arguments (5 and 3) are copied to the local variables `a` and `b`. Then, inside `addition`, another local variable is declared (**`int`** `r`), and by means of the expression `r = a + b`, the result of `a` plus `b` is assigned to `r`. The final statement within the function,

```
return r;
```

ends function `addition`, and returns the control back to the point where the function was called; in this case, to function `main`. At this precise moment, the program resumes its course on `main` returning exactly at the same point at which it was interrupted by the call to `addition`. Additionally, because `addition` has a return type, the call is evaluated as having a value, and this value is the value specified in the **`return`** statement that ended `addition`: in this particular case, the value of the local variable `r`, which at the moment of the return statement had a value of 8. In summary, the call to `addition` is an expression

with the value returned by the function, as if the entire function call (`addition(5,3)`) was replaced by the value it returns (i.e., 8). Using this principle, it becomes clear that function calls can also be nested, such as in

```
std::cout << "Adding 5 to 3 to 2 yields " << addition(5, addition(3, 2));
```

Functions that return no value

Not all functions need to produce a return value; there can also be functions that are called simply because they perform some useful task, such as printing a message to the screen. For such functions, it is common to specify the return type as **`void`**, which is a special type to represent the absence of value. Here is an example:

```
#include <iostream>

void printmessage()
{
  std::cout << "I'm a very useful function!";
}

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

`void` can also be used in the function's parameter list to explicitly specify that the function takes no actual parameters when called. For example, `printmessage` could have been declared as:

```
#include <iostream>

void printmessage(void)
{
  std::cout << "I'm a very useful function!";
}

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

The use of **`void`** in the argument list was popularized by the C language, where this is a requirement. However, in C++, leaving the parameter list empty is perfectly fine. Something that in no case is optional are the parentheses that follow the function name, neither in its declaration nor when calling it: even when the function takes no parameters, at least an empty pair of parentheses shall always be appended to the function name (see how `printmessage` was called the example above: `printmessage();`).

Passing function arguments by reference

In the functions seen earlier, arguments have always been passed *by value*. This means that, when calling a function, what is passed to the function are the values of these arguments on the moment of the call, which are copied into the variables represented by the function parameters. For example, take:

```
int x = 5, y = 3, z;
z = addition(x, y);
```

In this case, function addition is passed 5 and 3, which are copies of the values of `x` and `y`, respectively. These values (5 and 3) are used to initialize the variables set as parameters in the function's definition, but any modification of these variables within the function has no effect on the values of the variables `x`

and y outside it, because x and y were themselves not passed to the function on the call, but only copies of their values at that moment.

In certain cases, though, it may be useful to access an external variable from within a function. To do so, arguments can be passed *by reference*, instead of by value. For example, the function duplicate in this code duplicates the value of its two arguments, causing the variables used as arguments to actually be modified by the call:

```
#include <iostream>

void duplicate (int &a, int &b)
{
  a *= 2;
  b *= 2;
}

int main ()
{
  int x = 1, y = 3;
  std::cout << "before: x=" << x << ", y=" << y << std::endl;
  duplicate (x, y);
  std::cout << "after:  x=" << x << ", y=" << y;
  return 0;
}
```

The output of the program clearly shows that the variables x and y are modified by the function call:

```
before: x=1, y=3
after:  x=2, y=6
```

To gain access to its arguments, the function must declare its parameters as references. In C++, references are indicated with an ampersand (&) following the parameter type, as in the parameters taken by duplicate in the example above.

When a variable is passed by reference, what is passed is no longer a copy, but the variable itself. The variable identified by the function parameter, becomes somehow associated with the argument passed to the function, and any modification of the corresponding local variables within the function is reflected in the variables passed as arguments in the call. In fact, a and b become *aliases* of the arguments passed on the function call (x and y) and any change on a within the function is actually modifying variable x outside the function, and any change on b modifies y. That is why when, in the example, function duplicate modifies the values of variables a and b, the values of x and y are affected.

If instead of defining duplicate as

```
void duplicate(int &a, int &b)
```

we would have defined the parameters without the ampersand signs as:

```
void duplicate(int a, int b)
```

then, the variables would not be passed by reference, but by value, creating instead copies of their values. In this case, the output of the program would have been different:

```
before: x=1, y=3
after:  x=1, y=3
```

Calling functions efficiently

Calling a function with parameters taken by value causes copies of the values to be made. This is a relatively inexpensive operation for fundamental types such as **int**, but if the parameter is of a large compound type, it may result in certain overhead. For example, consider the following function:

```
std::string concatenate (std::string a, std::string b)
{
  return a + b;
}
```

This function takes two strings as parameters (by value), and returns the result of concatenating them. By passing the arguments by value, the function forces `a` and `b` to be copies of the arguments passed to the function when it is called. And if these are long strings, it may mean copying large quantities of data just for the function call.

Making the copy can be avoided altogether if both parameters are made references:

```
std::string concatenate (std::string &a, std::string &b)
{
  return a+b;
}
```

Arguments by reference do not require a copy. The function operates directly on (aliases of) the strings passed as arguments, and, at most, it might mean the transfer of certain memory addresses to the function. In this regard, the version of concatenate taking references is more efficient than the version taking values, since it does not need to copy expensive-to-copy strings. On the flip side, functions with reference parameters are generally perceived as functions that modify the arguments passed, because that is why reference parameters are actually for. The solution is for the function to guarantee that its reference parameters are not going to be modified by this function. This can be done by qualifying the parameters as constant:

```
std::string concatenate (const std::string &a, const std::string &b)
{
  return a+b;
}
```

By qualifying the parameters as **`const`**, the function is forbidden to modify the values of neither `a` nor `b`, but can actually access their values as references (aliases of the arguments), without having to make actual copies of the strings.

Therefore, **`const`** references provide functionality similar to passing arguments by value, but with an increased efficiency for parameters of large types. That is why they are extremely popular in C++ for arguments of compound types. Note though, that for most fundamental types, there is no noticeable difference in efficiency, and in some cases, **`const`** references may even be less efficient!

13.4 Compound data types

Fundamental types represent the most basic types handled by the machines where the code may run. But one of the major strengths of the C++ language is its rich set of compound types, of which the fundamental types are mere building blocks.

`std::string` objects

An example of a compound type is the `std::string` class. Variables of this type are able to store sequences of characters, such as words or sentences. Compound types are used in the same way as fundamental types: the same syntax is used to declare variables and to initialize them. A difference with fundamental data types is that in order to declare and use objects (variables) of this type, the program needs to include the header where the type is defined within the standard library (header `<string>`):

```
#include <iostream>
#include <string>

int main ()
{
  std::string aString;
  aString = "\"Real men wear boxer shorts!\"\n";
  std::cout << "Arnold says: " << aString;
  return 0;
}
```

As you can see in the previous example, strings can be initialized with any valid string literal, just like numerical type variables can be initialized to any valid numerical literal. As with fundamental types, all initialization formats are valid with strings:

```
std::string aString = "\"Real men wear boxer shorts!\"\n";
std::string aString("\"Real men wear boxer shorts!\"\n" );
std::string aString {"\"Real men wear boxer shorts!\"\n"};
```

Strings can perform all basic operations that fundamental data types can, like being declared without an initial value and change its value during execution. They also behave intuitively in expressions, owing to the fact that some operators have been redefined for strings. For example, the ordinary plus operator can be applied to string objects to concatenate them. For example:

```
1  #include <iostream>
2  #include <string>
3
4  int main ()
5  {
6    std::string aString;
7    aString = "\"Real men wear boxer shorts!\"\n";
8    std::cout << "Arnold says: " + aString;
9    return 0;
10 }
```

results in the output

```
Arnold says: "Real men wear boxer shorts!"
```

Additional functionality for `string` objects can be accessed via the many member functions of the `string` class, which allow you to insert, remove or replace characters at specific positions; paste or cut substrings into/from another string; or search a string for subsequence patterns. For details, please consult Module 9 or the reference manual on www.cplusplus.com.

std::vector objects

The C++ Standard Template Library (STL) provides several additional predefined compound data types. One that is used frequently is the container class `std::vector<>` which can be used to work with collections of identical elements. The elements can be of an arbitrary type that has to specified as a template argument. For instance, the following code defines a vector of integer variables:

```
1  #include <vector>
2
3  int main ()
4  {
5    std::vector<int> myVector;
6    return 0;
7  }
```

Note that, in order to use vector objects, it is necessary to include the header `vector` from the STL.

Vector objects can be constructed and initialised in a variety of ways:

`std::vector<int> vec;`	Creates a vector with no elements initially
`std::vector<int> vec(8);`	Creates a vector of eight elements, which will not be initialised
`std::vector<int> vec(8, 0);`	Creates a vector of eight elements, which will all be initialised with value 0
`std::vector<int> vecCopy(vec);`	Creates a vector that is initialised as a copy of `vec`
`std::vector<int> fibnc {1, 1, 2, 3, 5, 8};`	Creates a vector with a copy of each of the elements in the initialisation list, in the same order.
`std::vector<int> rabbits(fibnc.begin(),fibnc.end());`	Creates a vector with elements copied from the range `fibnc.begin()` to one element before `fibnc.end()`.

In addition, just like strings, they have many member functions that allow you to manipulate entire vector objects or individual elements. The following code, which simulates the growth of a rabbit population under idealised conditions, illustrates some of them: `[ ]` to access vector elements, `back()` to access the last element, `push_back()` to add an element to the end and `size()` to retrieve the number of elements in the vector. The output of the program is the well-known Fibonacci sequence

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 (and so on)
```

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

int main()
{
   // create initial vector
   std::vector<int> rabbits {0, 1};

   // iterate recursion until rabbits fill the earth
   for (int n0 = rabbits[0], n1 = rabbits[1]; n1 < 999999; n0 = n1, n1 = rabbits.back())
      rabbits.push_back(n0 + n1);

   // show growth of rabbit population on screen
   for (int t = 0; t < rabbits.size(); ++t)
      std::cout << rabbits[t] << ' ';

   // empty vector object
   rabbits.clear();

   return 0;
}
```

Custom-made compound data types

Since `std::vector` (and other STL-containers) can contain elements of arbitary type, the possibilities to build up complex data structures are almost limitless. For example, a `std::vector` of `std::vector` objects can act as a convenient representation of a matrix. For example,

```
std::vector<std::vector<int> > myMatrix(4, std::vector<int>(3, 0));
```

defines a 4×3 matrix of integers; each row of the matrix is initialised with a `std::vector<int>` of 3 zeroes. In a similar way, one can build a data structure representing a phone directory by declaring a `std::vector` of `std::pair` objects, each containing a `std::string` (for the name) and an integer (to represent the phone number). Here is how to do it:

```
#include <utility>    // contains definition of std::pair
#include <string>
#include <vector>
#include <iostream>

int main()
```

```
{
   // declare directory as a std::vector of std::pair<std::string, int> objects
   std::vector<std::pair<std::string, int> > directory;

   // build up phone directory from user input
   for (;;) {
      std::cout << "Please enter a name, or type STOP to quit : ";
      std::string name;
      std::cin >> name;
      if (name == "STOP") break;

      std::cout << "Please enter the phone number for " << name  << " : ";
      int number;
      std::cin >> number;

      directory.push_back(std::make_pair(name, number));
   }

   // show phone directory on screen
   std::cout << "The following data were entered.\n";
   for (int i = 0; i < directory.size(); ++i)
      std::cout << directory[i].first << " : " << directory[i].second << '\n';

   return 0;
}
```

If the built-in compound data types do not fit your requirements, you can also build custom compound data types from scratch. The basis for doing so are **`struct`** and **`class`** objects, which allow you to define objects that contain multiple member variables. The member variables can be of different type. Oftentimes, the variables are protected from being manipulated from outside the class. This so-called *data encapsulation* is achieved by declaring class data members as **`private`**. In order to enable the user to manipulate the data inside an object, the class must then typically be extended with **`public`** member functions that provide controlled access to the data members. The following program shows a simple example of a **`class`** definition with data encapsulation. The class implements the familiar hangman game; internally, it has a secret word that has to be guessed letter-by-letter by the user. It also keeps track of the letters that were guessed correctly and counts the number of wrong guesses. There are only two member functions: one sets up a new game, the other processes a guess entered by the user. Both manipulate the internal data members, to which the user has no direct access.

Code listing 13.1: Data encapsulation in a hangman game

```
#include <string>
#include <iostream>

//*** class definition ****************************

class HangmanGame {
public:
   void setSecretWord(const std::string&);
   bool guess(char);
private:
   std::string word, correctLetters;
   int remainingGuesses;
};

//*** definition of the member functions **********

void HangmanGame::setSecretWord(const std::string &newWord)
// starts a new game
{
   word = newWord;
   correctLetters.clear();
   remainingGuesses = 9;
```

```
}

bool HangmanGame::guess(char ch)
// processes the letter guessed by the user and provides feedback
// returns false if no more guesses remain or the user guessed the word
{
   // check if the word contains the letter...
   std::string::size_type found = word.find(ch);
   if (found == std::string::npos) {
      --remainingGuesses;
      std::cout << "Sorry, wrong guess! You have " << remainingGuesses << " guesses
     remaining.\n";
      if (remainingGuesses == 0) {
         std::cout << "Too bad... You lost.\n";
         return false;
      }
   }
   else correctLetters.push_back(ch); // ... if so, add it to the correctLetters string

   // show the word for as far as it has been guessed
   bool isSomeLettersWrong = false;
   std::cout << "\n\tGuessed so far: ";
   for (int i = 0; i < word.size(); ++i) {
      found = correctLetters.find(word[i]);
      if (found == std::string::npos) {
         isSomeLettersWrong = true;
         std::cout << '.';
      }
      else std::cout << word[i];
   }

   if (!isSomeLettersWrong)
      std::cout << "\n\nCongratulations! You guessed the word.\n";
   else
      std::cout << "\n\nPlease try again. ";

   return isSomeLettersWrong;
}

//*** main() program ****************************

int main()
{
   // declare HangmanGame object and set secret word
   HangmanGame hangman;
   hangman.setSecretWord("beekeeper");

   // start game
   char ch;
   do {
      std::cout << "Enter a letter to guess the secret word : ";
      std::cin >> ch;
   } while (hangman.guess(ch));

   return 0;
}
```

13.5 Input/output

C++ uses so-called stream objects to perform input and output operations to the screen, the keyboard or a file. A stream is an entity where a program can either insert or extract characters to/from. There is no need to know details about the media associated to the stream or any of its internal specifications. All we need to know is that streams are a source/destination of characters, and that these characters are provided/accepted sequentially (i.e., one after another).

Standard stream objects

The standard library defines a handful of stream objects that can be used to access standard sources and destinations in the environment where the program runs:

stream	description
`std::cin`	standard input stream
`std::cout`	standard output stream
`std::cerr`	standard error (output) stream
`std::clog`	standard logging (output) stream

On a standard computer, `std::cin` receives information from the keyboard, whereas the three output streams `std::cout`, `std::cerr` and `std::clog` print information on the screen. In fact, `std::cerr` and `std::clog` essentially work in the same way as `std::cout`, and their only purpose is to create streams specific for error messages and logging. If desired, these streams can be individually redirected, to separate error messages, log information and other program output into different files, for example. For example, the following code illustrates how information sent to `std::clog` can be redirected towards a file `message.log`.

Code listing 13.2: Redirecting a standard output stream

```
#include <iostream>
#include <fstream>

int main()
{
   // std::cout and std::clog are coupled to the screen initially
   std::cout << "The first message will be shown on the screen\n";
   std::clog << "The second message will also be shown on the screen\n";

   // create file stream
   std::ofstream ofsLog("message.log");

   // redirect the read buffer of std::clog to the file
   std::clog.rdbuf(ofsLog.rdbuf());

   // now, std::cout and std::clog are separate
   std::cout << "The third message will still be shown on the screen\n";
   std::clog << "The fourth message will be sent to a file\n";

   return 0;
}
```

File streams

As suggested by the previous example, it is relatively straightforward to set-up a stream to a file. These can be accessed directly (i.e., without redirecting one of the standard streams to them) either to store output or to read in information from the disk drive. File streams are defined in the header `fstream`, and there are two basic types, `std::ofstream` for output to a file, and `std::ifstream` for input from a file. The following expressions illustrate how to work with these stream objects

`std::ofstream ofs(file.txt);`	Create stream `ofs` for write access to file `file.txt` (previous contents of the file will be overwriten).
`std::ofstream ofs(file.txt, std::ios::app);`	Create stream `ofs` for write access to file `file.txt` (new information will be appended at the end).
`std::ifstream ifs(file.txt);`	Create stream `ifs` for read access to file `file.txt`.
`fs.open(file.txt);`	Open file `file.txt` for access via stream `fs`; (to open a file after the declaration of the file stream).
`fs.is_open();`	Check file access status for stream `fs`.
`ifs >> x;`	Read new value for `x` from input stream `ifs`.
`ofs << x;`	Write value of `x` to output stream `ofs`.
`fs.close();`	Close input or output stream `fs`.

In summary, once you have defined a `std::ifstream` or `std::ofstream` object, you can use it for input and output in exactly the same way as you would otherwise use `std::cin` and `std::cout`, respectively. The examples in the following section therefore readily generalise to file stream operations.

Output operations

To produce formatted output, the stream `std::cout` must be used together with the insertion operator, which is written as <<. For example:

`std::cout << "Hello";`	prints: `Hello`
`std::cout << Hello;`	prints the value of the variable `Hello`
`std::cout << false;`	prints: `0`
`std::cout << 'a';`	prints: `a`
`std::cout << +1.00;`	prints: `1`

As illustrated by these examples, the behaviour of the insertion operator depends on the type of its second operand and it may automatically implement certain conversions. Multiple insertion operations (<<) may be chained in a single statement:

```
std::cout << "This statement" << " produces " << "a single message.";
```
prints: `This statement produces a single message.`

Chaining insertions is especially useful to mix literals and variables in a single statement, such as in:

```
std::cout << "Master Yoda is " << yodaAge << " years old and lives on the planet " <<
                                  yodaHomePlanet << '.';
```

Assuming that the `yodaAge` variable contains the value `900` and `yodaHomePlanet` is a `std::string` containing `"Dagobah"`, the output of the previous statement would be:

```
Master Yoda is 900 years old and lives on the planet Dagobah.
```

`std::cout` does not automatically add line breaks, but you can instruct it to do so by inserting a newline character `'\n'`. Alternatively, you can force a line break by inserting the output manipulator `std::endl`. Accordingly,

```
1 std::cout << "A crash reduces\nYour expensive computer\nTo a simple stone."
2 std::cout << "A crash reduces" << std::endl
3       << "Your expensive computer" << std::end
4       << "To a simple stone."
```

are two statements that produce the same output (note that long output statements can be split across multiple lines in the C++ code)

```
A crash reduces
Your expensive computer
To a simple stone.
```

Still, there is a suble difference between the effects of `'\n'` and `std::endl`: in addition to inserting a line break (`'\n'`), `std::endl` will also force the output buffer to be flushed.

Apart from `std::endl`, there are several other I/O manipulators that allow you to precisely control the format of output. For example, `std::scientific` forces floating point values to be written in scientific notation, and `std::boolalpha` causes boolean values to be shown as `true` or `false` instead of `1` or `1`, respectively. Table 10.1 provides a list of commonly used I/O manipulators. The ones that take an argument can only be used after including the header `iomanip`.

Input operations

Much of what was said for output operations applies analogously to input operations. In particular, `std::cin`, which takes input from the keyboard in most environments, is used in combination with an extraction operator >>. This operator is followed by a variable whose value has to be modified by the input operation. Multiple input operations can be concatenated, and the way input information is interpreted can be controlled by inserting I/O modifiers.

An issue to be aware of is that input operations are quite vulnerable to unexpected user behaviour. `std::cin` will ignore leading whitespace characters (spaces, tabs and line breaks) when trying to interpret user input, but otherwise the input has to exactly match to whatever `std::cin` expects for the type of the variable that is being read. For example, in the following input operation,

```
bool condition;
std::cin >> condition;
```

the user has to enter either a 0 or 1; other values, including `true` or `false` will not be interpreted correctly, causing the input operation to fail. In this particular example, user input `true` or `false` will be processed correctly when the input stream is first modified by the `std::boolalpha` manipulator, as in:

```
bool condition;
std::cin >> std::boolalpha >> condition;
```

However, in that case, entering 0 or 1 will result in unexpected behaviour. The key point here is that it is always very important to verify that input operations occurred as intended. This can be done by checking the error status of the input stream and by verifying that the value assigned to the variable is reasonable. Several `std::ifstream` member functions are available to diagnose potential problems, including `eof()` (signals that the end of the input file was reached), `bad()` (signals read/write errors) `fail()` (signals logical errors) and `good()` (signals that everyting is fine). Moreover, failed input operations also reveal themselves by their expression value (`std::cin >> x` evaluates to **`false`** when the input operation went wrong for one reason or another). This feature is exemplified on line 15 of the following code, which will read strings from an input file until the end of the file is reached:

```
1  #include <iostream>
2  #include <fstream>
3  #include <string>
4
5  int main() {
6
7     // open file
8     std::ifstream ifs("ArnoldsJuicySecrets.txt");
9
10    if (ifs.is_open()) {
11       std::cout << "The file was opened correctly.\n";
12
13       // read contents from file word by word, and print to screen
14       std::string word;
15       while (ifs >> word)
16          std::cout << word << ' ';
17    }
18    else std::cout << "Unable to open file.\n";
19    return 0;
20 }
```

Program arguments

It is possible to pass information to a C^{++} program at run-time by way of providing command-line arguments. These arguments are processed by two-predefined optional parameters of the function `main()`. The first is an integer, commonly defined as **`int`** `argc`, which will contain the number of arguments passed to `main()`. The second is an array of C-style strings, defined as **`char*`** `argv[]`, which contains the actual arguments. There is always at least one argument, i.e., `argv[0]` is a C-style string that stores the name of the executable of your program.

One common application of program arguments is to pass the name of an input datafile to the program (for example, the program could then be started from the command prompt as

`my_program my_parameter_file.txt`)

The following code fragment shows how such a command-line argument could be processed to open a file:

```
1 #include <cstdlib>
2 #include <iostream>
3 #include <fstream>
```

```
4
5  int main(int argc, char *argv[])
6  {
7     // process program arguments
8     if (argc > 1) {
9        // open an input file; the filename was provided as a program argument stored in argv[1]
10       std::ifstream ifs(argv[1]);
11       if (ifs.is_open()) {
12          // read data from file
13          // ifs » ...
14       }
15       else {
16          std::cerr << "Unable to open input data file " << argv[1] << '\n';
17          exit(EXIT_FAILURE);
18       }
19    }
20    else {
21       std::cerr << "Please provide name of input data file as a program argument\n";
22       exit(EXIT_FAILURE);
23    }
24
25    // enter more code here ...
26
27    return EXIT_SUCCESS;
28 }
```

13.6 Exercises

13.1. The hangman game. Building on the code of Listing 13.1, extend the functionality of the hangman game so that it will automatically read the secret word from a text file `hangman_words.txt` containing a list of challenging hangman words. To prevent the game from becoming boring, the list of words in the file has to be changed every time the code is run. This will be done by cycling words in the upper half of the word list, and moving words that have been guessed correctly to the bottom. Also, the list of hangman words must be allowed to grow. This will be done by rewarding users who guessed the secret word correctly by an opportunity to suggest a new word challenge. Proceed in the following steps:

1. Create a plain text file `hangman_words.txt` with some challenging hangman words, such as

absurd	funny	nightclub
awkward	icebox	quorum
bagpipes	jackpot	rhubarb
beekeeper	jazz	subway
buzzard	keyhole	wavy
espionage	microwave	zipper

   and store this file in the running directory of your C++ program.

2. Write code to open the file, and read its contents into a `std::vector<std::string>` object.

3. Use the first word in the list to set the secret word of the hangman game.

4. Write code to manipulate the list of secret words after the game has ended. If the user has guessed the word correctly, then the secret word of the current game has to be moved to the end of the word list, and the user is asked to suggest a new challenge word, which is added to the top of the word list. The first half of the word list is next cycled (irrespective of whether the word was guessed correctly or not), in such a way that the word on the second position moves to the top, and all other words in the first half of the list move up one position, except for the first word, which is moved down to halfway in the list.
   Hint: A simple way to infer the outcome of the game, is to extend the `class HangmanGame` with a member function

   ```
   int getRemainingGuesses() const {return remainingGuesses;}
   ```

If this function returns 0 after the game has ended, the user lost; otherwise, the game ended because the user guessed the word correctly.

5. Write the contents of the new word list to the file `hangman_words.txt`, overwriting its previous contents.

13.2. The hangman game – continued (⋆). Extend the hangman game of the previous exercise to enable users to request a hint when they enter a question mark instead of a letter while they are playing the game. Requesting a hint must be costly (i.e., subtract one additional `remainingGuesses` for each hint), and hints should only be given if three or more characters are unknown and the user has at least three `remainingGuesses`. For every hint requested by the user, the program should reveal one previously unknown letter of the secret word.

Test your skill

13.3. The neighbour-joining method for phylogenetic tree reconstruction (⋆⋆). Various algorithms exist for reconstructing a phylogenetic tree from sequence data. A fast method that is not that difficult to implement is the *neighbour-joining* method, proposed by Saitou and Nei[1]. Neighbor joining takes as input a distance matrix that specifies the pairwise dissimilarity between taxa in a group of species. These dissimilarities are often obtained from sequence data, and may then reflect the number of mutational differences observed in a pairwise comparison of DNA sequences between each pair of species. Alternatively, the distances may be derived from morphological characters or other traits that serve to quantify the resemblance between species.

Starting from the distance matrix $\mathbf{D}$, the neighbour joining method proceeds in the following steps:

1. For each pair of species (i, j) from a group of n taxa, calculate a value $\mathbf{Q}(i, j)$ as follows:

$$\mathbf{Q}(i,j) = (n-2)\,\mathbf{D}(i,j) - \sum_{k=1}^{n} \mathbf{D}(i,k) - \sum_{k=1}^{n} \mathbf{D}(j,k)$$

   Here, $\mathbf{D}(i, j)$ is the element on row i and column j of the distance matrix, i.e., the dissimilarity between species i and j.

2. Find the pair of taxa i and j (with $i \neq j$) for which $\mathbf{Q}(i, j)$ is minimal. These taxa are joined to form an internal node (or branch point of the tree). Fig. 13.1 illustrates this step for a group of five species (labeled **a**-**e**). In the first step of constructing the tree (i.e., the step leading from the upper to the middle diagram in Fig. 13.1), the branches to species **a** and **b** are joined, creating an internal branch point **u**.

3. Calculate the distance of the nodes to the new internal node **u**. If i and j were joined in the previous step, then

$$\mathbf{D}(i,\mathbf{u}) = \frac{1}{2}\,\mathbf{D}(i,j) + \frac{1}{2}\,\frac{\sum_{k=1}^{n}\mathbf{D}(i,k) - \sum_{k=1}^{n}\mathbf{D}(j,k)}{n-2}$$

   and

$$\mathbf{D}(j,\mathbf{u}) = \mathbf{D}(i,j) - \mathbf{D}(i,\mathbf{u})$$

   For all other nodes ($k \neq i$, $k \neq j$), compute the distance as

$$\mathbf{D}(k,\mathbf{u}) = \frac{1}{2}\,(\mathbf{D}(i,k) + \mathbf{D}(j,k) - \mathbf{D}(i,j))$$

4. Remove the rows and columns corresponding to species i and j from the distance matrix, and add a row and column to store the distances for the new internal node **u**. Decrease the value of n by one. If $n = 3$, the tree is fully resolved, so that the algorithm can terminate. Otherwise, go back to step 1.

Write a C++ program to reconstruct a phylogenetic tree from a given distance matrix. Proceed in the following steps:

Step 1 – Develop and test your algorithm by first applying it to the following input data file (in plain `.txt` format):

[1] Saitou, N. and Nei, M. (1987). The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Mol. Biol. Evol.* **4**: 406-425.

```
5
a  b  c  d  e
0  5  9  9  8
5  0 10 10  9
9 10  0  8  7
9 10  8  0  3
8  9  7  3  0
```

The structure of the file is as follows: the first number indicates the number of species in the data set; this is followed by a list of species names (each can be read in as a single `std::string`); finally, the file contains the distances between taxa, organised as a matrix such that the element at row i and collumn j corresponds to $\mathbf{D}(i, j)$. This data corresponds exactly to the example presented on https://en.wikipedia.org/wiki/Neighbor_joining. There you can also see in detail how the algorithm reconstructs the phylogenetic tree in two steps, allowing you to check if your code works as expected. Initially, focus on implementing the neighbour-joining algorithm, without worrying about how to visualise the resulting tree. Proceed to step 2 of this exercise once your confident that the algorithm works correctly.

Hint: Note that the distance matrix is symmetric ($\mathbf{D}(i, j) = \mathbf{D}(j, i)$), and so is the matrix $\mathbf{Q}$. In your progam, you can therefore store the data for both matrices in a single square matrix object: for example, use the elements below the diagonal for storing $\mathbf{D}$, and the ones above the diagonal for storing $\mathbf{Q}$.

Step 2 – Adjust your neighbour-joining algorithm in such way that it gradually builds up a representation of the phylogenetic tree in Newick format. In order to do so, store the list of species names from the input file in a `std:vector<std::string>` object. Every time the algorithm joins two nodes, remove the labels for these nodes from the vector of strings and add a new label for the new internal node at the appropiate position in the vector (make sure the elements continue to match up with the rows of the distance matrix). This new label will represent a subtree of joined nodes according to the syntax rules of the Newick format. In particular, if species **a** is joined to species **b**, forming an internal node **u** (as in the middle diagram of Fig. 13.1), and if $\mathbf{D}(\mathbf{a}, \mathbf{u}) = 2$ and $\mathbf{D}(\mathbf{b}, \mathbf{u}) = 3$, then the new label for node **u** is given by the string `(a:2,b:3)`. In subsequent steps of the tree building process, it is possible that internal nodes are joined to other internal nodes or to leave nodes. Then, if node i, with label `iStr`, is joined to node j, with label `jStr`, forming an internal node **z**, the label for the new internal node is given by `(iStr:`$\mathbf{D}(i, \mathbf{z})$`,jStr:`$\mathbf{D}(j, \mathbf{z})$`)`, where $\mathbf{D}(i, \mathbf{z})$ and $\mathbf{D}(j, \mathbf{z})$ represent the distances to the new internal node calculated in step 3 of the neighbour-joining algorithm. The easiest way to construct these labels in C++ is to make use of a `stringstream` object (defined in header `sstream`). Its use is illustrated in the following piece of code:

Code listing 13.3: Stringstream objects

```
#include <string>
#include <iostream>
#include <sstream>

int main()
{
    // iStr and jStr represent labels of two nodes that will be joined
    std::string iStr("(a:1,b:2)"), jStr("c");
    // the following are hypothetical distance values to the new internal node
    double Diz = 4, Djz = 3;

    // declare an output string stream object
    std::ostringstream oss;

    // you can write to a stringstream just as you would to std::cout
    oss << '(' << iStr << ':' << Diz << ',' << jStr << ':' << Djz << ')';

    // the member function str() allows you to assign the contents of the stream to a string
    std::string zStr = oss.str();

    // show output on screen
    std::cout << zStr;
}
```

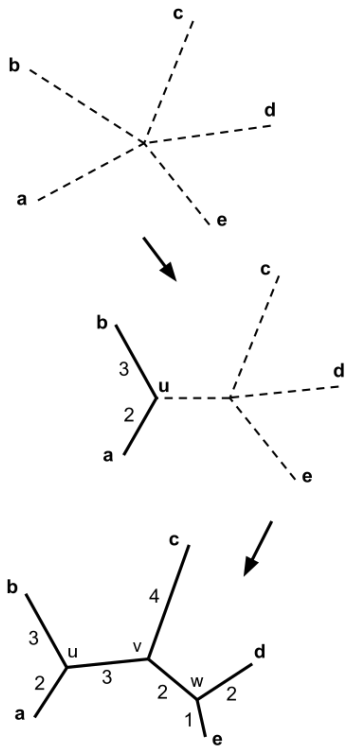


Fig. 13.1: Illustration of two steps of the neighbour-joining method for a group of 5 species. Initially, the tree topology is unresolved, and all species connect to a central node (star topology; upper diagram). Species a and b, which are the nearest neighbours according to the algorithm, are joined first, creating an internal node u (middle diagram). The algorithm then removes a and b from the data set, replacing them by an internal node u, and calculates the distances between the new internal node and the remaining other nodes. This completes one iteration of the algorithm. Subsequent iterations, each one involving the selection of a nearest neighbour pair, the replacement of this pair by an internal node and the recalculation of distances, follow until only three nodes remain (i.e., the algorithm completes in $n-3$ iterations). The lower diagram shows the final tree for the group of 5 species. If the original distances between species were additive, then the dissimilarity between any two species can be reconstructed by adding up the lengths of the tree segments along the path connecting the two species (e.g., based on the neighbour joining tree, we estimate $\mathbf{D}(\mathbf{a},\mathbf{c}) = \mathbf{D}(\mathbf{a},\mathbf{u}) + \mathbf{D}(\mathbf{u},\mathbf{v}) + \mathbf{D}(\mathbf{v},\mathbf{c}) = 2+3+4 = 9$, which is in agreement with the value in the original distance matrix). Image created by Tomfy at English Wikipedia, CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0), via Wikimedia Commons.

The output generated by this program is:

```
((a:1,b:2):4,c:3)
```

Step 3 – After the final iteration, the distance matrix will be reduced to a 3×3 matrix, and the vector of node labels will contain only three elements. If we refer to these as `label[0]`, `label[1]` and `label[2]`, the entire tree can be represented in Newick format as follows:

$$\texttt{(label[0]:}d_0\texttt{,label[1]:}d_1\texttt{,label[2]:}d_2\texttt{);}$$

where

$$d_0 = \tfrac{1}{2}(\mathbf{D}(1,0) + \mathbf{D}(2,0) - \mathbf{D}(2,1))$$
$$d_1 = \tfrac{1}{2}(\mathbf{D}(1,0) + \mathbf{D}(2,1) - \mathbf{D}(2,0))$$
$$d_2 = \tfrac{1}{2}(\mathbf{D}(2,0) + \mathbf{D}(2,1) - \mathbf{D}(1,0))$$

Construct the Newick representation for the entire tree and write it to an output file. From there, you can load the tree specification into a tree visualisation program, or you can copy-paste the information into an online Newick tree viewer (e.g., http://www.trex.uqam.ca; on Trex-online, choose the 'radial' option to compare your result to the representation in Fig. 13.1).

13.7 Solutions to the exercises

Exercise 13.1. The hangman game.

```
#include <string>
#include <iostream>
#include <fstream>
#include <vector>


//*** class definition ***************************

class HangmanGame {
public:
    void setSecretWord(const std::string&);
    bool guess(char);
    int getRemainingGuesses() const {return remainingGuesses;}
private:
    std::string word, correctLetters;
    int remainingGuesses;
};

//*** definition of the member functions **********

void HangmanGame::setSecretWord(const std::string &newWord)
// starts a new game
{
    word = newWord;
    correctLetters.clear();
    remainingGuesses = 9;
}

bool HangmanGame::guess(char ch)
// processes the letter guessed by the user and provides feedback
// returns false if no more guesses remain or the user guessed the word
{
    // check if the word contains the letter...
    std::string::size_type found = word.find(ch);
    if (found == std::string::npos) {
        --remainingGuesses;
        std::cout << "Sorry, wrong guess! You have " << remainingGuesses << " guesses
    remaining.\n";
        if (remainingGuesses == 0) {
            std::cout << "Too bad... You lost.\n";
            return false;
        }
    }
    else correctLetters.push_back(ch); // ... if so, add it to the correctLetters string

    // show the word for as far as it has been guessed
    bool isSomeLettersWrong = false;
    std::cout << "\n\tGuessed so far: ";
    for (int i = 0; i < word.size(); ++i) {
        found = correctLetters.find(word[i]);
        if (found == std::string::npos) {
            isSomeLettersWrong = true;
            std::cout << '.';
        }
        else std::cout << word[i];
    }

    if (!isSomeLettersWrong)
        std::cout << "\n\nCongratulations! You guessed the word.\n";
    else
```

```
60          std::cout << "\n\nPlease try again. ";
61
62      return isSomeLettersWrong;
63  }
64
65  //*** main() program *****************************
66
67  int main()
68  {
69      // declare HangmanGame object
70      HangmanGame hangman;
71
72      // open file to read in list of secret words
73      std::vector<std::string> secretWords;
74      std::ifstream ifs("hangman_words.txt");
75      if(ifs.is_open()) {
76          std::string word;
77          while(ifs >> word)
78              secretWords.push_back(word);
79          if(secretWords.empty()) {
80              std::cerr << "Error: Empty word list\n";
81              return 1;
82          }
83          else hangman.setSecretWord(secretWords.front());
84          // close file
85          ifs.close();
86      }
87      else {
88          std::cerr << "Error: Unable to retrieve word list\n";
89          return 1;
90      }
91
92      // start game
93      char ch;
94      do {
95          std::cout << "Enter a letter to guess the secret word : ";
96          std::cin >> ch;
97      } while (hangman.guess(ch));
98
99      // ask user to suggest a new challenge word if the game was won
100     if(hangman.getRemainingGuesses()) {
101         std::cout << "Please suggest a new challenge word: ";
102         std::string word;
103         std::cin >> word;
104         // move old secret word to end of list and add new suggestion at the top
105         secretWords.push_back(secretWords[0]);
106         secretWords[0] = word;
107     }
108
109     // cycle word list
110     std::string word = secretWords[0];
111     const int n = secretWords.size() / 2;
112     for(int i = 1; i < n; ++i) secretWords[i - 1] = secretWords[i];
113     secretWords[n - 1] = word;
114
115     // store new word list in file
116     std::ofstream ofs("hangman_words.txt");
117     if(ofs.is_open()) {
118         for(int i = 0; i < secretWords.size(); ++i) ofs << secretWords[i] << '\n';
119         ofs.close();
120     }
121     return 0;
122 }
```

Exercise 13.2. The hangman game – continued. To process hint requests, rewrite the function `bool HangmanGame::guess(char)` as follows:

```
bool HangmanGame::guess(char ch)
// processes the letter guessed by the user and provides feedback
// returns false if no more guesses remain or the user guessed the word
{
    if(ch == '?') {
        // provide a hint
        int nrUnknownLetters = 0;
        for(int i = 0; i < word.size(); ++i) {
            // determine which characters have not been guessed
            std::string::size_type found = correctLetters.find(word[i]);
            if (found == std::string::npos) {
                ++nrUnknownLetters;
                ch = word[i]; // potential hint character
            }
        }
        if(nrUnknownLetters >= 3 && remainingGuesses > 2) {
            correctLetters.push_back(ch);
            remainingGuesses -= 2;
            std::cout << "You have " << remainingGuesses << " guesses remaining.\n";
        }
        else
            std::cout << "Forget it, you have to guess for yourself from here on.\n";
    }
    else {
        // check if the word contains the letter...
        std::string::size_type found = word.find(ch);
        if (found == std::string::npos) {
            --remainingGuesses;
            std::cout << "Sorry, wrong guess! You have " << remainingGuesses << " guesses
    remaining.\n";
            if (remainingGuesses == 0) {
                std::cout << "Too bad... You lost.\n";
                return false;
            }
        }
        else correctLetters.push_back(ch); // ... if so, add it to the correctLetters string
    }

    // show the word for as far as it has been guessed
    bool isSomeLettersWrong = false;
    std::cout << "\n\tGuessed so far: ";
    for (int i = 0; i < word.size(); ++i) {
        std::string::size_type found = correctLetters.find(word[i]);
        if (found == std::string::npos) {
            isSomeLettersWrong = true;
            std::cout << '.';
        }
        else std::cout << word[i];
    }

    if (!isSomeLettersWrong)
        std::cout << "\n\nCongratulations! You guessed the word.\n";
    else
        std::cout << "\n\nPlease try again. ";

    return isSomeLettersWrong;
}
```

Module 14

Exploring the STL

Jamie explains what's on the menu

In this module you will:

- Be introduced to function and class templates.
- Become familiar with different kinds of data structures available in the Standard Template Library (STL), including linked lists, arrays, bitsets and maps.
- Learn about the different ways to iterate over the different STL data containers.
- Explore a selection of useful elementary algorithms available in the STL.

14.1 The Standard Template Library (STL)

The Standard Template Library (STL) is a relatively recent addition to the C^{++} programming language that provides many useful building blocks for constructing C^{++} simulation code. It contains definitions for often-used data structures such as vectors, arrays, linked lists, queues and stacks, as well as elementary algorithms for manipulating these data structures (e.g., sorting a list, finding the smallest element in a vector, or rearranging values in a random order). Even if many of these basic data structures and elementary algorithms are easy to code by yourself, knowing to utilise the STL can save you enormous amounts of time. Moreover, since the STL algorithms and data structures were developed by professional programmers, they will often be faster and more efficient than code that you have developed from scratch. Last but not least, the STL is extremely flexible, making it possible to write code that is reusable and applicable to almost any type of data.

The major problem in using the STL is that its syntax is rather daunting. Also, it relies on some of the most advanced features of the C^{++} language, so you need to have a decent understanding of programming, in order to be able to understand what happens behind the screens when you are using an STL data structure or algorithm. To overcome these problems, we will start at the very foundation of the STL library, and first explore *templates*, a feature of the C^{++} language that allows you to write generic functions and classes that can be applied to various kinds of data. Practising with home-made template functions and classes will then help you understand some of the peculiar syntax that is used in the STL.

14.2 Templates

Generic functions

The template is arguably one of C++'s most nifty features, which allows functions and classes to operate with generic data types. Generic code can be very useful, because many algorithms follow exactly the same logic, irrespective of the type of data that they operate on. However, without a special trick (read: templates!), there is a problem: in C++ you always have to specify exactly the types of a function's arguments and return value. For example, if you define a function `mySwap()`

```
1 void mySwap(int &x, int &y)
2 {
3     int tmp = x;
4     x = y;
5     y = tmp;
6 }
```

then, this function will work fine for **`int`** arguments but not for other types of data. For instance, if you would want to swap two `std::vector<double>` objects in the same program, you would need to define a second version of `mySwap()`:

```
1 void mySwap(std::vector<double> &x, std::vector<double> &y)
2 {
3     std::vector<double> tmp = x;
4     x = y;
5     y = tmp;
6 }
```

Given the possibility of function overloading, there is no fundamental objection against having multiple functions with the same name (as long as they are distinguishable by their argument list). Yet, the solution feels a bit clumsy, because the definitions of the two versions of `mySwap()` are completely identical except for the type names. In addition, if we need to execute the elementary operation of swapping at many different points in our program, it is likely that we may need to add yet additional versions, for swapping `std::string`s, **`char`**s or **`bool`**eans, for example.

To solve the problem once and for all, it is much more convenient to define a function template. This can be done in the following way for the function `mySwap()`:

```
1 template <class T>
2 void mySwap(T &x, T &y)
3 {
4     T tmp = x;
5     x = y;
6     y = tmp;
7 }
```

As you can see, a template definition starts with the keyword **`template`**, followed by a list of template arguments between a pair of angle brackets < >. In this case, there is a single template argument that is specified as **`class`** `T`. This syntax indicates that the template argument has to be specified in the form of a type name, and that the argument `T` in the subsequent specification of the function will serve as a placeholder for this type. Now imagine that you replace all occurrences of `T` by **`int`**, and you will see that the resulting function is exactly the same as the original **`void`** `mySwap(`**`int`**`&, `**`int`**`&)` above. Similarly, replacing `T` by `std::vector<`**`double`**`>` produces a function that is identical to the second version, **`void`** `mySwap(std::vector<`**`double`**`>&, std::vector<`**`double`**`>&)`. Indeed, by providing the adequate template argument, we can now call `mySwap()` to swap arbitrary kinds of data, as shown in the example code below for two **`int`**s and two `std::vector<`**`double`**`>` objects:

Code listing 14.1: A template function in action

```
1  int main()
2  {
3      // swapping to integers
4      int a = 2, b = 3;
5      std::cout << "a = " << a << ", " << "b = " << b <<  '\n';
6      mySwap<int>(a, b);
7      std::cout << "a = " << a << ", " << "b = " << b <<  '\n';
8
9      // swapping to std::vectors
10     std::vector<double> u(1, 0.0), v(5, 1.0);
11     std::cout << "size(u) = " << u.size() << ", " << "size(v) = " << v.size() <<  '\n';
12     mySwap< std::vector<double> >(u, v);
13     std::cout << "size(u) = " << u.size() << ", " << "size(v) = " << v.size() <<  '\n';
14
15     return 0;
16 }
```

The more recent versions of C^{++}do not even require that you specify the template argument in the call to a template function. This type will be deduced automatically by the compiler from the type of the function arguments. In other words, modern compilers will allow you to write:

```
mySwap(a, b);
```

and

```
mySwap(u, v);
```

on line 6 and 12, respectively.

It is important to realise that function templates do not actually circumvent the need to have multiple variants of the same function, unlike you would perhaps be led to believe at first. Templates are, in essence, just a way to instruct the compiler to *automatically generate* the required versions of the function, according to the various ways in which it is called by your program. Hence, if `mySwap()` is called using three different kinds of data, **int**s, `std::string`s and **double**s, the compiler will automatically expand the function template into three overloaded function definitions, one with **int** arguments, one with `std::string` arguments and another one with **double** arguments. These three overloaded functions are referred to as template *specialisations*, i.e., specific instances of the template function.

The process of creating the template specialisations, i.e., *template instantiation*, occurs at compile-time, so before the linking stage. As a consequence, the *definition* of a template must be available within the same code unit as from where it is called. Template function definitions are therefore put in header files, where they can easily be included in multiple code units. By contrast, for ordinary functions it is common to put only a function declaration in the header file, so that the definition of the function can be kept separate in another `.cpp` file.

Jamie Oliver
on nested template specifications

The statement `mySwap< std::vector<double> >(u, v)` on line 12 of code listing 14.1 is an example of a nested template specification, which already foreshadows the flexibility of the STL. In this case, a template data structure (std::vector<double>) is used as a type argument to a function template (`mySwap()`). Make sure to leave a space between the angle brackets in such nested template specifications, because subsequent << or >> may otherwise be confused with the I/O or bitshift operators. Recall that the same problem applies when you declare a complex variable, such as a `std::vector< std::vector<int> >` to store a matrix of integers.

Generic classes

Besides creating templated functions, we can also write template class definitions. For instance, to create our own version of a `std::pair` object, all we need to do is declare a template class (or rather, a template **`struct`**), as in the following example:

Code listing 14.2: A home-grown version of the `std::pair` object

```
1  #include <iostream>
2
3  template <class U, class V>
4  struct MyPair {
5      U first;
6      V second;
7  };
8
9  int main()
10 {
11     MyPair<int, double> P {1, 0.2};
12
13     std::cout << P.first << ' ' << P.second << '\n';
14
15     return 0;
16 }
```

As before, the template definition starts with the keyword **`template`**, followed by a template argument list. The `MyPair` template has two arguments, which both have to be specified as a type name (**`class`** `U` and **`class`** `V`). In the subsequent class/struct definition, `U` serves as a placeholder for the type of the first member of the pair, whereas `V` serves as a placeholder for the type of the second member. On line 11 of the code, you see how an instance of `MyPair<int, double>` is declared and initialised by means of the standard initialiser-list method. Explicitly specifying the template arguments **`int`** and **`double`** is compulsory in this case, because the types of the data members cannot always be inferred automatically from the members of the initialisation list.

A pair structure is a relatively simple object, because all data members are publicly accessible. Typical class objects have encapsulated **`private`** data and member functions that provide controlled access. Creating templates of such more elaborate classes is not extremely complicated, as long as you know the basic syntax rules for defining template member functions. They are illustrated in the following program, which introduces a generic class for counting categorical data:

```
1  #include <vector>
2  #include <iostream>
3
4  template <class T>
5  class CountCases {
6  public:
7      void add(const T&);
8      void print();
9  private:
10     std::vector< std::pair<T, int> > cases;
11 };
12
13 template <class T>
14 void CountCases<T>::add(const T &x)
15 {
16     for(int i = 0; i < cases.size(); ++i)
17         if(cases[i].first == x) {
18             ++cases[i].second;   // increment count for existing case
19             return;
20         }
21     // a new element is added if x was not found among existing cases
22     cases.push_back({x, 1});
23 }
```

```
24
25 template <class T>
26 void CountCases<T>::print()
27 {
28     for(int i = 0; i < cases.size(); ++i)
29         std::cout << cases[i].first << ' ' << cases[i].second << '\n';
30 }
31
32 int main()
33 {
34     const std::string text = "There is a theory which states that if ever anyone discovers
      exactly what the Universe is for and why it is here, it will instantly disappear and be
      replaced by something even more bizarre and inexplicable. There is another theory which
      states that this has already happened. - Douglas Adams, The Hitchhiker's Guide to the
      Galaxy.";
35
36     CountCases<char> letters;
37     CountCases<int> wordLength;
38     for(int i = 0, j = 0; i < text.size(); ++i) {
39         if(text[i] == ' ') {
40             wordLength.add(j);
41             j = 0;
42         }
43         else {
44             letters.add(text[i]);
45             ++j;
46         }
47     }
48
49     wordLength.print();
50     std::cout << '\n';
51     letters.print();
52
53     return 0;
54 }
```

Have a look first at the template class definition on line 4-11. Here, you can see that `CountCases` is a class that depends on one template argument (**`class`** `T`). As before, `T` serves as a placeholder for a datatype. Here, it reflects the type of data that we want to count. For example, the function `main()` invokes two template specialisations (see line 36 and 37): one is used for determining character frequencies in a piece of text (`CountCases<`**`char`**`>`); the other is used for counting word lengths (`CountCases<`**`int`**`>`).

Line 10 of the class definition shows that the **`private`** class data take the form of a `std::vector` of `std::pair` objects. The first and second member of each pair are intended to represent, respectively, a categorical data value and its frequency count in the data set. Correspondingly, the members `std::pair::first` and `std::pair::second` are of type `T` and int, respectively.

The class definition further indicates that `CountCases<T>` has two **`public`** member functions. Their definitions are listed separately, on line 13-24 and 26-31. Note the particular syntax on the first and second line of each function definition. The first part, **`template`** `<`**`class`** `T>`, is to identify the member function as a template. The template argument `T` subsequently appears as part of the class name (in `CountCases<T>::`) and, where needed, as a type specifier for function arguments and local variables.

Lines 36-51 in the function `main()` provide a brief example of how the template class in used in practice. Here, two objects are declared, using distinct template specialisations, one for counting integers, the other for counting characters. The program then reads character-by-character through a piece of text, and delimits the words based on the occurrences of a white-space character. The characters and word lengths obtained from the text analysis are added to the respective count data object on line 40 and 44. The final lines of the code (49-51), call the member functions `print()` to write the observed cases and their frequency counts to the screen.

Non-type and default template parameters (optional)

If you would like to start working with templates yourself, it is helpful to know that template arguments need not necessarily be type names. They can also be numerical values of integral type, or other types of discrete values (such as memory addresses, for instance). Another useful feature, which is also exploited in many STL libraries, is that template arguments can have default values, so that the specification of the argument in a template specialisation becomes optional.

The code below illustrates both features for a template function that generates a sequence of unique numbers $(x_1, x_2, \ldots, x_n \in (0, 1))$ with low discrepancy. A low discrepancy sequence has the following special property: for every subsequence and every interval $I = (a, b) \subseteq (0, 1)$, the proportion of the points within the subsequence that fall within I is very close to the length of the interval, $b - a$, similar to what would be observed *on average* if the x_i were sampled from a uniform distribution on the unit interval. Because of this property, low discrepancy sequences can be utilised to obtain an unbiased sample of values from a domain, without requiring that the number of sample points is known in advance (this restriction precludes the use of a regular sampling grid). Another option is to sample the points randomly from a uniform distribution, but then the sampling variation tends to be much higher.

Before further exploring the application of low discrepancy sequences, let us first take a closer look at the their implementation by a template function. In this case, we choose to define the template in a header file `low_discrepancy_seq.h`:

Code listing 14.3: file `low_discrepancy_seq.h`

```
template <int b, class T = int>
double vanDerCorput()
{
    static T n = static_cast<T>(0);

    ++n;
    const double fac = 1.0 / b;
    double out = 0.0, aux = 1.0;
    for(T k = n; k; )
    {
        aux *= fac;
        out += (k % b) * aux;
        k /= b;
    }
    return out;
}
```

The particular algorithm used for generating the low-discrepancy sequence is one that was first described by a Dutch mathematician, J.G. van der Corput. The n-th number from the sequence it generates is constructed by first writing n as a base-b number, and then reversing this b-ary representation[1]. In the template definition above, the base number b is provided as a template argument, in the form of an integer *value* of type `int` (i.e., not as a type name). In addition, the template has a type argument `class T`, which defaults to the type `int`. This can be used, if desired, to adjust the type of the static variable `n`. For example, if the template function needs to be called extremely often, type `T` can be adjusted to `long` in order to prevent integer overflow.

In order to call the template function, it is required to specify the value of the base number (this can not be deduced automatically by the compiler), but specifying the type argument is optional. For example, the statement

```
for(int i = 0; i < 10; ++i) std::cout << vanDerCorput<2>() << ' ';
```

will write to the screen the first ten values of the binary van der Corput sequence:

```
0.5 0.25 0.75 0.125 0.625 0.375 0.875 0.0625 0.5625 0.3125
```

Similarly,

```
for(int i = 0; i < aZillion; ++i) std::cout << vanDerCorput<3, long>() << ' ';
```

[1] Worked-out examples of how the algorithm works for different base numbers are available on the wikipedia page https://en.wikipedia.org/wiki/Van_der_Corput_sequence. Additional information about low-discrepancy sequences and alternative methods for generating them can be found on: https://en.wikipedia.org/wiki/Low-discrepancy_sequence.

will do the same for the first a-zillion values of the ternary sequence, using **long** integers for the internal calculations.

Although this *optional* section in an *advanced* chapter is the perfect platform for introducing something as esoteric as low-discrepancy sequences, we should perhaps now quickly return to a practical application. As a matter of fact, the following code shows how ET's function template can be employed to generate sampling points for numerically approximating the integral under a complicated function

$$I = \int_0^1 \int_0^1 \sin(2\pi x)\, \cos(2\pi y)\, e^{-x^2-y^2}\, dx\, dy$$

```
#include <cstdlib>
#include <iostream>
#include <cmath>
#include "low_discrepancy_seq.h" // includes vanDerCorput<>() template definition

double f(double x, double y) {
    const double PI2 = 6.28318530717959;
    return sin(PI2 * x) * cos(PI2 * y) * exp(-x * x - y * y);
}

int main()
{
    const double epsilon = 1.0e-6;

    // method 1: calculate integral by sampling points from low-discrepancy sequences
    double I1 = 0.0, sumzz = 0.0;
    int it1;
    for(it1 = 0; ; ++it1) {
        double x = vanDerCorput<2>();
        double y = vanDerCorput<3>();
        double z = f(x, y);
        I1 += z;
        sumzz += z * z;
        if(it1 > 1 && (sumzz - I1 * I1 / (it1 + 1.0)) / (it1 * (it1 + 1.0)) < epsilon) break;
    }
    I1 /= it1;
    std::cout << "Integral = " << I1 << "\niterations = " << it1 << '\n';

    // method 2: calculate integral by sampling points from a uniform distributiomn
    double I2 = 0.0;
    sumzz = 0.0;
    int it2;
    for(it2 = 0; ; ++it2) {
        double x = rand() * 1.0 / RAND_MAX;
        double y = rand() * 1.0 / RAND_MAX;
        double z = f(x, y);
        I2 += z;
        sumzz += z * z;
        if(it2 > 1 && (sumzz - I2 * I2 / (it2 + 1.0)) / (it2 * (it2 + 1.0)) < epsilon) break;
    }
    I2 /= it2;
    std::cout << "Integral = " << I2 << "\niterations = " << it2 << '\n';

    return 0;
}
```

The algorithm for approximation the integral numerically evaluates the function value at different sampling points (x, y) in the unit square, and keeps track of the sum of all $f(x, y)$ values, as well as their standard error. When the standard error of the integral falls below a threshold value (dependent on a tolerance value `epsilon`), the iteration loop breaks, and the computed value of the integral is shown on the screen. When the program is run, method 1 (based on sampling points generated by `vanDerCorput()`) provides the result $I_1 = -0.00222162$ after 89439 function evaluations. This is very close to the value obtained from evaluating the analytical solution, $I = -0.00221218$. Method 2 (based on random sampling

points) converges a bit more quickly (after 88836 function evaluations), but provides an estimate that is far less accurate, $I_2 = -0.00178993$. The reason is that points from a uniform distribution oversample certain regions of the integration domain, at the cost of undersampling other regions (Figure 14.1). By contrast, the sampling points generated by the low-discrepancy sequences are distributed much more evenly over the integration domain, so that the evaluation of the integral is much more reliable.

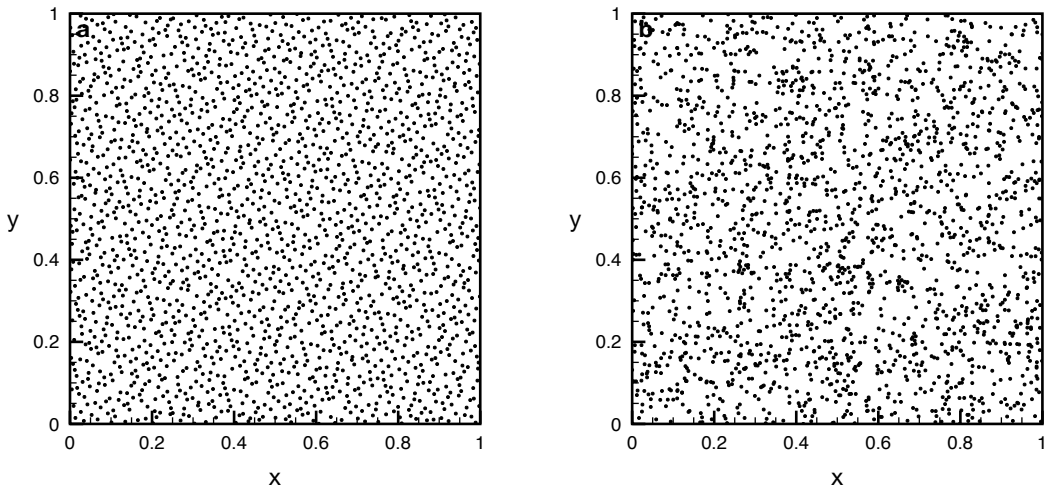


Fig. 14.1: Sampling points for the evaluation of the integral of a function over the unit square. Both plots show 2048 sampling points, generated either by constructing a low-discrepancy (van der Corput) sequence (a), or by sampling from a uniform distribution (b).

One issue to worry about when you develop class or function templates is that users may attempt to create specialisations for types of data that the template is not intended to handle. For example, `vanDerCorput<>()` only works for integral data types that can be handled by the modulo operator, but you can be pretty sure that one day, somewhere, some person will attempt to call `vanDerCorput<float>()` or some other template specialisation that makes no sense. If you're lucky, the compiler will immediately warn that the requested template specialisation is invalid (in this particular example, the compiler will probably signal that the modulo operator cannot handle `float`s), but in other cases the user may be able to create senseless template specialisations without receiving a warning.

Fortunately, the standard library header `<type_traits>` provides some helpful methods to safeguard against the unintended use of templates. It contains a large number of class templates that can be used to check whether a particular data type has the right properties for creating a valid template specialisation. The code below illustrates an application in the context of the `vanDerCorput<>()` function template. Take particular note of the statement on line 7: here the program checks whether the type provided to the template is an integral type; if not, the function throws an exception. Many other kinds of type property evaluation are available in the library `<type_traits>`; you can check online reference databases like `cppreference.com` for an exhaustive list of the possibilities.

```
1  #include <exception>
2  #include <type_traits>
3
4  template <int b, class T = int>
5  double vanDerCorput()
6  {
7      if(!std::is_integral<T>::value)
8          throw std::runtime_error("template argument is not an integral type");
9
10     static T n = static_cast<T>(0);
11
12     ++n;
13     const double fac = 1.0 / b;
14     double out = 0.0, aux = 1.0;
15     for(T k = n; k; )
16     {
17         aux *= fac;
18         out += (k % b) * aux;
```

```
19          k /= b;
20      }
21      return out;
22  }
```

14.3 STL containers

An important core component of the STL is a set of headers that define commonly-used data structures, such as vectors, lists, queues, stacks and sets. They have all been constructed as templates classes, so that they can serve as containers for arbitrary kinds of data. Two of them, you have already met in earlier modules, the `std::vector` and the `std::string`.

Why not to use a `std::vector`

We have worked with `std::vector`s a lot and – so far – they have served us very well in solving all kinds of programming problems. So you might wonder why the STL features so many other container types. The answer is that the different STL container classes have been optimised for different tasks and functions. A `std::vector` object, for example, is great if your application relies on quick and direct access to all elements that are stored in the container. To enable this mode of element retrieval (a.k.a. *random access*), the elements in a vector are stored at adjacent memory addresses. In this way, the value of the `i`-th element in a `std::vector<T>` can be retrieved directly by moving to the memory location of the first element in the vector plus `i` times **`sizeof`**`(T)`. Unfortunately, storing data at consecutive memory addresses has some drawbacks as well. First of all, it is very costly to insert elements somewhere midway in a vector. Since a correct order of the elements in the memory needs to be retained, a large number of items must first be moved to adjacent memory positions in order to create an empty space for the new element. Second, if an element is added to a vector, the memory address following its final element may already be occupied by another variable, making it necessary to first copy all of the vector elements to a sufficiently large block of empty memory space, before the new element can be added. The STL `std::vector` class handles all of these issues very elegantly without you noticing them but, obviously, the copying of elements and allocation of memory space that happens behind the screens will negatively affect the speed of your application.

To give you some insight into how much copying, moving and assigning actually has to be done when you add or insert elements (in)to a vector, the following program defines a simple **`struct`** with explicit definitions for a default constructor, a copy constructor, a move constructor and an assignment operator. Normally, these functions are constructed automatically by the compiler, without you having to worry about them. This time, however, their default implementations have been replaced by explicit versions that will write a character to the screen whenever they are called: this will be an `'N'`, when a new object is created from scratch; a `'C'` when an object is being constructed as a copy of another; an `'M'` when an object is being moved and a `'='` when one object is assigned to another, already existing object of type `A`. Next, six instances of **`struct`** `A` are created and added to a `std::vector<A>` container by repeated calls to the function `push_back()`. After creating these six instances, a seventh element is inserted midway in the vector, by a call to `insert()`.

```
1  #include <iostream>
2  #include <vector>
3
4  struct A {
5      A() { std::cout << 'N';}
6      A(const A&)  { std::cout << 'C'; }
7      A(const A&&) { std::cout << 'M'; }
8      A& operator=(const A&) { std::cout << '='; return *this; }
9  };
10
11 int main()
12 {
13     std::vector<A> vec;
14     std::cout << "6 calls to push_back()\n";
15     for(int i = 0; i < 6; ++i) {
16         std::cout << vec.size() << ' ' << vec.capacity() << ": ";
17         vec.push_back(A());
```

```
18          std::cout << '\n';
19      }
20
21      std::cout << "insertion of element at third position\n";
22      std::cout << vec.size() << ' ' << vec.capacity() << ": ";
23      vec.insert(vec.begin() + 2, A());
24      std::cout << '\n';
25
26      return 0;
27  }
```

The output of the program gives some clues as to how the memory allocation for `std::vector` objects is implemented. Each line below shows the size of the vector and its capacity, followed by a sequence of characters that reveals which of the member functions of **`struct`** `A` were called during the execution of the subsequent `push_back()` and `insert()` operations:

```
6 calls to push_back()
0 0: NM
1 1: NMC
2 2: NMCC
3 4: NM
4 4: NMCCCC
5 8: NM
insertion of element at third position
6 8: NM====
```

Note that the addition of the first element to the empty vector involves a call to the default constructor (to create the element) and another call to the move constructor (to move the element into its correct position in the vector object). At that point, the vector has size one and capacity for one element. The next time `push_back()` is called, the capacity of the vector is insufficient to accommodate a second element. Therefore, a larger block of memory must be reserved, and the existing element of the vector must be copied to that new location (as revealed by the call to the copy constructor). The capacity of the vector is also increased before a third element is added (from two to four elements), and once again (from four to eight elements) before a fifth element is added. In both cases, the existing elements must all be copied to their new memory locations. The final line of output illustrates the problem of inserting elements in a vector: in this particular example, inserting a single element can be seen to require four assignment operations.

After all of this, the message should be clear: stay away from adding or inserting vector elements whenever possible, and optimise your code accordingly (for example, if the required size of a vector is known, constructing a `std::vector` with exactly the right number of elements is much more efficient than creating an empty vector and then adding elements at the end one-by-one.) However, what should you do when adding, inserting and removing elements is unavoidable? Well, that depends on what other functionality the container object needs to be equipped with.

The next considerations may help you decide which of the STL containers is the most appropriate one for a given programming problem:

1. The first question you must ask yourself is whether it is critical to maintain random access. If yes, your best options are the `std::vector`, or potentially the `std::deque` (double-ended queue), which is a data structure that offers random access and efficient addition and removal of elements at the start and the end of the container. Inserting or deleting elements in the middle is a costly operation, both in `std::vector`s and `std::deque`s.

2. If the required size of the container is constant and known at compile-time, you may consider the more efficient but less flexible `std::array` container, for storing vectors of fixed size. An efficient alternative for storing fixed-length sequences of boolean variables is `std::bitset`.

3. If you can live with *sequential access* to the data and you need to be able to efficiently insert or delete elements in the middle, then your default option is the `std::list`. A `std::list` allows you to iterate from one element in the container to the next, in both the forward and backward direction, so it is still straightforward to iterate through all elements from the start to the end (or in the reverse direction). If backward iteration is not needed, an alternative with slightly less overhead is the `std::forward_list`.

4. When you are looking for a container type that minimises overhead and all you need to do is process elements in a particular order, it is worth considering the STL containers `std::queue` and

Table 14.1: Pros and cons of different STL containers

Container class	Design objectives	Constraints
`std::vector`	Random access[1], flexibility	Costly element insertion and deletion; Adding elements at the end requires memory reallocation
`std::array`[2]	Random access[1], minimal overhead	Fixed size
`std::deque`[3,4]	Random access[1], efficient addition of elements at start and end	Costly element insertion and deletion
`std::list`	Fast insertion and deletion, flexibility, bidirectional iteration[5]	Sequential access[6]
`std::set`	Fast look-up, maintenance of sorted list of unique elements	Sequential access, elements cannot be modified inside container

[1] I.e., fast access to each element individually. With random access, you get to the n-th element in the container by jumping n element memory blocks beyond the memory location of the first element, and retrieving the value stored there.
[2] An alternative for storing constant size vectors of **bool**eans is `std::bitset`.
[3] Deque stands for double-ended queue.
[4] The container classes `std::queue` and `std::stack` are simplified adaptors of the `std::deque` that implement, respectively, a queue (*first-in, first-out* system) and a stack (*last-in, first-out* system) .
[5] If forward-only iteration is sufficient, `std::forward_list` is an alternative with less overhead.
[6] With sequential access, you get to the n-th element in the container by locating the first, then jumping from there to the second, the third, and so on, until you get to the n-th element.

`std::stack`. These offer limited access to the data: for a queue, you can only add elements at the end and remove them from the start (so items are processed at a *first come*, *first serve* basis); for a stack, you can only add and remove elements at the top (i.e., a stack behaves as a *last-in*, *first-out* system). For convenience, STL also provides the container class `std::priority_queue`, which emulates a queue where items are placed in order of decreasing priority. This container class is normally implemented as a wrapper around a `std::vector` object, so it is not particularly efficient.

5. If your primary objective is to create a searchable collection of objects, then the most suitable STL containers are often the `std::set` and `std::map`. A `std::set` object maintains an ordered collection of unique elements; elements can be added and removed, but not modified once they are in the container. The container `std::map` is similar, except that it stores collections of *key-value* pairs, like in a phone directory that contains a list of phone numbers (*values*) ordered alphabetically by name (*key*). In a `std::map`, the *keys* cannot be modified once objects have been added to the container, but the *values* can. The strength of the `std::set` and `std::map` containers is that they allow you to look-up specific elements very efficiently. A `std::set` is a suitable choice if you would like to search for items by their value; a `std::map` is intended to search efficiently for items by their key, not their value. The container classes `std::multiset` and `std::multimap` are available to create searchable collections that contain duplicated elements. However, their ability to deal with multiple elements with identical values or keys comes at the cost of reduced search efficiency.

The sections below provide short example programs highlighting some of the different STL container types mentioned above. These examples are intended to illustrate the relative strengths of each container type; they do not provide a comprehensive overview of all of the functionality that is available in the STL (which would require much more space than we have available here). More info and additional example programs can be found on the internet[2].

Fixed-size containers with random access: `std::array` *and* `std::bitset`

The `std::array` is a *no-nonsense* container type designed to store a pre-determined, fixed number of objects of the same type. It is the modern C^{++} equivalent of the array in the predecessor language C, which is used to store multiple variables of the same type at adjacent addresses in the memory. Like the C-style array (and the more versatile C^{++} `std::vector` class), `std::array` has an element access operator `[ ]` that allows direct access to each individual element stored in the container. The same functionality is provided by the member function `at()`, but unlike the regular member access operator `[ ]`, `at()` checks whether its argument is within the range of valid element indices (i.e., larger than or equal to zero and smaller than the size of the array).

The container `std::array` also has a number of familiar member functions like `size()`, `front()` and `back()`, which provide information about the number of elements or that retrieve the value of the first

[2] see, e.g., Master Yoda's favourite online reference manuals `cppreference.com` or `cplusplus.com`

and last element, respectively. Some other functions that are available for `std::vector`, such as `insert()`, `erase()` and `resize()`, are not available for `std::array`, because they change the number of elements in the container. The size of a `std::array` object is specified by a template argument. As a consequence, the size of an array is not just constant, but it must also be known at compile-time. Code listing 14.4 shows an application of the `std::array` in a simulation of Fibonacci's insightful model for the growth of a rabbit population.

Code listing 14.4: `std::array`

```
#include <iostream>
#include <array>

int main()
{
    // template arguments specificy the type of
    // data stored in the std::array and its size
    std::array<int, 10> nrOfRabbitPairs {1, 1}; // initialisation of the first two
                                                // elements by an initialisation list

    // simulate recursion N[t] = N[t - 1] + N[t - 2]
    for(int t = 2; t < nrOfRabbitPairs.size(); ++t)
        nrOfRabbitPairs.at(t) = nrOfRabbitPairs.at(t - 1) + nrOfRabbitPairs.at(t - 2);
        // it is easy to make mistakes with the range of the loop variable t, so to be
        // on the safe side, we used at() instead of the regular element access operator []

    // show values on screen
    for(int t = 0; t < nrOfRabbitPairs.size(); ++t)
        std::cout << nrOfRabbitPairs[t] << ' '; // regular element access operator []

    return 0;
}
```

The STL container type `std::bitset` behaves similar to a `std::array` of `bool` variables, but it has been optimised internally to allow for a much more efficient storage of the data: rather than reserving `sizeof(bool)` bits for storing a single element, `std::bitsets` use only a single bit per element. The `std::bitset` also has additional member functions for bit operations and conversions between bitsets and other types of data, such as `std::string` representations of binary sequences. Finally, variables of type `std::bitset` can be manipulated by the standard bitwise logical operators `&` (bitwise AND), `|` (bitwise OR), `^` (bitwise XOR), `~` (bitwise negation), `<<` (bitshift left) and `>>` (bitshift right) to allow for logical operations on bitstring sequences. Code listing 14.5 illustrates how the versatility of `std::bitsets` can be exploited to sample genotype sequences generated by the recombination of two parental genotypes. A representative output from this program is given by

```
parents:
sequence A: 00000000000000000000000000000000000000000000000000
sequence B: 11111111111111111111111111111111111111111111111111

recombinants:
sequence C: 11111111111111111111111111111111111111111111111111
sequence D: 11111111111111111111111110000000000000000000000000
sequence E: 00000000000011111111111111111111111111111111111000
sequence F: 00000001111111111111111111111111111111111111111111
sequence G: 00000000000000001111111111111111111111111111111111
sequence H: 00000000000000000000000000000000000000000000000000
sequence I: 11111111111111111110000000000000000000000000000000
sequence J: 11111111111111111111111111111111111111111111111111
sequence K: 00001111111111111110000000000000000000001111111111
sequence L: 11111111111100000000000000001111111111111111111111
```

Code listing 14.5: `std::bitset`

```
#include <iostream>
#include <cstdlib>
```

```cpp
#include <bitset>

const int L = 50; // Let's assume that individuals are variable for L loci and that two alleles segregate
                  // at each locus. Then, individual genotypes can be represented by a std::bitset<L>.
const double r = 0.02;  // probability of recombination between subsequent loci

std::bitset<L> recombine(const std::bitset<L> &seq1, const std::bitset<L> &seq2)
// recombines the parental haplotype sequences seq1 and seq2
{
    // Create a bitset rec. Bits in this bitset will specify whether the offspring sequence
    // inherits the allele from parental haplotype sequence 1 or 2.
    std::bitset<L> rec;

    // The first element in rec is random
    if(rand() % 2)
        rec.set(0);      // sets the first bit to one
    else
        rec.reset(0);    // sets the first bit to zero

    for(int i = 1; i < L; i++) {
        // Subsequent loci are inherited from the same haplotype, unless there is a recombination event.
        rec[i] = rec[i - 1];     // access individual bits via the regular access operator []
        if(static_cast<double>(rand()) / RAND_MAX < r)
            rec.flip(i);       // flips the i-th bit, causing alleles to be inherited from the other
                               // sequence from here on until the next crossover event
    }

    // recombine the parental sequences
    return (rec & seq1) | (~rec & seq2);

    /* an alternative implementation that avoids the use of bitwise logical operators is:
    std::bitset<L> out;
    for(int i = 1; i < L; i++) out[i] = rec[i] ? seq1[i] : seq2[i];
    return out;
    */
}

int main()
{
    // seed random number generator
    srand(25784);

    // create two parental genotype sequences
    std::bitset<L> seqA, seqB;    // the default constructor initialises with all zeroes
    seqB.set();                   // sets all bits in the second sequence to one


    std::cout << "parents:\n";
    std::cout << "sequence A: " << seqA << '\n';
    std::cout << "sequence B: " << seqB << '\n';


    // create a bunch of recombinant sequences:
    std::cout << "\nrecombinants:";
    for(char ch = 'C'; ch < 'C' + 10; ++ch)
        std::cout << "\nsequence " << ch << ": " << recombine(seqA, seqB);

    return 0;
}
```

Sequential access to data via a `std::list`

As explained at the start of this section on STL containers, storing individual data elements at adjacent positions in the memory interferes with the ability to quickly insert or delete elements somewhere in the

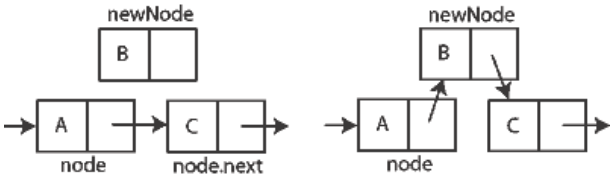


Fig. 14.2: The elements in a linked list are stored together with the memory address of the next element in the sequence (arrows), making it possible to find back all values without requiring them to be stored at adjacent memory positions. However, retrieving the value of an arbitrary element from a linked list is not very efficient: it requires jumping from one element to the next, starting all the way from the beginning of the list. To make up for that, a linked list allows for the insertion and deletion of elements in the middle of the sequence without requiring that existing elements are moved or copied. All that is needed is a rewiring of the memory addresses (right).

middle. But is there an alternative way of storing data that does not suffer from this problem? The good news is: there is, if we are willing to store items together with a little bit of additional information that tells us where they are located in the computer memory. In particular, if we store each item in a sequence of data *together with the memory address of the next item*, and we keep track of where the first element of the sequence is located, then we can jump to that memory location to look up the address of the second item; jump from there to the third memory location; and so on, until we arrive at the specific element in the data sequence that we want to know the value of. The resulting data structure is a so-called linked list, and it is implemented by the STL container `std::list`.

Inserting items somewhere in the middle of a linked list involves a bit of bookkeeping to ensure that the memory addresses point to the right elements, but it does not require the copying of any of the items (Fig. 14.2). Similarly, element deletion can be handled by rewiring the list rather than by moving the elements. The STL container `std::list` handles all of these operations behind the screens, so you do not have to worry about them as a user. In fact, `std::list` is slightly more sophisticated than the singly linked list of Fig. 14.2; it also stores the memory addresses of previous items together with each element, making it possible to also traverse the list from its end to the beginning.

Intermezzo: STL iterators

Because the elements in a `std::list` are not stored at consecutive memory positions, there is no operator `[ ]` that you can use to access individual elements. Instead, iterating through a `std::list` is achieved by jumping from one element to the next by means of special iterator objects. These STL objects are themselves defined as classes that are nested within the data container for which they serve as an iterator. To declare an iterator for stepping through a `std::list` of **int**s, for example, we must write

```
std::list<int>::iterator it;
```

The following code listing illustrates how the iterator can next be integrated into a loop statement that iterates over all elements in a list and writes their values to the screen:

```
#include <iostream>
#include <list>

int main()
{
    std::list<int> myList {1, 2, 6, 24, 120};

    std::list<int>::iterator it;
    it = myList.begin();

    while(it != myList.end()) {
        std::cout << *it << '\n';
        ++it;
    }
    return 0;
}
```

To understand how this program works, take a closer look at:

line 9 Here, the iterator variable `it` is made to point to the start of the list `myList` by calling the member function `begin()`.

line 13 This is where the operator is made to jump one step ahead, to the next element in the linked list. Analogous to how you increment a normal loop counter, advancing an iterator by one step is done by applying the operator `++`.

line 11 As a counterpart to the member function `begin()`, STL containers also have an `end()` function, whose value can be compared to the iterator to determine if it has reached the end of the container. Read more important information about the function `end()` in Yoda's comment below.

line 12 An iterator points to a memory location where the value of a particular list element is stored; therefore, its value reflects a memory address, not the value of the list element! So, how can we retrieve the value of the element that is stored at the location specified by an iterator? In C++ this operation is referred to as *de-referencing* and it has a special operator, denoted as `*`. The dereferencing operator is easily confused with the multiplication operator, but it can be recognised by the fact that it has only one operand, not two. Line 12 illustrates how it is used: the expression `*it` instructs the compiler to look up the value that is stored at the memory location identified by `it`. The value is next passed to the output operator, causing it to appear on the screen.

***Master Yoda* on `end()`**

You would perhaps expect that `end()` returns the memory location of the last element in a container, but this is wrong. The value of `end()` actually points one step beyond the last element of the container. In other words, an iterator variable becomes equal to the value of `end()` once it has stepped beyond the final element of the container. This convention may seem weird, but it is quite helpful. It allows you to incorporate STL iterators in iterative statements in the same way as you would work with ordinary loop counter variables. For example, to iterate over the elements of a `std::array<int, 42> vec` you may write:

```
for(int i = 0; i != 42; ++i) { vec[i] = ...}
```

If you would use STL iterators to perform the same task, you would write:

```
for(std::array<int, 42>::iterator it = vec.begin(); i != vec.end(); ++it) { *it = ...}
```

which uses a qualitatively similar syntax for initialisation, continuation and update.

As suggested by Yoda's example, iterators are not only available for linked lists but also for many other STL containers. Not all iterators have the same functionality however. Those for `std::list` are *bidirectional iterators*, that can be incremented (using operator `++`) or decremented (using operator `--`) to jump to the next or the preceding data item, respectively. Containers that do not allow for reverse iteration, such as `std::forward_list` for example, have iterators that can only be incremented, not decremented. Such iterators are referred to as *forward iterators*. Iterators for random access containers such as `std::vector` and `std::array`, finally, are a bit more versatile than those for a `std::list`. They have all the capabilities of a bidirectional iterator but, in addition, you can also change them in larger steps to move ahead or backward in arbitrarily large jumps. Working with such *random-access* iterators relies on a syntax similar to what you would use in an ordinary `for` loop. For example, assuming that `vec` is an object of type `std::vector<double>`, the following statements are identical in effect

```
for(int i = 0; i < vec.size(); i += 2) { ... },
```

and

```
for(std::vector<double>::iterator it = vec.begin(); i != vec.end(); it += 2) { ... }
```

Many programmers prefer the first method and rely on the random-access operator (`vec[i]`) rather than on random-access iterators to retrieve individual vector elements, but the second method is - at least on paper - a tiny bit faster (in practise, modern compilers will eliminate the difference by optimisation).

***ET* on iterator access rights**

Besides knowing about the *access modes* of iterators (random, directional or forward), it is important to also be aware of the *access rights* of iterators; this is because it is allowed to assign a value to a dereferenced iterator. This operation is, in fact, the same as assigning a value to the element that the iterator is pointing to. In other words, iterators give a programmer a kind of back-door access to the elements of a container, which can then be altered in any way s/he likes. To make this concrete, imagine that you have a `std::vector<double> vec` that is initialised with 10 copies of the value `0.0` and an iterator `std::vector<double>::iterator it` pointing to `vec.begin()`. Then writing something like `*(it + 2) = 1.0;` will have exactly the same result as writing `vec[2] = 1.0;`, as you can easily verify by executing the code in listing 14.6. The message to take home from this is that regular iterators offer both read and write access to your data.

***Nefertiti* on `const_iterators`**

Similar to how you use the keyword **`const`** to control the behaviour of reference parameters, there is a simple way to restrict the access rights of an iterator to *read-only*. Doing so is advisable in all cases where full access is not critical. To create an iterator with *read-only* access rights to the data, simply declare the iterator as a `const_iterator`. There are special version of `begin()` and `end()` that return a `const_iterator` to the start and end of a container. These are called `cbegin()` and `cend()`. Lines 16 and 19 of listing 14.6 provides an illustration.

Code listing 14.6: Access to data via the back door, and how to prevent it

```
1  #include <iostream>
2  #include <vector>
3
4  int main()
5  {
6      std::vector<double> vec(10, 0.0);
7      std::vector<double>::iterator it = vec.begin();
8      std::cout << "vec[2] = " << vec[2] << '\n';
9
10     // change the value of the second element via the 'back door', using iterator
11     *(it + 2) = 1.0;
12
13     std::cout << "vec[2] = " << vec[2] << '\n';
14
15     // to prevent write access, work with const iterators:
16     std::vector<double>::const_iterator cIt = vec.cbegin();
17     *(cIt + 3) = 5.0;    // write access through 'back door' : NOT allowed -> triggers fatal compiler error
18     vec[3] = 5.0;        // write access through 'front door' : perfectly fine
19     for(; cIt < vec.cend(); ++cIt)
20         std::cout << *cIt << '\n'; // no issues when de-referencing a const iterator to read data
21
22     return 0;
23 }
```

Back to `std::list`

Iterators are not only used in loop statements, but also by many STL algorithms and member functions of STL containers, including functions like `insert()`, `erase()` and `splice()` for manipulating `std::lists`. Some of them are demonstrated in code listing 14.7, which reveals the top secret recipe for Jamie's renowned sorted alphabet letter soup with *al dente* vowels. This program also introduces `advance()` (line 28), a helpful function for making a sequential iterator jump multiple steps forward (or backward, if

you specify its second argument as a negative offset). Finally, the program offers a sneak preview of an application of `std::set`, STL's container type for storing collections of unique elements that can be searched efficiently to determine whether a given value is present or not (lines 13 and 18).

Code listing 14.7: `std::list`

```
1  #include <iostream>
2  #include <set>
3  #include <list>
4
5  int main()
6  {
7      // this program implements Jamie's recipe for sorted alphabet letter soup with 'al dente' vowels
8
9      // step 1 : open a can of tomato soup and add some alphabet soup letters
10     std::list<char> letterSoup {'s', 'o', 'u', 'p', 'l', 'e', 't', 't', 'e', 'r', 's'};
11
12     // step 2 : bring to a boil, and immediately scoop out the vowels with a spoon
13     std::set<char> vowels {'a', 'e', 'i', 'o', 'u', 'y'};
14     std::list<char> spoon;
15     for(std::list<char>::iterator it = letterSoup.begin(); it != letterSoup.end();) {
16
17         // search for letter in list of vowels
18         if(vowels.find(*it) != vowels.end()) {
19             // if found, scoop it out of the soup, using spoon
20             spoon.push_back(*it);
21             it = letterSoup.erase(it);
22         }
23         else ++it; // otherwise, consider next letter
24     }
25
26     // step 3 : after boiling for a while, locate the third letter of the soup and add the vowels back in there
27     std::list<char>::iterator it = letterSoup.begin();
28     advance(it, 3);
29     letterSoup.splice(it, spoon);
30
31     // step 4: carefully stir until letters appear in alphabetic order, and serve
32     letterSoup.sort();
33
34     return 0;
35 }
```

To help you follow what's going on, here's the output that you would see if you wrote the state of the soup to the screen after each step of the recipe:

```
step 1: s o u p l e t t e r s
step 2: s p l t t r s
step 3: s p l o u e e t t r s
step 4: e e l o p r s s t t u
```

Getting to know the other container types

`std::vector` and `std::list` are many programmer's favourite workhorses for solving the majority of coding problems. Once you learn to work with them, replacing them by any of the other STL containers is not so difficult anymore, because they are using similar syntax. Of course, you must be aware that the different containers have some different functionalities, corresponding to their intended use: for example, `std::set` has a member function `find()`, because it is designed for efficient element search, but `std::list` does not. Conversely, `std::list` has a function `sort()` specifically designed for the sorting of linked lists, but `std::set` does not need one, because its elements are anyhow kept in sorted order. Random-access containers, like `std::vector`, also do not have their own `sort()` function, not because sorting a vector is not useful, but because they can readily be sorted by general-purpose sorting algorithms available in the STL library `<algorithm>`. Clearly, as a novice programmer, you can hardly expect to know all these things from the start, so the best attitude is to explore, to experiment, and to rely on the working hypothesis that whenever there is something weird about the STL, there is a good reason for it that you will discover

as you become more experienced.

To get you started, here is a couple of brief examples. The first one shows an application of `std::stack`, the STL implementation of a *first-in-last-out* container. There is really not that much that you can do with a `std::stack` (or its *first-in-first-out* counterpart `std::queue`), other than adding elements using `push()`, removing them using `pop()`, and retrieving the value on top of the stack using `top()`.

```
1  #include <iostream>
2  #include <stack>
3
4  int main()
5  {
6      // adding characters from a string on top of a stack one by one:
7      std::string str = "desserts";
8      std::stack<char> S;
9      for(int i = 0; i < str.length(); ++i) S.push(str[i]);
10
11     // taking the elements out again and printing them to screen
12     while(!S.empty()) {
13         std::cout << S.top();
14         S.pop();
15     }
16     return 0;
17 }
```

Because the `top()` element of a stack is the one that was added last, what the program does is that it prints `stressed` (which is `"desserts"` written backwards). The next example shows that is relatively easy to replace the `std::stack` by a `std::queue`. All that is needed are small changes on line 2 and 8 (replacing `stack` by `queue`), and a small change on line 13 to replace `top()` by `front()` (the change in the name of the variable from `S` to `Q` is optional). However, if you get the point of how a queue works, you will quickly realise that the whole exercise is a waste of time: the resulting program is just a very complicated way of writing the original string to the screen, because the `front()` item of a `std::queue` corresponds to the value that was added first of all the other values in the container.

```
1  #include <iostream>
2  #include <queue>
3
4  int main()
5  {
6      // adding characters from a string at the end of a queue one by one:
7      std::string str = "desserts";
8      std::queue<char> Q;
9      for(int i = 0; i < str.length(); ++i) Q.push(str[i]);
10
11     // taking the elements out again and printing them to screen
12     while(!Q.empty()) {
13         std::cout << Q.front();
14         Q.pop();
15     }
16     return 0;
17 }
```

The last two container types that we will discuss are `std::set` and `std::map`. As you have already seen, sets are used for storing unique elements. You can add elements to a `std::set` (at initialisation or using `insert()`), take them out again (using `erase()` and an iterator pointing towards the element that needs to be removed), and check whether a certain element is present already (using `find()` or `count()`). Attempting to add an already existing element has no effect.

Sets have one important restriction: once an element has been added it cannot be changed any more. This is because the elements are stored according to a strict order, which is the secret to success for being able to look up elements fast. If you would be allowed to modify items, elements would potentially have to be resorted every time, which would make the container very inefficient. The next example program illustrates some of these features. If you look at the code carefully, you will see that the iterator variables

declared on line 7 and 30 are `const_iterators`, rather than normal ones. This makes explicit that the iterators have read-access to the set elements only.

```
1  #include <iostream>
2  #include <set>
3
4  void checkVocabulary(const std::string &word, const std::set<std::string> &vocabulary)
5  {
6      std::cout << "Is \"" << word << "\" found in the vocabulary? - ";
7      // search for the word using find()
8      std::set<std::string>::const_iterator it = vocabulary.find(word);
9      // if find returned vocabulary.end(), this means the word was not found
10     std::cout << (it != vocabulary.end() ? "yes" : "no") << '\n';
11 }
12
13 int main()
14 {
15     // initialising a set of strings with a couple of words
16     std::set<std::string> vocabulary {"the", "quick", "fox", "jumps", "over", "the", "lazy",
       "fence" };
17
18     // elements that already belong to a set cannot be added twice
19     std::cout << "The word \"the\" occurs in the vocabulary " << vocabulary.count("the") << "
       times\n";
20
21     // add an element to the set
22     vocabulary.insert("brown");
23
24     // check whether a word occurs in the vocabulary
25     checkVocabulary("cat", vocabulary);
26     checkVocabulary("fox", vocabulary);
27
28     // print all words in the vocabulary
29     std::cout << "\nCurrent words in the vocabulary: ";
30     for(std::set<std::string>::const_iterator it = vocabulary.begin(); it != vocabulary.end();
       ++it)
31         std::cout  << *it << ' ';
32
33     return 0;
34 }
```

The first line of the output of the program confirms that "the" has indeed been added only once, although it occurs twice in the initialiser list. Furthermore, the listing of the full vocabulary suggests that the elements have internally been stored in alphabetic order:

```
The word "the" occurs in the vocabulary 1 times
Is "cat" found in the vocabulary? - no
Is "fox" found in the vocabulary? - yes

Current words in the vocabulary: brown fence fox jumps lazy over quick the
```

At first sight, the container `std::map` appears to be quite different from a `std::set`, because it contains records consisting of a key and value pair. However, the records are stored in a similar way as in a `std::set`: every key can appear only once, and the records are internally sorted with respect to their keys, so that the values can be looked up very quickly based on the keys. The syntax for element retrieval is similar to the familiar access to elements of a `std:vector` object, except that you are using the element access operator `[ ]` with key values instead of integer indices.

The textbook example for illustrating `std::map` is a phone directory, like you have coded earlier for the members of the PGCPPPHD (Pan-Galactic C++ Programming Help Desk; see Exercise 11.3). The two listings below show the original solution to Exercise 11.3 on the left (slightly modified to fit within the column), next to an alternative solution that employs `std::map`. In the solution on the right, we use names as keys, and phone numbers as values, because we want to be able to quickly find a number for a given person.

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <vector>
#include <string>

int main()
{
    // open file
    std::ifstream ifs("phonedirectory.txt");
    if(!ifs.is_open()) {
        std::cerr << "error: unable to open
    phonedirectory.txt\n";
        exit(EXIT_FAILURE);
    }

    struct PhoneRecord {
        std::string mstrName;
        int miExtension;
    };

    // read data from file
    std::vector<PhoneRecord> dir;
    for(;;) {
        PhoneRecord tmp;
        ifs >> tmp.mstrName;
        if(ifs.fail()) break;
        ifs >> tmp.miExtension;
        dir.push_back(tmp);
    }


    // allow user to search directory
    for(;;) {
        std::cout << "enter name"
            << "(or stop to quit).\n> ";
        std::string strInput;
        std::cin >> strInput;

        // terminate loop
        if(strInput == "stop") break;

        // attempt to find name
        int i;
        for(i = 0; i < dir.size(); ++i)
            if(dir[i].mstrName == strInput) {
                std::cout << "please reach "
                    << dir[i].mstrName
                    << " at: "
                    << dir[i].miExtension
                    << "\n\n";
                break;
            }
        if(i == dir.size())
            std::cout << "sorry, who is"
                      << strInput << "?\n\n";
    }
    return 0;
}
```

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <map>
#include <string>

int main()
{
    // open file
    std::ifstream ifs("phonedirectory.txt");
    if(!ifs.is_open()) {
        std::cerr << "error: unable to open
    phonedirectory.txt\n";
        exit(EXIT_FAILURE);
    }









    // read data from file
    std::map<std::string, int> dir;
    for(;;) {
        std::string name;
        ifs >> name;
        if(ifs.fail()) break;
        int extension;
        ifs >> extension;
        dir[name] = extension;
    }

    // allow user to search directory
    for(;;) {
        std::cout << "enter name"
            << "(or stop to quit).\n> ";
        std::string strInput;
        std::cin >> strInput;

        // terminate loop
        if(strInput == "stop") break;

        // attempt to find name
        std::map<std::string, int>::
            const_iterator it =
                dir.find(strInput);
        // check if name was found
        if(it == dir.cend())
            std::cout << "sorry, who is "
                      << strInput << "?\n\n";
        else
            std::cout << "please reach "
                      << strInput << " at: "
                      << dir[strInput]
                      << "\n\n";

    }
    return 0;
}
```

Note that the most striking differences are on:

line 16-19 Since `std::maps` already store data as pairs of key and value, there is no need to declare our own **`struct`** for representing phone records anymore.

line 22-30 In the new implementation, name and extension are stored temporarily in two variables. The statement on line 29 creates a new item in the phone directory. Note that the element access operator `[ ]` is used as if `name` were an integer index value (whereas it is actually a `std::string`).

line 42-55 Instead of browsing through the directory from start to end to look up a given name, the new implementation on the right employs the efficient `find()` member function of `std::map` (line 45). If a name is not found in the directory, `find()` will return the value of `dir.end()`.

The new implementation on the right is much more efficient than the one on the left, and this difference becomes very noticeable when you have to search elements in large datasets. However, the code on the right-hand side can still be optimised a little bit more: on line 53, we write the value of `dir[strInput]` to the screen, but by using the member access operator `[ ]`, the program will search through the `std::map` once more to find the relevant record, even though we already know exactly where it is located. In fact, we can retrieve the key/value pair of interest immediately by dereferencing the iterator `it`, which was initialised by the call to `find()` on line 45. Key and value can next be written to the screen separately, by retrieving the familiar elements of the `std::pair`, `first` and `second`. In other words, we can replace the statements on lines 51–54 by:

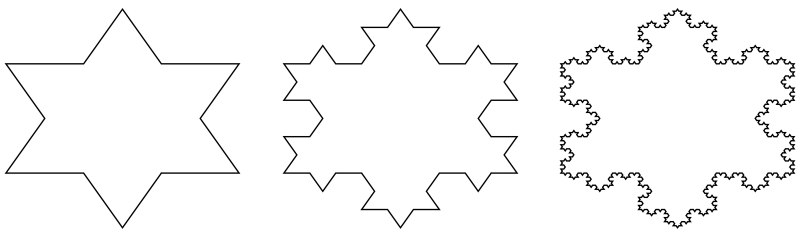
```
1 std::cout << "please reach "
2                 << (*it).first << " at: "
3                 << (*it).second
4                 << "\n\n";
```

It is important not to forget the brackets around `*it`, because dereferencing has lower priority than member access. In practise, you do not have to worry about this detail a lot, because the combination of dereferencing and member access occurs so often, that C^{++} provides a special operator for it. It is denoted as `->`, which has the added benefit of being easier to interpret than the confusing notation used above. Using the new operator for access to a class member via an iterator, lines 51–54 of the original program are written as follows:

```
1 std::cout << "please reach "
2                 << it->first << " at: "
3                 << it->second
4                 << "\n\n";
```

Three tricks that will make your life easier

Because the STL container classes are template classes, you can use them to build up a large variety of nested compound data structures, but dealing with the required template and iterator notation can be daunting. Consider for example the program in Code listing 14.8, which produces a list of (x, y) coordinates that define a closed curve resembling a snowflake (Fig. 14.3).



(a) after 1 iteration (b) after 2 iterations (c) after 4 iterations

Fig. 14.3: The program of listing 14.8 relies on an iterative process to produce self-similar curves that converge to a fractal known as the Koch snowflake. The process starts from an equilateral triangle. In the first iteration, each of its line segments is divided into three parts, and the middle one of these parts is replaced by two line segments pointing outwards. The resulting curve is shown on the left. In the second iteration, the same update rule is applied to all of the line segments once more, and so on for subsequent iterations.

The snowflake curve is constructed by replacing each of the line segments of an equilateral triangle by a pattern of four smaller line segments, and recursively applying the same rule over and over again on the resulting curves. If the snowflake curve is represented by the coordinates of its corner points, then these construction steps can be implemented by splicing new corner points between existing ones, an operation that is accomplished most efficiently using a linked list. It makes sense to store the 2D coordinates of each individual corner point in a `std::pair<double, double>`, so the data structure that would be needed to store a snowflake curve would be a `std::list<std::pair<double, double> >`.

Code listing 14.8 illustrates how Master Yoda uses this data structure to elegantly solved the programming problem, cleverly exploiting the member function `splice()` of `std::list` (see line 61). However, the process of looping over the corner points in the different list objects requires awkward notation to declare the necessary iterators (see, e.g., lines 27-28, 56-57 and 70-71), and Master Yoda had to type "`std::pair<double, double>`" and "`std::list<std::pair<double, double> >`" so many times that he almost lost his cool.

Code listing 14.8: Awkward STL notation

```
1  #include <cstdlib>
2  #include <cmath>
3  #include <iostream>
4  #include <fstream>
5  #include <utility> // for std::pair
6  #include <list>
7
8  // interior points subdividing a unit line segment in the construction of a Koch curve:
9  const std::list<std::pair<double, double> > interiorPoints
10     {{1.0 / 3.0, 0.0}, {0.5, sqrt(3.0) / 6.0}, {2.0 / 3.0, 0.0}};
11
12
13 std::list<std::pair<double, double> > mapInteriorPoints(const std::pair<double, double> &A,
14                                                          const std::pair<double, double> &B)
15 // returns the interior points subdividing the line segment from point A to
16 // point B in one step of the iterative construction of the Koch snowflake curve
17 {
18     // compute length and orientation of line segment
19     const double dx = B.first - A.first;
20     const double dy = B.second - A.second;
21     const double L = sqrt(dx * dx + dy * dy);
22     const double cosTheta = dx / L;
23     const double sinTheta = dy / L;
24
25     // construct segment of Koch curve
26     std::list<std::pair<double, double> > segment;
27     for(std::list<std::pair<double, double> >::const_iterator it = interiorPoints.cbegin();
28         it != interiorPoints.cend(); ++it) {
29
30         // retrieve relative coordinates of the point
31         std::pair<double, double> U = *it;
32
33         // compute absolute coordinates by rotation, scaling and translation
34         std::pair<double, double> V = A;
35         V.first  += L * (cosTheta * U.first - sinTheta * U.second);
36         V.second += L * (sinTheta * U.first + cosTheta * U.second);
37
38         // add coordinate to line segment
39         segment.push_back(V);
40     }
41     return segment;
42 }
43
44
45 int main()
46 {
47     // define initial curve
48     std::list<std::pair<double, double> > snowflake
```

```
        {{0.0, 0.0}, {0.5, 0.5 * sqrt(3.0)}, {1.0, 0.0}};

    const int maxIt = 4;
    for(int i = 0; i < maxIt; ++i) {

        // loop through all line segments, starting from the one connecting the last to the first point
        std::pair<double, double> A = snowflake.back();
        for(std::list<std::pair<double, double> >::const_iterator itB = snowflake.cbegin();
            itB != snowflake.cend(); ++itB) {

            // apply construction rule of the Koch curve to each segment:
            // compute interior points between point A and point B and insert them before point B
            snowflake.splice(itB, mapInteriorPoints(A, *itB));

            // point B becomes point A of the next line segment
            A = *itB;
        }
    }

    // write coordinates to file
    std::ofstream ofs("koch_curve.csv");
    for(std::list<std::pair<double, double> >::const_iterator it = snowflake.cbegin();
        it != snowflake.cend(); ++it)
        ofs << it->first << ',' << it->second << '\n';

    return 0;
}
```

Fortunately, there are several ways to avoid much of the awkward STL notation in listing 14.8. The first method is to introduce shorthand notation for the type names, using the keywords **typedef** or **using**. For example, we can write **typedef** `std::pair<double, double> Point;` or **using** `Point = std::pair<double, double>;` to introduce `Point` as a *type alias* that you can use from there on to refer to the type `std::pair<double, double>` by another, simpler name.

Type aliases can be nested as well, i.e., after defining `Point`, we can write **typedef** `std::list<Point> Curve;` or **using** `Curve = std::list<Point>;` to create a type alias for `std::list<std::pair<double, double> >`. Moreover, once `Curve` has been introduced, we can write **typedef** `Curve::const_iterator CurvecCIt;` or **using** `CurveCIt = Curve::const_iterator;` to introduce `CurveCIt` as shorthand notation for `std::list<std::pair<double, double> >::const_iterator`. Below, you can see, for a small piece of listing 14.8, how using these type aliases really helps to make things look simpler:

```
#include ...

using Point = std::pair<double, double>;
using Curve = std::list<Point>;
using CurveCIt = Curve::const_iterator;

// interior points subdividing a unit line segment in the construction of a Koch curve:
const Curve interiorPoints {{1.0 / 3.0, 0.0}, {0.5, sqrt(3.0) / 6.0}, {2.0 / 3.0, 0.0}};

Curve mapInteriorPoints(const Point &A, const Point &B)
{
   ...

    // construct segment of Koch curve
    Curve segment;
    for(CurveCIt it = interiorPoints.cbegin(); it != interiorPoints.cend(); ++it) {

        // retrieve relative coordinates of the point
        Point U = *it;

        // compute absolute coordinates by rotation, scaling and translation
        Point V = A;
        V.first  += L * (cosTheta * U.first - sinTheta * U.second);
        V.second += L * (sinTheta * U.first + cosTheta * U.second);
```

```
25
26          // add coordinate to line segment
27          segment.push_back(V);
28      }
29      return segment;
30  }
```

A second trick that can simplify your life as a STL programmer is to rely on the compiler to infer the type of complex variables automatically. You can request it to do so by using the keyword **auto**, which functions as a placeholder for a type name that is to be derived by the compiler. For example, an alternative way to code the **for**-loop on line 16 above is:

```
16      for(auto it = interiorPoints.cbegin(); it != interiorPoints.cend(); ++it) {
17          ...
```

In this example, the variable `it` is initialised by a value that is returned by the `cbegin()` member function of `interiorPoints`, which is a `std::list` of `std::pair<double, double>` objects. By using **auto** we are telling the compiler to use all of this information to determine that the type of `it` must be a `const_iterator` of a `std::list<std::pair<double, double> >`.

Trick number three helps to further simplify **for** loops, specifically those that loop over all elements of a container and apply the same operation to all of them. In modern editions of C++, you can code such loops by means of a so-called range-based **for** loop, which lack the explicit control expressions of the traditional **for** loop. The following code illustrates the syntax of a range-based for loop, and its application to write all values in a `std::vector` to the screen:

```
1       std::vector<int> vec;
2       ...
3       for(int x : vec) {
4           std::cout << x << '\n';
5       }
```

The range-based **for**-loop does not have a loop counter, but instead it introduces a variable (specified before the colon) that will take the values of consecutive elements of a container (specified after the colon). This variable relates to the container elements in the same way as we have seen for function arguments and the variables that are passed to a function. The default behaviour of the range-based **for**-loop is that the values of the container elements are copied, so if we were to change the variable `x` in the body of the loop above, that would have no effect on the corresponding element of the container `vec`. Therefore, if we need to have read access to the container, it is necessary to pass the container elements *by reference*. This can be done as follows:

```
1       std::vector<int> vec;
2       ...
3       for(int &x : vec) {
4           ++x;
5       }
```

Now, the variable `x` of the range-based **for**-loop will behave as a reference to consecutive elements of the container `vec`. Therefore, the loop will cause all the elements of `vec` to be incremented.

Like for function arguments, reference variables are also used in a range-based **for**-loop if we want to avoid unnecessary copying. But if write access to the container is not needed, it is best to declare the references as **const**. Below is how we can apply this approach to rewrite the first **for**-loop of code listing **??**:

The **auto** type placeholder can be combined with type modifiers like **const** or `&` to create constant variables of the inferred type or reference parameter

14.4 Exercises

Code listing 14.9: The Towers of Hanoi

```
#include <iostream>
#include <cstdlib>
#include <exception>
#include <map>
#include <stack>

void move(std::map<char, std::stack<int> > &towers, const char source, const char destination)
// moves the top disk from source to destination stack
{
    static int step = 0;

    // remove disk from source stack
    int disk  = towers[source].top();
    towers[source].pop();

    // verify that the move is valid
    if(!(towers[destination].empty() || disk < towers[destination].top()))
        throw std::logic_error("sorry, that move is not allowed\n");

    // add disk to destination stack
    towers[destination].push(disk);

    // inform the user after each move
    ++step;
    std::cout << "step " << step << " :  Move disk " << disk << " from pile " << source << " to
    " << destination << '\n';
}

void solve(std::map<char, std::stack<int> > &towers, const int n, const char source, const char
    destination, const char aux)
// moves the n smallest rings from source to destination pile, using aux as an auxiliary stack
{
    if(n == 1) {
        // move smallest disk from source to destination pile
        move(towers, source, destination);
    }
    else {
        // use auxiliary stack to temporarily store n - 1 smallest disks
        solve(towers, n - 1, source, aux, destination);

        // move n-th largest disk from source to destination pile
        move(towers, source, destination);

        // move the smaller disks from the auxiliary stack back to the destination pile
        solve(towers, n - 1, aux, destination, source);
    }
}


int main()
{
    try {
        // This program solves the Towers of Hanoi problem for d disks
        const int d = 3;

        // create a std::map representing piles holding a stack of disks
        // each pile is identified by a char key 'A', 'B', or 'C'
        std::map<char, std::stack<int> > towers;

        // initialise piles with no disks (three empty stacks)
        towers['A'] = towers['B'] = towers['C'] = std::stack<int>();

        // create initial stack of disks at pile A
        for(int disk = d; disk > 0; --disk) towers['A'].push(disk);
```

```
63
64          // call recursive solution algorithm to move all disks from A to B
65          solve(towers, d, 'A', 'B', 'C');
66      }
67      catch(std::exception &err)
68      {
69          std::cerr << err.what();
70          exit(EXIT_FAILURE);
71      }
72      return EXIT_SUCCESS;
73  }
```

Module 15

Designing algorithms

Jamie explains what's on the menu

In this module you will:

- Design algorithms using flow-charts and pseudo-code.
- Be introduced to different strategies for detecting and handling errors.
- Implement simulations of matrix population models and other deterministic discrete-time dynamical systems.

15.1 Algorithms

Despite ongoing developments in the field of Artificial Intelligence, we cannot yet expect computers to come up with new and original ways to solve problems all by themselves. On the contrary, present-day computers must be told exactly what they have to do. Before solving any problem with the use of a computer program in any programming language, it is therefore necessary to design an algorithm that the computer program can follow in order to obtain a desired output from a set of program inputs.

Designing a functional algorithm is just as important as being able to implement that algorithm in a programming language like C++. So far, we have focused on developing coding skills, but in this Module we will turn our attention to the algorithm design phase of a programming project. An algorithm is a step-by-step procedure for calculations. For example, an algorithm to brush your teeth might be the following:

1. Put tooth paste on your tooth brush
2. Water the tooth paste on your brush
3. Brush your teeth
4. Clean the tooth paste from the brush
5. Rinse your mouth

What about computer algorithms? As a first example, assume that your task is to find the largest number in a list of numbers. One option is to use the Standard Template Library function `sort()` to sort the numbers in ascending order and take the final number in the sorted list. This approach is not

very efficient however, since it not necessary to sort all the numbers in the list. A more straightforward alternative is to locate the maximum in the following way:

1. Store the value of the first item in the list in variable x
2. Examine each of the remaining items $y_1, y_2, \ldots$ in the list. If $y_i > x$ assign the value of y_i to x. Next, move on to the next item in the list.
3. When the end of the list is reached, the value of x is equal to the largest value in the list.

Listing 15.1 illustrates how this algorithm is implemented in C^{++}.

Code listing 15.1: Finding the largest element in a vector

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

double findLargest(const std::vector<double> &vec)
// returns largest element in vec, assuming that vec is not empty
{
    double max = vec[0];
    for(int i = 0; i < vec.size(); ++i)
        if(vec[i] > max)
            max = vec[i];
    return max;
}

int main()
{
    std::vector<double> vec(10, 0.0);
    vec[4] = 1.7;

    std::cout << findLargest(vec);

    return 0;
}
```

A problem that is a bit more complicated is to find the greatest common divisor of two non-prime numbers A and B. The Greek mathematician Euclid proposed an elegant algorithm to solve this problem as Proposition II in Book VII (*Elementary Number Theory*) of his famous *Elements*. The algorithm, which was implemented in Listing 6.1 is the following:

1. Input two integers, A and B
2. If $B = 0$ then go to step 5
3. If $A > B$, let $A = A - B$; otherwise let $B = B - A$
4. Go back to step 2
5. Print A

Written descriptions of algorithms are often presented in *pseudocode*, an artificial and informal language that is somewhere midway between plain English and computer code. Algorithms can also be depicted graphically, by means of flowcharts. Figure 15.1 shows a flowchart for Euclid's algorithm, and Algorithm 1 is its representation in pseudocode. Flowcharts and pseudocode are useful tools when you need to explain a computational model to someone who is not familiar with the programming language you used for the implementation. Moreover, they help you keep an overview while you are developing code. Therefore, designing a flowchart or pseudocode representation of your program before you start coding is a very good idea. Initially, take the time to develop your ideas on a piece of paper, and draw out how your envisioned algorithm operates on the variables in your program. With some practice you will be able to accomplish some of these steps mentally but, even then, pen and paper can be indispensable to help guide your intuition through tricky parts of the programming process. In any case, resist the temptation

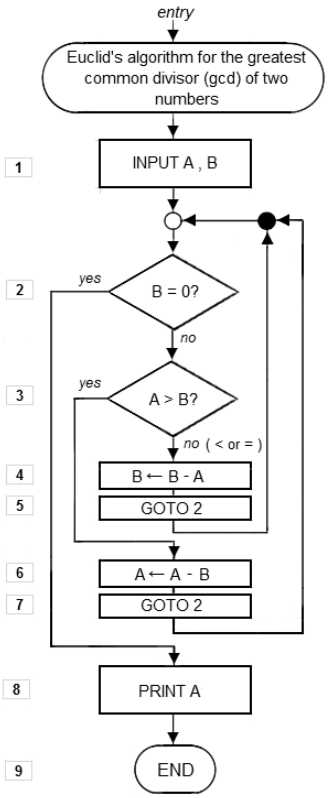


Fig. 15.1: Flowchart of Euclid's algorithm for finding the greatest common divisor of two numbers A and B. Image source: http://en.wikipedia.org/wiki/File:Euclid_flowchart_1.png

Data: integers A and B
Result: greatest common divisor of A and B
while $B \neq 0$ **do**
  if $A > B$ **then**
    $A \leftarrow A - B$
  else
    $B \leftarrow B - A$
  end
end
return A;

Algorithm 1: Euclid's algorithm for finding the greatest common divisor of two numbers

to start coding without a clear plan: it will be very hard to fix initial design mistakes at a later stage, because some pieces of code will probably depend on the part you need to fix.

In subsequent Modules, we will look in more detail at different kinds of algorithms for simulating biological processes. For example, a simplified algorithm for studying the change of gene frequencies as a result of selection, drift and recombination is the following:

1. Initialise a population of individuals expressing a set of heritable characters that affect reproductive success and/or survival at some point of the life cycle
2. For each individual in the population:
   a Determine the phenotype of the individual as a function of its genotype
   b Simulate the life history of the individual, taking into account its phenotypic characteristics
   c Assess the individual's fitness at the end of its life
3. Randomly select individuals from the population based on their relative fitness
4. Create a new generation of offspring individuals by copying or mating the selected individuals from the parental generation
5. Mutate the progeny
6. Replace the parental generation by the offspring
7. Return to step 2 until the desired number of generations has been reached.

This description of an evolutionary algorithm is still far from an actual implementation in a computer program, because every single step of the procedure contains an algorithm itself. For example, the sampling step (3), will require the implementation of a weighted lottery, which in turn, will rely on algorithms to generate pseudo-random numbers. Accordingly, a fruitful general strategy for program design is to recursively split a big problem into a number of small sub-problems, until you arrive at a sufficiently detailed level of specification that you can start translating your ideas into computer code. The elementary sub-problems that you end up with will quite often be very different in nature from the main problem. Moreover you may encounter some problems, like finding the largest value in a list, sorting an array, or implementing a weighted lottery, again and again in very different contexts. Oftentimes, efficient algorithms are available for solving these general programming problems, which you can then use as building blocks for your own program.

15.2 Debugging and error handling

An unfortunate but unavoidable aspect of programming is the process of debugging, i.e., tracking down the errors ('bugs') that interfere with the successful execution of a program. There can be several different kinds of bugs in your program. Some bugs prevent successful compilation or linking. These so-called compile-time errors are often easy to find: your compiler will generate an error message that allows you to locate and correct the error. Some other bugs appear only at run-time, i.e., when the program is executed. These bugs are the nasty ones. Even when they have obvious effects, such as causing your computer to crash, it can be hard to find out which part of the code is responsible for the problem. Other bugs may be easy to fix once you know they occur, but may go completely unnoticed unless you examine the program output very carefully. The worst kind of bug, of course, is the one that is both difficult to discover and difficult to fix. That such bugs will be present in your program is not at all unlikely: even professional programmers have, on average, one bug in every 200 lines of code. For this reason, it is necessary to incorporate error checking and error handling into the design of your algorithms, in order to create code that is maximally robust against potential programming mistakes.

The `assert` statement

The hunt for bugs already starts when your program is still under construction. During this phase, errors frequently arise when one part of your code produces unanticipated results that cause another part of the code to go wrong. For example, in a program that calculates the harmonic mean of a set of n positive values, $\bar{x}_{\mathrm{H}} = n/(\sum_{i=0}^{n} x_i^{-1})$, you may have relied on the assumption that all values were strictly positive. However, floating-point underflow errors may occasionally and unexpectedly produce a value that is equal to zero, so that the calculation of the harmonic mean causes a division-by-zero error. The `assert` statement, defined in the library `cassert` provides a way to detect such a mistake. Code listing 15.2 shows how the `assert` statement is applied in the context of a function `calculateHarmonicMean`. If, by accident, the function is called with an empty vector as argument, the following message is written to the error stream:

```
Assertion failed: (!vec.empty()), function calculateHarmonicMean,
file /Users/Yoda/CppCourse/main.cpp, line 6.
```

Likewise, if the `std::vector` that is passed to the function accidentally contains a value that is not strictly positive, a similar error message appears:

```
Assertion failed: (xi > 0.0), function calculateHarmonicMean,
file /Users/Yoda/CppCourse/main.cpp, line 10.
```

You can see that the error messages were triggered by the `assert` statements on line 6 and 10, respectively, which behave as a special kind of conditional statement: if the argument of `assert` evaluates to **`true`**, nothing happens, but if it evaluates to **`false`**, the failed assertion is reported and the program is terminated. In other words, the `assert` statement provides a way to make sure that a critical assumption, needed for your algorithm to function correctly, is valid.

Code listing 15.2: Verifying assumptions using the `assert` statement

```
1  #include <cassert>
2  #include <vector>
3
4  double calculateHarmonicMean(const std::vector<double> &vec)
5  {
6      assert(!vec.empty());
7      double dHarmonicMean = 0.0;
8      for(int i = 0; i < vec.size(); ++i) {
9          const double xi = vec[i];
10         assert(xi > 0.0);
11         dHarmonicMean += 1.0 / xi;
12     }
13     return vec.size() / dHarmonicMean;
14 }
```

The precise message produced by a failed assertion differs between compilers. For example, some versions of *Visual Studio* are more considerate (or verbose, depending on your taste):

```
Assertion failed: (xi > 0.0), file /Users/Nefertiti/CppCourse/main.cpp, line 10.

This application has requested the Runtime to terminate it in an unusual way.
Please contact the application's support team for more information.
```

Considering that the application's support team is no one other than yourself, the seemingly helpful suggestion at the end can feel a bit cynical at times when you are desperately trying to identify the cause of a failed assertion. However, even this error message points you to the exact position of the `assert` that triggered program termination, so that you at least can start looking for a solution to the problem at the right spot. Another nice feature is that `assert` statements can be automatically cut out of your code by the preprocessor when you are confident that your program works as intended. To do this, just include the pre-processor instruction **`#define`** `NDEBUG` before the instruction **`#include`** `<cassert>`. For example, the assertion `assert (2 + 2 == 5)` on line 7 of the following program will not produce an error message, although the assertion obviously does not hold. This is due to the instruction on line 1, which causes all assert statements to be removed before the program is compiled.

```
1  #define NDEBUG
2  #include <cassert>
3
4  int main()
5  {
6      assert (2 + 2 == 5);
7      // more code...
8      return 0;
9  }
```

It is not recommended to use the `assert` statement for error-handling in the consolidated version of a program, after you have completed developing the algorithm and verified that it functions correctly. There are dedicated methods for error handling during normal program execution that will be introduced at the end of this section.

Debugging tools

IDEs such as *MS Visual Studio* and *XCode* include a debugger, an elaborate tool that can assists you to find and fix bugs of any kind. A debugger commonly keeps track of all program threads, i.e., all branches that are encountered during program execution, and logs when and where which variable is assigned which value. In this way, it enables stepwise program execution so that you can check program performance after each line of code. The debugger also allows you to set checkpoints at strategic places in your program so that you can evaluate your program piece by piece.

The default project setup in *Visual Studio* enables the *Debug* configuration for the project. You can change the project configuration in the *Configuration Manager*, which can be found as an option in the *Build* menu. The *Debug* menu includes the most-often used debugging options, such as setting and removing breakpoints, starting and stopping the debugger, and executing code per line or per function. You can also make these options permanently available by activating the *Debug* toolbar in the *View* menu. The following options are useful to detangle complex run-time errors:

Breakpoints: A breakpoint tells the debugger to stop executing code just before the line where the breakpoint is set. Setting breakpoints can help you analyse large programs section by section.

Start Debugging: This option will start the compiler in the (default) *Debug* configuration. If no breakpoints are set and no run-time errors are encountered, the program will be compiled, linked and executed. When a breakpoint is set the debugger will stop program execution just before the breakpoint and open two output windows at the bottom of your screen. Both windows have several tabs that enable you to view all kinds of information about your program that apply at the moment where program execution was stopped. The tabs *Auto* and *Locals* show the current value of all (local) variables, so that you can check whether your variables have values at all, and if so, whether these values are what you expect them to be. Variables without a value commonly indicate initialisation errors in your program. In case of a program crash due to a run-time error the debugger will display the program status just before the error occurred.

Step Into: This option will let you execute code line by line starting from the breakpoint. When a function is called in a statement, the yellow indicator arrow will jump to the corresponding function definition. Line by line execution of your code generally enables you to see exactly when and where an error occurs.

Step Over: This option allows you to skip line-by-line execution of code within statements and functions. This is very useful to skip the standard C^{++} functions that are also executed line by line when using the previous option.

Step Out: This option allows you to leave the current statement or function and jump back to the calling line to continue line-by-line execution.

This overview is by no means complete; the debugger has many more (advanced) tracking and tracing options that are particularly useful when you start building large, complex programs.

Exception handling

Even algorithms that work well most of time may go wrong under exceptional circumstances, or when a user applies them to problems that the program is not designed to handle. To ensure the reliability and robustness of software, it is critical to detect such situations in order to avoid run-time errors that would otherwise result from them. This kind of error checking, also referred to as *exception handling* is fundamentally different from catching bugs during the development phase (which can be detected with the help of the debugger or the `assert` statement). Exception handling has to be done all the time, under all circumstances, and not only during the development phase. The goal of exception handling is to prevent software that *normally* achieves its intended goal from going wrong when things do not work out as planned. One example of where you need to think of exception handling is when your program attempts to open a file. Under normal circumstances, you can rely on `std::fstream::open()` to open the file correctly, but problems may occur for example when the file does not exist or is already open in another program. To handle such an event, we can terminate the program if the file was not opened correctly by means of a conditional `exit()` construction:

```
std::ifstream ifs("my_file.txt");
if(!ifs.is_open()) {
    std::cerr << "error: unable to open file\n";
    exit(EXIT_FAILURE);
}
```

You may recall that `exit()` provides a controlled way to terminate a program in case of a fatal error. This means that running processes are terminated normally, i.e., by clearing all variables and deallocating all allocated memory. The function takes an integer argument that represents some status code that is returned to the calling process. By convention, a status code of 0 (or `EXIT_SUCCESS`) indicates successful termination, whereas any other number may be used to indicate different types of fatal errors (the default error code is 1, or `EXIT_FAILURE`).

***Arnold* on `abort()`**

The brute-force equivalent of `exit()` is the function `abort()` (also defined in `cstdlib`), which performs an immediate emergency stop of a program. This function terminates a program without any cleaning up, which may result in memory loss because allocated memory is not properly de-allocated. The function `abort()` is invoked automatically by `assert()` after a failed assertion; it returns no status code or other feedback.

15.3 Simulation algorithms for matrix population models and other discrete-time dynamical systems

In following modules, we will examine how algorithm design and exception handling are applied in the context of various kinds of biological simulations. To get started, we will first take a look at discrete-time dynamical models that can be analysed by straightforward iteration. One important application of these models is in population ecology and life-history theory, where they are studied to investigate the dynamics of age- or class-structured populations. While we also focus on this particular biological context here, bear in mind that matrix population models are representative of a much larger class of dynamical systems (including discrete-time Markov chains) that can be analysed by very similar methods.

In matrix population models, the state of a population at time t is represented by a (column) vector $\mathbf{n}_t = (n_1(t), n_2(t), \ldots, n_k(t))$ that contains the densities of individuals in a set of mutually exclusive, discrete states, identified by the indices $1, 2, \ldots, k$. Often, the different states correspond to age-classes, so that $\mathbf{n}_t$ reflects the age-distribution of individuals at time t.

The state of the population at the next time step, $\mathbf{n}_{t+1}$, is obtained by multiplying the vector $\mathbf{n}_t$ by a matrix $\mathbf{L}$ that contains the per-capita fecundities f_i and survival probabilities s_i of individuals in each of the different states. The general form of this so-called Leslie matrix is the following

$$\mathbf{L} = \begin{pmatrix} f_1 & f_2 & f_3 & \ldots & f_k \\ s_1 & 0 & 0 & \ldots & 0 \\ 0 & s_2 & 0 & \ldots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \ldots & 0 & s_{k-1} & 0 \end{pmatrix}$$

That is, the age-specific fecundities f_i form the first row of the matrix, while the probabilities of survival from age class i to age class $i+1$ appear on its sub-diagonal. Assuming that the initial state of the population is known, we can now calculate the state at the next time step by evaluating the recurrence relationship

$$\mathbf{n}_{t+1} = \mathbf{L}\,\mathbf{n}_t$$

Working out the matrix-vector multiplication yields

$$n_1(t+1) = \sum_{i=1}^{k} f_i\, n_i(t)$$
$$n_{i+1}(t+1) = s_i\, n_i(t) \text{ for } 1 \leq i < k$$

In other words, the number of newborn individuals at the next time step is given by the sum of the current densities of individuals in the different states, weighted by their respective fecundities. Moreover, surviving individuals show up in the next age class at the next time step.

It can be shown mathematically that the age distribution will eventually converge to a stable distribution $\mathbf{v}$ and that the total population will eventually grow by a constant factor λ every time-step. Hence, for large t

$$\mathbf{n}_t \sim \lambda^t \mathbf{v}$$

The value of λ is given by the leading eigenvalue of the matrix $\mathbf{L}$, and $\mathbf{v}$ is the corresponding eigenvector. Accordingly, whenever $\lambda \neq 1$, the total size of the population will either shoot off to infinity, or shrink to zero in the long run. While this fact indicates that something essential (i.e., density regulation) is missing from matrix population models, the geometric growth rate λ can still be interpreted as a measure of the ability of a population to increase in numbers when it is introduced in the current environment at a low initial density.

The tendency of matrix population models to show geometric growth is problematic from a computational perspective, because population densities will eventually become so small or so large that the densities can no longer be accurately represented by floating point numbers. Therefore, it is necessary to renormalise the total population at each time step by a factor c_t, that is chosen in such a way that the total population density remains constant. The modified system,

$$\mathbf{n}_{t+1} = \frac{1}{c_t} \mathbf{L}\, \mathbf{n}_t,$$

reaches equilibrium when $\mathbf{n}_{t+1} = \mathbf{n}_t = \mathbf{v}$. In that case, $c_{t+1} = c_t = c^*$ is determined by the equilibrium condition

$$\mathbf{v} = \frac{1}{c^*} \mathbf{L}\, \mathbf{v} = \frac{\lambda}{c^*} \mathbf{v}$$

In other words, once the population has converged on the stable age distribution, the value of the normalisation constant must be equal to the leading eigenvalue of $\mathbf{L}$.

After this brief review of the mathematical background theory, we can now start to design a simulation algorithm for calculating the stable age distribution $\mathbf{v}$ and the geometric growth rate λ. In broad steps, what we need to do is:

1. initialise the matrix $\mathbf{L}$.
2. set the initial state of the population $\mathbf{n}_0$.
3. iterate the equation $\mathbf{n}_{t+1} = \frac{1}{c_t} \mathbf{L}\, \mathbf{n}_t$ until equilibrium has been reached.
4. show the stable age distribution $\mathbf{v}$ and the long-term geometric growth rate λ.

Algorithm 2 shows a more detailed description of these steps in pseudocode. Here, the vector $\mathbf{n}$ is used to store the current state of the population, $\mathbf{v}$ serves as a temporary storage for the next population state, and λ doubles as a variable to store the current renormalisation constant (λ and $\mathbf{v}$ will eventually converge to the dominant eigenvalue and the stable age-distribution, which are exported as return values). In addition, the notion of iterating the population dynamics until convergence to an equilibrium age distribution has been made more concrete: a new variable δ was introduced to quantify the difference between the new and old normalised age distributions at each time step, and the iteration is continued until δ has become smaller than a small positive constant ϵ. Choosing a small value of ϵ will make the final result more precise, until the point where numerical inaccuracies start to interfere with the calculations. However, it will also make the algorithm slower. It takes a bit of coding effort still to translate Algorithm 2 into C^{++}, since several auxiliary functions are needed for initialisation and producing screen output. Yet, as you can see in Listing 15.3, the pseudo-code representation already contains all essential elements of the core of the final C^{++} program.

Code listing 15.3: Numerical analysis of an age-structured population model

```
1 /* This program iterates a Leslie matrix population
2    model and computes the stable age distribution
3    and the long-term geometric growth rate of the
4    population.
5
6    By adjusting the specification of the transition
7    matrix in the function initialise(), the same
```

Data: Leslie matrix $\mathbf{L}$ and precision $0 < \epsilon \ll 1$.
Result: Stable age distribution $\mathbf{v}$ and long-term population growth rate λ.
Set $t = 0$ and initialise population state vector $\mathbf{n}$;
repeat
  Compute $\mathbf{v} = \mathbf{L}\,\mathbf{n}$;
  Set $\lambda = \sum_{i=1}^{k} v_i$ equal to the total density of the updated population;
  Update $\mathbf{v} \leftarrow \mathbf{v}/\lambda$ to normalise the total population density;
  Calculate $\delta = ||\mathbf{v} - \mathbf{n}||$ to quantify change in age distribution;
  Update $\mathbf{n} \leftarrow \mathbf{v}$;
  Increment time $t \leftarrow t + 1$;
until $\delta < \epsilon$;
return λ and $\mathbf{v}$;

Algorithm 2: Numerical analysis of an age-structured population model

```
    code can be used to find the leading eigenvalue
    and corresponding eigenvector for any kind of
     linear discrete -time model

    x_{t+1} = M x_{t}

    for any kind of transition matrix M
    Written on 14-11-2016 by Master Yoda. */

#include <iostream>
#include <iomanip>
#include <vector>
#include <cstdlib>
#include <exception>

//*** iteration algorithm ****************************

void iterateMatrixModel(const std::vector<std::vector<double> > &transitionMatrix,
     std::vector<double> &vecv, double &lambda)
// iterates matrix population model to find the stable age distribution
// (which will be stored in vecv) and the long-term geometric growth
// rate (which will be stored in lambda).
{
    // constants
    const double    epsilon = 1.0e-6;
    const int       itMax = 9999;

    // prepare for iteration
    const int iA = transitionMatrix.size();
    if(iA == 0) // throw exception if number of age classes is zero
        throw std::logic_error("transition matrix is empty.\n");

    std::vector<double> vecn(iA, 1.0 / iA);
    vecv = std::vector<double>(iA);

    // iterate until convergence
    int it;
    for(it = 0; it < itMax; ++it) {
        // execute matrix-vector multiplication to obtain next state
        lambda = 0.0;
        for(int i = 0; i < iA; ++i) {
            vecv[i] = 0.0;
            for(int j = 0; j < iA; ++j)
                vecv[i] += transitionMatrix[i][j] * vecn[j];
            lambda += vecv[i];
        }
        // renormalise next state vector and quantify change in age distribution
        double delta = 0.0;
        for(int i = 0; i < iA; ++i) {
            vecv[i] /= lambda;
```

```
57              double tmp = vecv[i] - vecn[i];
58              delta += tmp * tmp;
59          }
60          // check convergence criterion
61          if(delta < epsilon * epsilon)
62              break;
63          // update population state
64          vecn = vecv;
65      }
66
67      if(it == itMax)
68          throw std::runtime_error("slow convergence in function iterateMatrixModel().\n");
69
70      std::clog << "iteration in function iterateMatrixModel() converged after " << it << "
        steps.\n";
71  }
72
73  //*** main program ***********************************
74
75  int main()
76  {
77      // prototypes of auxiliary functions
78      void initialise(std::vector<std::vector<double> >&);
79      void show(const std::vector<std::vector<double> >&);
80      void show(const std::vector<double>&, const double);
81
82      // initialise transition matrix and show it on screen
83      std::vector<std::vector<double> > transitionMatrix;
84      initialise(transitionMatrix);
85      show(transitionMatrix);
86
87      // calculate stable age distribution and growth rate
88      std::vector<double> stableAgeDistribution;
89      double longTermGeometricGrowthRate;
90      iterateMatrixModel(transitionMatrix, stableAgeDistribution, longTermGeometricGrowthRate);
91
92      // show result
93      show(stableAgeDistribution, longTermGeometricGrowthRate);
94
95      return 0;
96  }
97
98  //*** auxiliary functions ****************************
99
100 void initialise(std::vector<std::vector<double> > &matL)
101 // initialises the transition matrix
102 {
103     // create a matrix of the desired size and initialise with zeroes
104     const int iA = 10; // number of age classes
105     matL = std::vector<std::vector<double> >(iA, std::vector<double>(iA, 0.0));
106
107     // set fecundities (values below are an arbitrary example)
108     matL[0][2] = 0.2;
109     matL[0][3] = 0.4;
110     matL[0][4] = 1.2;
111     matL[0][5] = 1.5;
112     matL[0][6] = 1.5;
113     matL[0][7] = 1.4;
114     matL[0][8] = 1.1;
115     matL[0][9] = 0.6;
116
117     //set survival probabilities (values below are an arbitrary example)
118     matL[1][0] = 0.4;
119     matL[2][1] = 0.9;
120     for(int i = 3; i < iA; ++i)
121         matL[i][i-1] = 0.95 * matL[i - 1][i - 2];
```

```
}

void show(const std::vector<std::vector<double> > &matL)
// shows transition matrix on the screen
{
    std::cout << "transition matrix: \n";
    const int iA = matL.size();
    for(int i = 0; i < iA; ++i) {
        for(int j = 0; j < iA; ++j)
            std::cout << ' ' << std::left << std::setw(6) << std::setprecision(3) <<
    matL[i][j];
        std::cout << '\n';
    }
    std::cout << '\n';
}

void show(const std::vector<double> &vecv, const double lambda)
// shows stable age distribution and growth rate on the screen
{
    std::cout << "\nstable age distribution: \n";
    for(int i = 0; i < vecv.size(); ++i)
        std::cout << ' ' << std::left << std::setw(6) << std::setprecision(3) << vecv[i];
    std::cout << '\n';

    std::cout   << "\nlong-term geometric growth rate: \n"
                << lambda << '\n';
}
```

15.4 Exercises

Data: Interval endpoints a and b; precision $0 < \epsilon \ll 1$.
Result: Value x between a and b such that $f(x) = 0$ for a specified function f.
if $f(a) \cdot f(b) \leq 0$ **then**
  if $f(a) > 0$ **then**
    Swap a and b;
  end
  while $|a - b| > \epsilon$ **do**
    Set $c = (a + b)/2$;
    if $f(c) \leq 0$ **then**
      $a \leftarrow c$;
    else
      $b \leftarrow c$;
    end
  end
  return $(a + b)/2$;
else
  Report error message: "unable to locate root between a and b";
end

Algorithm 3: Bisectioning algorithm for locating the root of a function f in an interval $[a, b]$.

15.1. The bisection algorithm for locating the root of a function. Algorithm 3 is a pseudo-code representation of a basic method for locating the root of a function f in an interval $[a, b]$. If $f(a)$ and $f(b)$ differ in sign, the algorithm will return a value $x \in [a, b]$ such that $f(x) = 0$ (more precisely, the algorithm guarantees that $f(x \pm \epsilon) = 0$ for some small value ϵ, so that the root lies very close to x).

(a) Examine the algorithm and explain how it works.

(b) Implement the algorithm in C^{++}, and apply it to locate the negative root of $f(x) = x^2 - 2$.

(c) Maximise the efficiency of the algorithm by minimising the number of times the function f is evaluated.

(d) Throw exceptions when the function f does not change sign between a and b, or when the number of steps needed to achieve the desired precision exceeds a predefined maximum number of iterations. Ensure that these exceptions are processed by appropriate exception handlers.

15.2. Practise your bug-hunting skills ($\star$). The following program is a simplified version of an individual-based simulation of genetic evolution. The program tracks a population of individuals that carry either a wildtype allele at a haploid locus, or a deleterious mutation. This mutation reduces the probability of survival to $1 - s$. While selection removes individuals carrying the deleterious mutation from the gene pool, mutation introduces new copies of the deleterious allele. In particular, μ is the probability of a mutation from the wildtype to the deleterious allele.

Mutation and survival are stochastic events that are implemented using the random number function `std::rand()` from the library `cstdlib`. This function, which is called on line 19 and 27, returns a random integer between 0 and a large value `RAND_MAX`, which is also defined in `cstdlib` (we will say much more about the generation of random numbers in *Module 17, Stochastic models*). The random number generator has to be seeded with an arbitrary seed that is passed to the function `std::srand()` (called on line 70). Change the seed to another value to obtain another realisation of the stochastic process of mutation and selection.

Study the program code and run a couple of different replicate simulations by varying the random seed. You will notice that the program runs fine for a couple generations, but then suddenly terminates after reporting a failed assertion. The problem is caused by a single subtle bug. Use the debugger and your bug-hunting skills to locate the error and fix the code to eliminate the bug.

```
1  #include <iostream>
2  #include <vector>
3  #include <cassert>
4  #include <cstdlib>
5
6  const int       iOffspring  = 1000;      // max population size
7  const double    s           = 0.01;      // selection coefficient
8  const double    mu          = 0.001;     // mutation rate
9
10 struct Individual {
11     bool isWTAllele, isAlive;
12 };
13
14 std::vector<Individual> population(iOffspring, {true, true});
15
16 void mutation()
17 {
18     for(int i = 0; i < population.size(); ++i)
19         if(population[i].isWTAllele && std::rand() < mu * RAND_MAX)
20             population[i].isWTAllele = false;
21 }
22
23 void selection()
24 {
25     // individuals carrying the deleterious mutation die with probability s
26     for(int i = 0; i < population.size(); ++i)
27         if(!population[i].isWTAllele && std::rand() < s * RAND_MAX)
28             population[i].isAlive = false;
29         else
30             population[i].isAlive = true;
31 }
32
33 void removeDeadIndividuals()
34 {
35     for(int i = 0; i < population.size(); ++i) {
36         if(!population[i].isAlive) {
37             // replace individual by last individual
38             population[i] = population.back();
39             population.pop_back();
40         }
41     }
```

```
}

void produceOffspring()
{
    assert(population.size() > 0);
    std::vector<Individual> offspring;
    for(int i = 0; i < iOffspring; ++i) {
        // select parent for offspring i
        const int j = i % population.size();
        assert(population[j].isAlive);
        offspring.push_back(population[j]);
    }
    // replace parents by their offspring
    population = offspring;
}

void showData(int t)
{
    int cnt = 0;
    for(int i = 0; i < population.size(); ++i)
        if(!population[i].isWTAllele)
            ++cnt;
    std::cout << t << ' ' << cnt * 1.0 / population.size() << '\n';
}

int main()
{
    // seed random number generator
    std::srand(127264);

    // start simulation
    for(int t = 0; t < 1000; ++t) {
        mutation();
        selection();
        removeDeadIndividuals();
        produceOffspring();
        showData(t);
    }
    return 0;
}
```

15.3. The efficiency/accuracy trade-off. For many algorithms, there is a positive relationship between the precision of the result and the number of calculations that had to be performed in order to achieve the desired accuracy. Put simply, if the result needs to be more accurate, you need to run the algorithm for a longer time. In this exercise, we will examine this fundamental trade-off for the Leslie matrix population model implemented in Listing 15.3.

The normalised age distribution $\mathbf{n}_t$ in a Leslie matrix population model eventually converges to the stable age distribution $\mathbf{v}$. It can be shown that the difference between $\mathbf{n}_t$ and $\mathbf{v}$ after a while decays by a constant factor each generation, such that

$$||\mathbf{n}_t - \mathbf{v}|| \sim \left(\frac{\lambda'}{\lambda}\right)^t$$

for large t. Here, λ' is the second largest eigenvalue of the Leslie matrix $\mathbf{L}$.

Given this result, derive an expectation for the number of steps you need to iterate Algorithm 2 in order to achieve a desired accuracy ϵ. Check your expectation against the results obtained from the program in Listing 15.3. Run simulations for values of ϵ ranging from $1 \cdot 10^{-3}$ to $1 \cdot 10^{-12}$ and note down the number of iteration steps required to converge to the equilibrium age distribution. Plot the results in Excel, and compare with your theoretical prediction. What do you conclude?

Test your skill

15.4. Modelling sexual selection ($\star$). The multivariate breeder's equation,

$$\Delta \mathbf{z} = \mathbf{G}\,\beta$$

provides a generic description of the evolution of quantitative traits and is widely applied in animal breeding and evolutionary quantitative genetics. According to this equation, the per-generation change $\Delta\mathbf{z}$ in the mean values of a set of n quantitative characters $\mathbf{z} = (z_1, z_2, \ldots, z_n)$, is determined by two factors. The first is the selection gradient vector β, which specifies the direction and intensity of selection in multi-dimensional trait space. The second factor is the additive genetic variance-covariance matrix $\mathbf{G}$, which gives information about the amount of genetic variation available in the population and the presence of genetic covariance between characters.

As an example of an application of the multivariate breeder's equation consider a classical model of sexual selection, where

$$\mathbf{z} = \begin{pmatrix} t \\ p \end{pmatrix}$$

contains the mean trait values of a male secondary sexual ornament, t, and a female choosiness value, p (females with $p = 0$ have no preference; those with $p > 0$ prefer larger values of the ornamentation trait, while those with $p < 0$ prefer smaller ornaments). For this model, the selection gradient is given by

$$\beta = \begin{pmatrix} \beta_t \\ \beta_p \end{pmatrix} = \begin{pmatrix} p - c_\mathrm{t}\, t \\ -c_\mathrm{p}\, p \end{pmatrix}$$

According to these expressions, the evolution of the male ornament is favoured by sexual selection if females are very choosy, but also opposed by natural selection, which reduces the survival of males carrying an exaggerated ornament. The selection gradient on the male ornamentation trait, $\beta_t = p - c\,t$, reflects a balance between these two forces, where c_t is a parameter that quantifies the relative viability cost of ornamentation. The viability selection optimum for females is to mate randomly, reflecting the assumption that mate choice is costly. However, stabilising natural selection on females is much weaker than on males, such that $c_\mathrm{p} \ll c_\mathrm{t}$.

The genetic variance-covariance matrix of the sexual selection model is given by

$$\mathbf{G} = \begin{pmatrix} V_\mathrm{t} & C_\mathrm{pt} \\ C_\mathrm{pt} & V_\mathrm{p} \end{pmatrix}$$

Here, V_t and V_p are the additive genetic variance of male trait and female preference, respectively, and C_pt is their genetic covariance.

Write a simulation algorithm in C^{++} to iterate the multivariate breeder's equation, in order to follow the mean trait values of the male ornament and the female preference as they change over the course of multiple generations. The algorithm should perform the following sequence of steps:

- Initialise the genetic variance-covariance matrix $\mathbf{G}$.
- Set initial trait values to $t = t_0$ and $p = p_0$.
- Start iteration loop. For time from 0 to the final generation:
    1. Compute the elements of the selection gradient vector from the current trait values.
    2. Multiply the selection gradient vector with the $\mathbf{G}$ matrix in order to calculate the change of the trait values $\Delta\mathbf{z}$.
    3. Update the mean trait values and increment time.

Construct the algorithm in such a way that you can easily adjust the code to simulate other quantitative genetic models. Also, do not forget to add appropriate exception handling. For example, take into account that the feasible values for the covariance are constrained by the magnitude of the variances: $|C_\mathrm{pt}| \leq \sqrt{V_\mathrm{t}\, V_\mathrm{p}}$.

(a) Run a simulation for 1000 generations, starting from the initial trait values $t_0 = 0$, $p_0 = 0.05$. Use the following parameter set: $V_\mathrm{t} = V_\mathrm{p} = 0.1$, $C_\mathrm{pt} = 0.02$, $c_\mathrm{t} = 0.5$ and $c_\mathrm{p} = 0.02$. Describe the observed dynamics. Can costly preferences be maintained?

(b) Do the same as in (a), except increase the genetic covariance to $C_\mathrm{pt} = 0.04$. Do your conclusions change?

(c) Increase the covariance even more, to $C_{\mathrm{pt}} = 0.06$. Relate the model's prediction for this parameter set to the observation that sexually selected traits are often highly exaggerated and costly. What is still missing from the model?

(d) The previous simulations indicate that the presence of a sufficiently large positive genetic covariance between female preference and male ornamentation genes is a prerequisite for the evolution of costly mating preferences. Do you expect such a positive genetic covariance to be present? Which factors contribute to the build-up of genetic covariance, and which are responsible for its decay over time?

15.5 Solutions to the exercises

Exercise 15.1. The bisection algorithm for locating the root of a function.

```
#include <iostream>
#include <cmath>
#include <exception>
#include <cstdlib>

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

double findRoot(double a, double b)
{
    const double    epsilon = 1.0e-8;    // desired precision
    const int       maxIt   = 100;       // maximum number of bisection steps

    // verify that f changes sign between a and b
    double fa = f(a);
    if(fa * f(b) > 0.0)
        throw std::logic_error("unable to locate root in interval.\n");

    // orient the search such that f(b) > 0
    if(fa > 0.0)
        std::swap(a,b);

    // start bisection loop
    for(int it = 0; it < maxIt; ++it) {
        double c = (a + b) / 2;
        if(fabs(a - b) < epsilon)
            return c;
        if(f(c) <= 0.0)
            a = c;
        else
            b = c;
    }
    throw std::runtime_error("too many iterations in findRoot().\n");
}

int main()
{
    try {
        std::cout << findRoot(-2.0, 0.0) << '\n';
    }
    catch(std::exception &error) {
        std::cerr << "error: " << error.what();
        exit(EXIT_FAILURE);
    }
    return 0;
}
```

Exercise 15.2. Practise your bug-hunting skills. Hint: the bug is caused by the fact that the function `removeDeadIndividuals()` does not always remove all dead individuals.

Exercise 15.3. The efficiency/accuracy trade-off. The number of steps needed to converge to the equilibrium age distribution is expected to be proportional to $-\log(\epsilon)$. Simulations of the program in Listing 15.3 are in close agreement with this prediction for ϵ ranging from $1 \cdot 10^{-3}$ to $1 \cdot 10^{-12}$. For even smaller values of ϵ, floating-point underflow errors start to interfere with the calculations, so that the results become less accurate.

Part II:

Implementing Biological Models in C++

Module 16

Numerical integration of Ordinary Differential Equations (ODEs)

Jamie explains what's on the menu

In this module you will:

- Be introduced to numerical integration algorithms, which are used for solving systems of ODEs on a computer.
- Apply these methods to simulate ODE models of biological systems.
- Learn a technique for controlling the approximation error of numerical integration methods.

16.1 Ordinary Differential Equations (ODEs)

Ordinary differential equations (ODEs) are widely used to model dynamical processes in biology. An ODE relates the value of an unknown function to that of its derivatives. In an ODE, all derivatives are taken with respect to a single variable (which often is time). Equations that contain derivatives with respect to multiple variables (e.g., time and one or more spatial coordinates) are so-called Partial Differential Equations (PDEs).

A well-known example of an ODE is the logistic growth equation, which captures the relationship between the density, $x(t)$, of a population at time t and the rate $\frac{d}{dt}x(t)$ at which it currently grows while experiencing competition for a limiting resource:

$$\frac{d\,x(t)}{dt} = r\,x(t)\left(1 - \frac{x(t)}{K}\right) \tag{16.1}$$

Here, K is the carrying capacity of the system, defined as the maximum population density of the species that the environment can sustain, and r is maximum per-capita growth rate.

ODEs are called linear when the equation depends linearly on the unknown function and its derivatives. The logistic growth equation is an example of a nonlinear ODE, because the right-hand side contains a term that is quadratic in $x(t)$. The distinction between linear and non-linear equations is important, because linear equations can be solved analytically, whereas the majority of non-linear equations cannot. If an ODE can be solved, this means that it is possible to write the dependent variable as an explicit

function of the independent variable. The logistic equation belongs to the minority of nonlinear equations that can be solved analytically. Its solution is given by

$$x(t) = \frac{x_0\, K\, \exp(r\, t)}{K + x_0\, (\exp(r\, t) - 1)} \tag{16.2}$$

where x_0 is the population density at $t = 0$. (You can verify that this function is a solution of Equation 16.1 by calculating its time derivative $\frac{d}{dt}x(t)$ and substituting the resulting expression into Equation 16.1 together with the expression for $x(t)$. After some algebraic manipulation, you will see that the resulting equation holds for all t.)

ODEs can also be classified as either autonomous or non-autonomous equations. Autonomous ODEs do not depend explicitly on the independent variable, non-autonomous ODEs do. The logistic equation is an autonomous ODE, but if either r or K would vary in time, for example, it would turn into an non-autonomous equation. Finally, ODEs need not only depend on the first derivative (ODEs that do are called first-order), but can also contain higher-order derivatives. As long as the higher-order derivatives appear in a linear fashion, however, higher-order ODEs can be rewritten into a system of first-order ODEs. For example, the linear second-order ODE

$$\frac{d^2\, x(t)}{dt^2} + \epsilon \frac{d\, x(t)}{dt} + b\, x(t) = 0 \tag{16.3}$$

which describes the motion of a pendulum subject to damping due to a frictional force, can be rewritten as a system of linear first-order ODEs

$$\begin{cases} \frac{d}{dt}x(t) = y(t) \\ \frac{d}{dt}y(t) + \epsilon\, y(t) + b\, x(t) = 0 \end{cases}$$

Many biological models take the form of systems of nonlinear, first-order ODEs. These models rarely have an analytical solution, but we can analyse their behaviour by computational methods that produce approximate numerical solutions. Two of these methods will be introduced in this Module. They can be applied to n-dimensional systems of the general form

$$\begin{cases} \frac{d}{dt}x_1(t) = F_1(t, x_1(t), x_2(t), \ldots, x_n(t)) \\ \frac{d}{dt}x_2(t) = F_2(t, x_1(t), x_2(t), \ldots, x_n(t)) \\ \qquad\qquad \vdots \\ \frac{d}{dt}x_n(t) = F_n(t, x_1(t), x_2(t), \ldots, x_n(t)) \end{cases} \tag{16.4}$$

The functions $F_1, F_2, \ldots, F_n$ that appear on the right-hand side are allowed to depend on all of the dependent variables $x_1, x_2, \ldots, x_n$ and there are no fundamental restrictions on the type of this dependence other than that $F_1, F_2, \ldots, F_n$ are sufficiently smooth. The functions are also allowed to depend on t directly (as in non-autonomous systems) but they do not need to be explicitly time-dependent. In other words, the two methods also work for autonomous systems.

16.2 Euler's method

The simplest method for the numerical integration of ODEs is Euler's method. This method relies on an approximation inspired by the limit definition of a derivative

$$\frac{d\, x(t)}{dt} = \lim_{h \to 0} \frac{x(t + h) - x(t)}{h}$$

If we ignore the fact that the definition involves a limit and simply use a small value for h, the above equality no longer holds exactly but still allows for a reasonable approximation of the derivative. Hence, we may replace the derivative in the differential equation

$$\frac{d\, x(t)}{dt} = F(t, x(t))$$

by a difference, to obtain

$$\frac{x(t + h) - x(t)}{h} \approx F(t, x(t))$$

Multiplying both sides with h and isolating $x(t + h)$ on the left-hand side, leads to a simple recipe for obtaining the function value at time $t + h$, given the value at an earlier time point t:

$$x(t + h) \approx x(t) + h\, F(t, x(t)) \tag{16.5}$$

Suppose now that the value of x is known at time $t = 0$. With this information, and recipe 16.5, we can then calculate x at $t = h$ and, from there, at $t = 2h$, and so on, to any arbitrary point in time.

Recipe 16.5 is readily generalised to systems of ODEs of the general form 16.4. All we need to do is evaluate the functions $F_1, F_2, \ldots, F_n$ for the current values of the variables $x_1(t), x_2(t), \ldots, x_n(t)$, and then update the variables simultaneously. Code listing 16.1 illustrates how this is done in C^{++}. The program computes a numerical solution of an ODE model of the interaction between a prey with density $x_1(t)$ and a predator with density $x_2(t)$ that change according to the equations

$$\frac{d\,x_1(t)}{dt} = r\,x_1(t)\left(1 - \frac{x_1(t)}{K}\right) - a\,x_2(t)\,\frac{x_1(t)}{H + x_1(t)}$$
$$\frac{d\,x_2(t)}{dt} = a\,c\,x_2(t)\,\frac{x_1(t)}{H + x_1(t)} - d\,x_2(t)$$

some explanation of the meaning of the model parameters is provided in the code. The program in Listing 16.1 can easily be adjusted to simulate different ODE models. All you need to do is adjust the number of variables `nvar` and the code listed under the header

```
//*** model specification ******************************
```

The function `rhs()` in this section specifies the right-hand side of the differential equation for each variable.

Code listing 16.1: Euler integration

```
#include <iostream>
#include <fstream>
#include <vector>
#include <exception>
#include <cstdlib>

//*** parameters of the integration algorithm *********

const int       nvar = 2;       // number of variables
const double    tEnd = 200.0;   // simulation time
const double    dt = 0.01;      // step size
const double    dtsav = 0.5;

//*** model specification ******************************

const double    r = 2.0;        // maximum exponential growth rate of prey
const double    K = 0.32;       // carrying capacity of the prey
const double    a = 1.0;        // predator attack rate
const double    H = 0.1;        // half-saturation constant of the predator functional response
const double    c = 0.2;        // #predator produced per consumed prey
const double    d = 0.1;        // death rate of the predator

void rhs(const double &t, const std::vector<double> &x, std::vector<double> &dxdt)
// definition of the right-hand side of the ODE model
// t is the current time
// vector x contains the current values of the variables
// vector dxdt is filled with the values of the derivatives dx/dt
{
    double aux = a * x[1] * x[0] / (H + x[0]);      // consumption of prey by predator

    dxdt[0] = r * x[0] * (1.0 - x[0] / K) - aux;    // prey equation
    dxdt[1] = c * aux - d * x[1];                   // predator equation
}

//*** ODE integration routine ************************

void eulerStepper(double &t, std::vector<double> &x, const double &h)
{
    // compute gradient
    std::vector<double> dxdt(nvar);
    rhs(t, x, dxdt);

```

```
    // advance solution
    for(int i = 0; i < nvar; ++i)
        x[i] += h * dxdt[i];
    t += h;
}

//*** function main() ********************************

int main()
{
    try {
        // open data file
        std::ofstream ofs("data.csv");
        if(!ofs.is_open())
            throw std::runtime_error("unable to open file.\n");

        // set initial conditions
        std::vector<double> x(nvar, 0.01);

        // start numerical integration
        for(double t = 0.0, tsav = 0.0; t < tEnd; ) {
            eulerStepper(t, x, dt);

            if(t > tsav) {
                std::cout   << t << ' ' << x[0] << ' ' << x[1] << '\n';
                ofs         << t << ',' << x[0] << ',' << x[1] << '\n';
                tsav += dtsav;
            }
        }
        ofs.close();
    }
    catch(std::exception &error) {
        std::cerr << "error: " << error.what();
        exit(EXIT_FAILURE);
    }
    return 0;
}
```

16.3 Runge-Kutta integration

The error made in approximation (16.5) is of the same order of magnitude as h^2. This means that the step size of the Euler integration method must be rather small whenever a high level of precision is required. Yet, if the step size is decreased, it takes proportionally longer for the simulation to complete, because the algorithm needs more steps to cover the same time interval. Whereas all integration algorithms have to deal with this fundamental speed-accuracy trade-off somehow, algorithms that invest a bit more effort in updating the solution from one integration step to the next can be much more efficient than Euler integration. In particular, an algorithm that is very often used in numerical integration routines is the Runge-Kutta method. The key idea of the Runge-Kutta method is to iteratively refine the estimate of the solution based on multiple evaluations of the right-hand side of the ODE for each integration step.

To illustrate the Runge-Kutta method, we consider once more a single variable problem:

$$\frac{d\,x(t)}{dt} = F(t, x(t))$$

Assuming that $x(t)$, the value of the variable at time t is known, $x(t+h)$ is calculated as

$$x(t+h) \approx x(t) + \frac{h}{6}\,(a + 2\,b + 2\,c + d) \tag{16.6}$$

where

$$
\begin{aligned}
a &= F(t, x(t)) \\
b &= F(t + \tfrac{1}{2}h, x(t) + \tfrac{1}{2}h\,a) \\
c &= F(t + \tfrac{1}{2}h, x(t) + \tfrac{1}{2}h\,b) \\
d &= F(t + h, x(t) + h\,c)
\end{aligned}
$$

After the solution has been projected forward to $x(t+h)$, the procedure can be repeated to find $x(t+2h)$, and so on, until the solution has been calculated over the desired time interval. Just as for Euler integration, the algorithm needs the function value $x(t_0)$ at an initial time point $t = t_0$ to get started.

A Runge-Kutta driver capable of handling systems of ODEs can be programmed in the same way as shown earlier for Euler integration. In fact, the Runge-Kutta integration stepper in Listing 16.2 can replace the function `eulerStepper()` in Listing 16.1. Just make sure to also replace the call to `eulerStepper()` on line 64 of the Listing 16.1 by a call to `rungeKuttaStepper()` to ensure that the solution is updated by the new method.

Code listing 16.2: Runge-Kutta integration stepper

```
1  void rungeKuttaStepper(double &t, std::vector<double> &x, const double &h)
2  {
3      // step 1
4      std::vector<double> dxdt1(nvar);
5      rhs(t, x, dxdt1);
6
7      // step 2
8      std::vector<double> xtmp(nvar);
9      for(int i = 0; i < nvar; ++i)
10         xtmp[i] = x[i] + 0.5 * h * dxdt1[i];
11     std::vector<double> dxdt2(nvar);
12     rhs(t + 0.5 * h, xtmp, dxdt2);
13
14     // step 3
15     for(int i = 0; i < nvar; ++i)
16         xtmp[i] = x[i] + 0.5 * h * dxdt2[i];
17     std::vector<double> dxdt3(nvar);
18     rhs(t + 0.5 * h, xtmp, dxdt3);
19
20     // step 4
21     for(int i = 0; i < nvar; ++i)
22         xtmp[i] = x[i] + h * dxdt3[i];
23     std::vector<double> dxdt4(nvar);
24     rhs(t + h, xtmp, dxdt4);
25
26     // advance solution
27     for(int i = 0; i < nvar; ++i)
28         x[i] += h * (dxdt1[i] + 2.0 * dxdt2[i] + 2.0 * dxdt3[i] + dxdt4[i]) / 6.0;
29     t += h;
30 }
```

16.4 Exercises

16.1. Oscillations in a predator-prey model. Consider the following simplified predator-prey model:

$$
\begin{aligned}
\frac{d\,x_1(t)}{dt} &= r\,x_1(t) - a\,x_1(t)\,x_2(t) \\
\frac{d\,x_2(t)}{dt} &= a\,c\,x_1(t)\,x_2(t) - d\,x_2(t)
\end{aligned}
$$

Adjust the code from Listing 16.1 to simulate the model. Set the parameter values at $r = 2$, $a = 1$, $c = 0.2$ and $d = 0.1$, and start the Euler integration from the initial conditions $x_1(0) = 0.4$ and $x_2(0) = 2.0$. (*a*) Run the simulation until $t = 200$, using integration step size $dt = 0.05$. Plot the simulation results, and describe the outcome. (*b*) Next, replace the Euler integration stepper by the Runge-Kutta stepper of Listing 16.2, and repeat the simulation using Runge-Kutta integration. Compare the two simulations. (*c*) A mathematical analysis indicates that the solutions of the model form an infinite set of periodic orbits. Accordingly, irrespective of where the simulation is started, prey and predator will periodically return to their initial densities. Given this information, can you explain why the Euler integration algorithm fails to produce the correct solution?

16.2. Lower- versus higher-order integration schemes. Since the equation for the damped pendulum (16.3) can be rewritten as a system of linear ODEs, it is possible to find an analytical solution for it. For the initial condition $x(0) = 1$, $y(0) = x'(0) = 0$, this analytical solution is given by

$$x_{\text{true}}(t) = \exp\left(-\frac{\epsilon}{2}\,t\right)\left(\cos(\omega\,t) + \frac{\epsilon}{2\,\omega}\sin(\omega\,t)\right) \tag{16.7}$$

where $\omega = \sqrt{b - \epsilon^2/4}$.

The fact that an analytical solution exists, allows us to quantify the approximation error made by different numerical integration routines. To do this, we have to numerically solve the ODE system for the damped pendulum,

$$\begin{cases} \frac{d}{dt}x(t) = y(t) \\ \frac{d}{dt}y(t) = -\epsilon\,y(t) - b\,x(t) \end{cases} \tag{16.8}$$

and compare the numerical solution $x_{\text{num}}(t)$ with the analytical solution $x_{\text{true}}(t)$ given in Eq. (16.7). To quantify the approximation error that was accumulated by the numerical integration algorithm up to the time point t, we can measure the absolute difference between the numerical and the analytical solution at that point:

$$\text{global error} = |x_{\text{num}}(t) - x_{\text{true}}(t)|$$

For answering the questions below, you can evaluate the error at $t = 1$ and set the parameters to $b = 1000$ and $\epsilon = 4$.

(a) Reduce the integration step size in small steps from $h = 0.01$ to $h = 1 \cdot 10^{-5}$ and study how this affects the global error for the Euler method. Plot the data in a log-log plot to verify that the global error scales as h^1. This scaling relationship indicates that Euler integration is a first-order method.

(b) Perform the same analysis for Runge-Kutta integration, and verify in a log-log plot that the global error scales as h^4. This indicates that the Runge-Kutta scheme is a fourth-order integration method. Which other source of errors might be responsible for the fact that the Runge-Kutta method appears to become less accurate below $h = 1 \cdot 10^{-4}$?

(c) How do the two methods compare in efficiency for this particular ODE system, taking into account that each Runge-Kutta step requires 4 evaluations of the right-hand sides of the ODEs?

(d) The Runge-Kutta integration method that you have so far used, belongs to a family of Runge-Kutta integration schemes that contain methods of different order. These differ in the way each integration step is subdivided into stages, resulting in a different scaling relationship between integration step size and the accumulated integration error. Would you expect that higher-order methods are always more efficient than lower-order ones? Why/why not?

16.3. Bistability in gene-regulation (⋆⋆). Consider the following model of an auto-catalytic genetic control system:

$$\begin{cases} \frac{d}{dt}x(t) = -a\,x(t) + y(t) \\ \frac{d}{dt}y(t) = \frac{x(t)^2}{1+x(t)^2} - b\,y(t) \end{cases} \tag{16.9}$$

Here, $x(t)$ represents the concentration of a protein that decays at rate a, and whose rate of production is proportional to the concentration of mRNA, $y(t)$. The mRNA is degraded at rate b and produced at a rate that is a saturating function of the protein concentration. This function reflects the common situation that two copies of the protein form a dimer which then binds to the gene promotor to induce activity.

It can be shown mathematically that the gene can be in two stable states when $2\,a\,b < 1$. One of these stable equilibria ($x = y = 0$) corresponds to the off-state of the gene; at the other equilibrium, the gene is active ($x > 0$ and $y > 0$).

(a) Write a function in C^{++} that numerically integrates equation system (16.9) from the initial condition $x(0) = 0$, $y(0) = y_0$. This function should take the initial value y_0 as an argument and return a boolean value to indicate which of the two stable equilibria the system converged to. Verify that the gene-regulatory system switches on only if $2\,a\,b < 1$ and the initial concentration of mRNA is sufficiently high.
Hint 1: To determine whether the system has reached equilibrium, check how much the variables are changing in a single integration step; if the amount of change is very small, you can assume the variables are close to equilibrium.
Hint 2: A mathematical analysis of the model indicates that the equilibrium protein concentration is not smaller than $1/(2\,a\,b)$ when the gene is active.

(b) Adjust the root-finding algorithm from exercise 15.1 in order to locate the initial mRNA concentration threshold for gene activation, y_0^*.

(c) Compute the mRNA threshold concentration as a function of the parameter a, for a ranging from 0.2 to 1.2 and $b = 0.5$.

Test your skill

16.4. The Hodgkin-Huxley model for the action potential ($\star$). A classic model of the initiation and propagation of action potentials by nerve cells was proposed by Alan Lloyd Hodgkin and Andrew Fielding Huxley in 1952, based on their electrophysiological measurements in the squid giant axon. Hodgkin and Huxley received the 1963 Nobel Prize in Physiology/Medicine for this work. The core component of the Hodgkin-Huxley model is an equation that describes the change of the membrane potential as a function of three ionic currents, a Na^+ current that depolarises the cell, a K^+ current that repolarises the cell, and a group of unspecified other currents that help to restore the resting state. The equation for the membrane potential V is

$$\frac{dV}{dt} = -\frac{1}{C}(120.0\,m^3\,h\,(V + 115.0) + 36.0\,n^4\,(V - 12.0) + 0.3\,(V + 10.5989) + S(t))$$

where $C = 1.0$ is the capacity of the membrane, $S(t)$ is a time-dependent stimulus current and time t is measured in ms. The variables m, h and n are gating variables that represent the state of the voltage sensitive sodium and potassium channels in the membrane. To simplify the notation, we do no longer write explicitly that these variables and the membrane potential are functions of t.

The dependence of the gating variables on the voltage was measured by Hodgkin and Huxley in clever patch-clamp experiments, in which they systematically blocked specific channels in order to measure the currents through each channel separately. Based on fits of logistic curves to the patch-clamp data, Hodgkin and Huxley proposed the following equations for the gating variables m, h and n

$$\begin{aligned}
\frac{dm}{dt} &= 0.1\,(1 - m)\frac{V + 25.0}{\exp(0.1\,V + 2.5) - 1} - 4.0\,m\,\exp(V/18.0) \\
\frac{dh}{dt} &= 0.07\,(1 - h)\,\exp(V/20.0) - \frac{h}{\exp(0.1\,V + 3.0) + 1} \\
\frac{dn}{dt} &= 0.01\,(1 - n)\frac{V + 10.0}{\exp(0.1\,V + 1.0) - 1} - 0.125\,n\,\exp(V/80.0)
\end{aligned}$$

Implement the Hodgkin-Huxley model in C^{++} and apply Runge-Kutta integration to obtain numerical solutions from $t = 0$ to $t = 150$ ms for the following conditions:

(a) The neuron is stimulated with a sinusoidal current $S(t) = 10.0\,\left(1 - \cos\left(\frac{2\pi\,t}{150.0}\right)\right)$.

(b) The neuron is stimulated with a stepwise stimulus current

$$S(t) = \begin{cases} 10.0 & \text{if } 50 < \text{t} < 100 \\ 0.0 & \text{otherwise} \end{cases}$$

Split the integration into three parts in order to deal with the discontinuity in the stimulus function.

Suitable initial conditions for both cases are: $V_0 = 0$, $m_0 = 0.05$, $h_0 = 0.59$, $n_0 = 0.32$. Given that the membrane potential can vary rapidly within a short time interval when an action potential occurs, think

carefully about what values are appropriate for the integration time step and the time interval between storing data points.

When you inspect the output of the program, you will notice that the trajectory of the voltage generated by the Hodgkin-Huxley model is upside-down relative to the usual representation of the action potential. This is because Hodgkin and Huxley used a different convention for defining the membrane potential. In particular, the membrane potential in the modern convention corresponds to $-V$ in the Hodgkin-Huxley model.

Adaptive step-size control

Do not continue with this section before completing exercise 16.2.

Numerical integration methods always produce approximate results, but we can exert a lot of control over the quality of the approximation by manipulating the integration step size. This begs the question how we can identify the largest possible step size while ensuring that the error does not exceed an acceptable tolerance level. Moreover, it would be ideal to be able to control the step size 'on the fly', so that the algorithm can move ahead cautiously when the integration is sensitive to errors without compromising its ability to go fast through other parts of the solution where the demands on precision are not that severe. To enable such adaptive step-size control, we must somehow obtain an estimate of the local error made during an integration step.

The local error of an Euler integration step is $\mathcal{O}(h^2)$ (read: 'of the order of' h^2). This means that the difference between the projected solution $x(t+h)$ and the real solution is guaranteed to lie between $\pm c\,h^2$ for some constant c and sufficiently small values of h. The local errors in each integration step accumulate to give rise to a global error. For the Euler method, this global error is $\mathcal{O}(h)$ (see exercise 16.2). This is because the number of steps needed to cover a given time interval is inversely proportional to h, so that the global error is $h^{-1} \times \mathcal{O}(h^2) = \mathcal{O}(h)$.

The Runge-Kutta integration method that was introduced in section 16.3 has a local error $\mathcal{O}(h^5)$, which is negligible relative to the error made in an Euler integration step of the same size. It turns out that we can exploit this fact to obtain an estimate of the error made by the Euler integration method, without having to know the real solution $x_{\text{true}}(t+h)$. To see how this works, imagine that we execute an Euler integration step from the point $x(t)$ at time t, and obtain the solution $x_{\text{Euler}}(t+h)$. Then, the deviation of this estimate from the real solution is given by

$$\epsilon_{\text{Euler}} = x_{\text{Euler}}(t+h) - x_{\text{true}}(t+h) = \mathcal{O}(h^2)$$

Next, we execute a Runge-Kutta integration step from the same starting point to obtain $x_{\text{RK}}(t+h)$. The error of this estimate is given by

$$\epsilon_{\text{RK}} = x_{\text{RK}}(t+h) - x_{\text{true}}(t+h) = \mathcal{O}(h^5)$$

From these two equations, it follows that

$$\begin{aligned} x_{\text{Euler}}(t+h) - x_{\text{RK}}(t+h) &= (x_{\text{true}}(t+h) + \epsilon_{\text{Euler}}) - (x_{\text{true}}(t+h) + \epsilon_{\text{RK}}) \\ &= \epsilon_{\text{Euler}} - \epsilon_{\text{RK}} \\ &= \epsilon_{\text{Euler}} - \mathcal{O}(h^5) \\ &\approx \epsilon_{\text{Euler}} \end{aligned}$$

The final approximation step makes use of the fact that, for all $k \geq 0$, terms $\mathcal{O}(h^{k+1})$ can safely be ignored relative to terms $\mathcal{O}(h^k)$ if h is sufficiently small.

Now, if the absolute error $|\epsilon_{\text{Euler}}|$ is larger than the tolerance level τ, we should reject the step and reduce the step size. Alternatively, if the absolute error $|\epsilon_{\text{Euler}}|$ is smaller than the tolerance level, there is room to increase the step size a bit. In particular, given that $\epsilon_{\text{Euler}} \propto h^2$, the step size could be updated in the following way

$$h_{\text{new}} = h_{\text{old}} \sqrt{\frac{\tau}{|\epsilon_{\text{Euler}}|}}$$

In practice, however, this recipe is a bit too optimistic to be useful. If the estimate of the error is extremely small, we should be careful not to increase the step size by too much. In addition, staying a bit below the theoretically predicted maximum step size helps to reduce the number of steps that are rejected. We should also be aware, finally, that there is a limit to what we can achieve by decreasing the

step size: eventually, the precision of the calculations will also start to suffer from floating point roundoff error, so we should impose a minimum step size.

To make our suggested algorithm a bit more efficient, it makes sense to replace the Runge-Kutta integration step by another integration scheme that requires fewer function evaluations. In the end, it is not necessary to obtain an estimate that is accurate up to $\mathcal{O}(h^5)$, we would already be able to work the estimate given by a second order method. One such method is Heun's method, which is an integration scheme from the family of Runge-Kutta algorithms that has two stages. In each integration step, the solution is calculated as

$$x(t+h) = x(t) + \frac{h}{2}(a+b) \tag{16.10}$$

where

$$a = F(t, x(t))$$
$$b = F(t+h, x(t) + h\,a)$$

Listing 16.3 illustrates how the combination of Heun's method with Euler integration is applied in a numerical integration routine with adaptive step size control.

Code listing 16.3: Adaptive step-size control (Euler-Heun)

```
#include <iostream>
#include <fstream>
#include <vector>
#include <cmath>
#include <exception>
#include <cstdlib>

//*** parameters of the integration algorithm *********

const int       nvar        = 2;        // number of variables
const double    tEnd        = 1.0;      // max simulation time
const double    dt0         = 0.001;    // initial step size
const double    dtsav       = 0.01;     // time interval between data points
const double    tolerance   = 1.0e-4;   // acceptable local error during numerical integration

//*** model specification ****************************

const double    epsilon = 4.0;  // normalised coefficient of friction
const double    b = 1000.0;     // normalised spring constant

void rhs(const double &t, const std::vector<double> &x, std::vector<double> &dxdt)
// definition of the righ-hand side of the ODE model
// t is the current time
// vector x contains the current values of the variables
// vector dxdt is filled with the values of the derivatives dx/dt
{
    dxdt[0] = x[1];
    dxdt[1] = -epsilon * x[1] - b * x[0];
}

//*** ODE integration routine ************************

const double kdShrinkMax    = 0.1;      // decrease step size by no more than this factor
const double kdGrowMax      = 5.0;      // increase step size by no more than this factor
const double kdSafety       = 0.9;      // safety factor in adaptive stepsize control
const double kdMinH         = 1.0e-6;   // minimum step size

bool eulerHeunAdaptiveStepper(double &t, std::vector<double> &x, double &h)
{
    // step 1
    std::vector<double> dxdt1(nvar);
    rhs(t, x, dxdt1);

    // step 2
```

```
45      std::vector<double> xtmp(nvar);
46      for(int i = 0; i < nvar; ++i)
47          xtmp[i] = x[i] + h * dxdt1[i];
48      std::vector<double> dxdt2(nvar);
49      rhs(t + h, xtmp, dxdt2);
50
51      double errMax = 0.0;
52      for(int i = 0; i < nvar; ++i) {
53          // propagate solution by Heun's method
54          xtmp[i] = x[i] + 0.5 * h * (dxdt1[i] + dxdt2[i]);
55          // compute error
56          double erri = fabs(0.5 * h * (dxdt1[i] - dxdt2[i])) / tolerance;
57          if(erri > errMax)
58              errMax = erri;
59      }
60
61      // adjust step size
62      const double fct = errMax > 0.0 ? kdSafety / sqrt(errMax) : kdGrowMax;
63      if(errMax > 1.0) {
64          // reduce step size and reject step
65          if(fct < kdShrinkMax)
66              h *=  kdShrinkMax;
67          else
68              h *=  fct;
69          if(h < kdMinH)
70              throw std::runtime_error("step size underflow in eulerHeunAdaptiveStepper().");
71          return false;
72      }
73      else {
74          // update solution and increase step size
75          x = xtmp;
76          t += h;
77          if(fct > kdGrowMax)
78              h *=  kdGrowMax;
79          else
80              h *=  fct;
81          return true;
82      }
83  }
84
85  //*** function main() ********************************
86
87  int main()
88  {
89      try {
90          // open data file
91          std::ofstream ofs("data.csv");
92          if(!ofs.is_open())
93              throw std::runtime_error("unable to open file.\n");
94
95          // set initial conditions
96          std::vector<double> x(nvar);
97          x[0] = 1.0;
98          x[1] = 0.0;
99
100         // start numerical integration
101         int nOK = 0, nStep = 0;
102         double dtMin = dt0, dtMax = kdMinH;
103         for(double t = 0.0, tsav = 0.0, dt = dt0; t < tEnd; ++nStep) {
104             if(eulerHeunAdaptiveStepper(t, x, dt))
105                 ++nOK;
106
107             if(dt < dtMin)
108                 dtMin = dt;
109             else if(dt > dtMax)
110                 dtMax = dt;
```

```
111
112             if(t > tsav) {
113                 ofs         << t << ',' << x[0] << ',' << x[1] << ',' << dt << '\n';
114                 tsav += dtsav;
115             }
116         }
117         ofs.close();
118
119         // report integration data
120         std::cout   << "integration complete.\n"
121                     << "number of steps : " << nStep << '\n'
122                     << "proportion bad steps : " << 1.0 - nOK * 1.0 / nStep << '\n'
123                     << "average step size : " << tEnd / nStep << '\n'
124                     << "min step size : " << dtMin << '\n'
125                     << "max step size : " << dtMax << '\n';
126     }
127     catch(std::exception &error) {
128         std::cerr << "error: " << error.what();
129         exit(EXIT_FAILURE);
130     }
131     return 0;
132 }
```

The method is applied to simulate the movement of a pendulum subject to a frictional force (Eqs. 16.3 and 16.8), which was the system used previously to compare the global errors of Euler and Runge-Kutta integration (exercise 16.2). The screen output of the program is given by

```
integration complete.
number of steps : 6790
proportion bad steps : 0.00338733
average step size : 0.000147275
min step size : 7.47758e-05
max step size : 0.00108494
```

In addition, Fig. 16.2 displays the data written to the output file and shows how the step size responds to the adaptive control during the simulation.

There are a few aspects of the code that deserve additional attention. First, note on line 54 that Heun's method (the higher-order integration scheme) is used to propagate the solution, whereas the algorithm controls only the error of the Euler scheme (the lower-order method). Usually, the higher-order method will give the more accurate solution, so there is little harm in doing this, but we should be aware that we do not actually know the error of the higher-order method. Second, on line 56-58, you can see that the error measure used for adaptive step-size control in systems of ODEs is actually the largest of the errors calculated for the individual variables. Hence, the step size is controlled by whichever equation is the most difficult to solve. Third, in the section that implements the adjustment of the step size (line 62-81), the new step size is constrained to be no smaller than `kdShrinkMax * h`, and no larger than `kdGrowMax * h`, to prevent huge jumps in step size. Moreover, a factor `kdSafety` is included to accommodate for the fact that the error estimates obtained by the algorithm are not exact. Finally, the algorithm raises an exception when the step size falls below a minimal value `kdMinH` (line 70).

More exercises

16.5. Making sense of adaptive step-size control. To compute the local error of the Euler integration step, the algorithm in Listing 16.3 evaluates a weighted difference between the gradients `dxdt1` and `dxdt2` (see line 56).

(a) Show that this expression does indeed quantify the difference between the estimate from the Euler method (Eq. 16.5) and that of Heun's method (Eq. 16.10).

Now examine the simulation of the oscillating pendulum in Fig. 16.2 and take a close look at how the integration step size is adjusted by the adaptive step-size controller during the simulation.

(b) Why does the step size reach a maximum at moments that the pendulum reverses its direction of movement?

(c) Why does the integration go faster on average towards the end of the simulation?

16.6. Runge-Kutta integration with adaptive step-size control ($\star\star$). Now that we have an adaptive integration routine based on Euler and Heun's method, it takes only a bit of additional effort to develop a higher-order method with adaptive step-size control. Such a method combines efficiency with the robustness of adaptive step-size control. To make things not too complex, let's attempt to combine two methods from the family of Runge-Kutta integration schemes with local errors $\mathcal{O}(h^3)$ and $\mathcal{O}(h^4)$. The resulting integration scheme is known as the Bogacki-Shampine method. Once you know how to implement this method, it takes little effort to upgrade to a state-of-the-art numerical integration routine, such as the Dormand-Prince method, which is the default ODE solver in several professional software packages. A list of Runge-Kutta integration schemes suitable for adaptive step size control is available on https://en.wikipedia.org/wiki/List_of_Runge-Kutta_methods.

In the Bogacki-Shampine method, the solution given by the lowest order method is calculated as

$$x_{\mathrm{lo}}(t+h) = x(t) + \frac{h}{24}\,(7\,a + 6\,b + 8\,c + 3\,d) \tag{16.11}$$

whereas the solution given by the highest order method is:

$$x_{\mathrm{hi}}(t+h) = x(t) + \frac{h}{9}\,(2\,a + 3\,b + 4\,c) \tag{16.12}$$

The coefficients that appear in both equations are given by:

$$\begin{aligned} a &= F(t, x(t)) \\ b &= F(t + \tfrac{1}{2}\,h, x(t) + \tfrac{1}{2}\,h\,a) \\ c &= F(t + \tfrac{3}{4}\,h, x(t) + \tfrac{3}{4}\,h\,b) \\ d &= F(t + h, x(t) + \tfrac{1}{9}\,h\,(2\,a + 3\,b + 4\,c)) \end{aligned}$$

The error of the lowest order method, ϵ_{lo}, is given -exactly as before- by the difference

$$\begin{aligned} \epsilon_{\mathrm{lo}} &= x_{\mathrm{lo}}(t+h) - x_{\mathrm{hi}}(t+h) \\ &= (x(t) + \frac{h}{24}\,(7\,a + 6\,b + 8\,c + 3\,d)) - (x(t) + \frac{h}{9}\,(2\,a + 3\,b + 4\,c)) \\ &= h\,(\tfrac{5}{72}\,a - \tfrac{1}{12}\,b - \tfrac{1}{9}\,c + \tfrac{1}{8}\,d) \end{aligned}$$

Since this error is $\mathcal{O}(h^3)$, the naive suggestion for the new step size is

$$h_{\mathrm{new}} = h_{\mathrm{old}}\,\sqrt[3]{\frac{\tau}{|\epsilon_{\mathrm{lo}}|}}$$

However, it is better to be a bit conservative, in the same way as we have been for the Euler-Heun integration scheme.

Implement the Bogacki-Shampine method. Take the code in Listing 16.3 as a starting point, or start from scratch if you like an extra challenge. Once you have made a first working version, attempt to make your algorithm more efficient by making us of the fact that the final gradient calculated during an integration step d, is equal to the first gradient calculated in the next step. This is because

$$d = F(t + h, x(t) + \tfrac{1}{9}\,h\,(2\,a + 3\,b + 4\,c)) = F(t + h, x(t + h)) = a_{\mathrm{next}}$$

16.5 Solutions to the exercises

Exercise 16.1. Oscillations in a predator-prey model. To implement the model, the section *//*** model specification **** in Listing 16.1 has to be adjusted as follows:

```
//*** model specification ****************************

const double    r = 2.0;        // maximum exponential growth rate of prey
const double    a = 1.0;        // predator attack rate
const double    c = 0.2;        // #predator produced per consumed prey
const double    d = 0.1;        // death rate of the predator

void rhs(const double &t, const std::vector<double> &x, std::vector<double> &dxdt)
// definition of the right-hand side of the ODE model
// t is the current time
// vector x contains the current values of the variables
// vector dxdt is filled with the values of the derivatives dx/dt
{
    double aux = a * x[1] * x[0];           // consumption of prey by predator

    dxdt[0] = r * x[0] - aux;               // prey equation
    dxdt[1] = c * aux - d * x[1];           // predator equation
}
```

The resulting simulations show:

(a) oscillations in prey and predator densities that slowly increase in amplitude

(b) oscillations in prey and predator densities that do not change in amplitude.

(c) Given that the true solutions form neutrally stable periodic orbits around the equilibrium point, the integration errors made by the Euler integration algorithm can accumulate. This happens because the Euler stepper all the time advances the solution to a point that lies a little bit outside the periodic orbit of the true (analytical) solution. Hence, at each step, the numerical approximation ends up on another solution curve that lies a little bit further from the equilibrium point.

Exercise 16.2. Lower- versus higher-order integration schemes.

(a,b) See Fig. 16.1 for the results that were generated by the C^{++} program listed below. The reduced efficiency of Runge-Kutta integration at low step sizes is due to floating-point round-off errors that start to interfere with the calculations.

(c) For this model, Runge-Kutta integration is way more efficient than Euler integration. For example, if we accept a global error of $1 \cdot 10^{-4}$, the number steps required by the Runge-Kutta method more than a thousand times lower.

(d) Higher-order methods need not always be more efficient: they can be unnecessarily complex and are more likely to break down when applied to tricky ODE systems. This includes situations when the righthand side of an ODE varies steeply with one of the variables, or when one variable changes very slowly, whereas another changes very fast. In practice, variants of Runge-Kutta algorithms with order 4 to 8 reflect an optimal compromise between efficiency and robustness. Therefore, these are the workhorse method of choice for day-to-day scientific applications. In case plain old Runge-Kutta integration does not deliver, you can find many more sophisticated integration methods tailored to specific challenging situations in the scientific computing literature.

```
#include <iostream>
#include <fstream>
#include <vector>
#include <cmath>
#include <exception>
#include <cstdlib>

```

```
8  //*** parameters of the integration algorithm *********
9
10 const int       nvar  = 2;       // number of variables
11 const double    tEnd  = 1.0;     // simulation time
12 double          dt    = 0.01;    // step size
13
14 //*** model specification ****************************
15
16 const double    epsilon = 4.0;   // normalised coefficient of friction
17 const double    b = 1000.0;      // normalised spring constant
18
19 void rhs(const double &t, const std::vector<double> &x, std::vector<double> &dxdt)
20 // definition of the righ-hand side of the ODE model
21 // t is the current time
22 // vector x contains the current values of the variables
23 // vector dxdt is filled with the values of the derivatives dx/dt
24 {
25     dxdt[0] = x[1];
26     dxdt[1] = -epsilon * x[1] - b * x[0];
27 }
28
29 //*** ODE integration routines ************************
30
31 void eulerStepper(double &t, std::vector<double> &x, const double &h)
32 {
33     // compute gradient
34     std::vector<double> dxdt(nvar);
35     rhs(t, x, dxdt);
36
37     // advance solution
38     for(int i = 0; i < nvar; ++i)
39         x[i] += h * dxdt[i];
40     t += h;
41 }
42
43 void rungeKuttaStepper(double &t, std::vector<double> &x, const double &h)
44 {
45     // step 1
46     std::vector<double> dxdt1(nvar);
47     rhs(t, x, dxdt1);
48
49     // step 2
50     std::vector<double> xtmp(nvar);
51     for(int i = 0; i < nvar; ++i)
52         xtmp[i] = x[i] + 0.5 * h * dxdt1[i];
53     std::vector<double> dxdt2(nvar);
54     rhs(t + 0.5 * h, xtmp, dxdt2);
55
56     // step 3
57     for(int i = 0; i < nvar; ++i)
58         xtmp[i] = x[i] + 0.5 * h * dxdt2[i];
59     std::vector<double> dxdt3(nvar);
60     rhs(t + 0.5 * h, xtmp, dxdt3);
61
62     // step 4
63     for(int i = 0; i < nvar; ++i)
64         xtmp[i] = x[i] + h * dxdt3[i];
65     std::vector<double> dxdt4(nvar);
66     rhs(t + h, xtmp, dxdt4);
67
68     // advance solution
69     for(int i = 0; i < nvar; ++i)
70         x[i] += h * (dxdt1[i] + 2.0 * dxdt2[i] + 2.0 * dxdt3[i] + dxdt4[i]) / 6.0;
71     t += h;
72 }
73
```

```
//*** function main() ********************************

int main()
{
    try {
        // open data file
        std::ofstream ofs("data.csv");
        if(!ofs.is_open())
            throw std::runtime_error("unable to open file.\n");

        for(int it = 0; it < 80; ++it) {
            // set initial conditions
            std::vector<double> x(nvar), y(nvar), z(nvar);
            x[0] = y[0] = 1.0;
            x[1] = y[1] = 0.0;

            // run Euler and Runge-Kutta integration in parallel
            double tx, ty;
            for(tx = 0.0, ty = 0.0; tx < tEnd; ) {
                // obtain numerical solutions
                eulerStepper(tx, x, dt);
                rungeKuttaStepper(ty, y, dt);
            }

            // compute true solution
            const double    delta = 0.5 * epsilon;  // decay rate
            const double    omega = sqrt(b - delta * delta);
            z[0] = exp(-delta * tx) * (cos(omega * tx) +
                                       delta * sin(omega * tx) / omega);

            // quantify approximation error
            double errx = fabs(x[0] - z[0]);
            double erry = fabs(y[0] - z[0]);

            std::cout   << it << '\n'
                        << "step size           = " << dt << '\n'
                        << "error (Euler)       = " << errx << '\n'
                        << "error (Runge-Kutta) = " << erry << '\n';
            ofs << dt << ',' << errx << ',' << erry << '\n';
            dt *= 0.9;
        }
    }
    catch(std::exception &error) {
        std::cerr << "error: " << error.what();
        exit(EXIT_FAILURE);
    }
    return 0;
}
```

Exercise 16.3. Bistability in gene-regulation. The program listed below produces a list of the critical initial mRNA concentrations for a ranging from 0.2 to 1.2 in steps of 0.05. For $b = 0.5$, the active state exists only if $a \leq 1$ explaining why the program throws an exception at $a = 1.05$.

```
0.2 0.0769713
0.25 0.104286
0.3 0.134928
0.35 0.169041
0.4 0.206831
0.45 0.248566
0.5 0.294595
0.55 0.345375
0.6 0.401506
0.65 0.463792
0.7 0.53335
0.75 0.611792
```

```
0.8 0.701593
0.85 0.806913
0.9 0.935834
0.95 1.10912
1 1.54297
1.05 error: unable to reach active state.
```

```
#include <iostream>
#include <fstream>
#include <vector>
#include <cmath>
#include <exception>
#include <cstdlib>

//*** parameters of the integration algorithm *********

const int       nvar  = 2;       // number of variables
const double    tMax  = 1000.0; // max simulation time
const double    dt    = 0.001;  // step size

//*** model specification ****************************

double          a = 0.5;         // degradation rate of protein
const double    b = 0.5;         // degradation rate of mRNA

void rhs(const double &t, const std::vector<double> &x, std::vector<double> &dxdt)
// definition of the righ-hand side of the ODE model
// t is the current time
// vector x contains the current values of the variables
// vector dxdt is filled with the values of the derivatives dx/dt
{
    dxdt[0] = - a * x[0] + x[1];
    double aux = x[0] * x[0];
    dxdt[1] = aux / (1.0 + aux) - b * x[1];
}

//*** ODE integration routine ************************

double rungeKuttaStepper(double &t, std::vector<double> &x, const double &h)
{
    // step 1
    std::vector<double> dxdt1(nvar);
    rhs(t, x, dxdt1);

    // step 2
    std::vector<double> xtmp(nvar);
    for(int i = 0; i < nvar; ++i)
        xtmp[i] = x[i] + 0.5 * h * dxdt1[i];
    std::vector<double> dxdt2(nvar);
    rhs(t + 0.5 * h, xtmp, dxdt2);

    // step 3
    for(int i = 0; i < nvar; ++i)
        xtmp[i] = x[i] + 0.5 * h * dxdt2[i];
    std::vector<double> dxdt3(nvar);
    rhs(t + 0.5 * h, xtmp, dxdt3);

    // step 4
    for(int i = 0; i < nvar; ++i)
        xtmp[i] = x[i] + h * dxdt3[i];
    std::vector<double> dxdt4(nvar);
    rhs(t + h, xtmp, dxdt4);

    // compute mean-square-displacement and advance solution
    double msd = 0.0;
```

```
    for(int i = 0; i < nvar; ++i) {
        const double dxi = (dxdt1[i] + 2.0 * dxdt2[i] + 2.0 * dxdt3[i] + dxdt4[i]) / 6.0;
        x[i] += h * dxi;
        msd += dxi * dxi;
    }
    t += h;
    return msd / nvar;
}

//*** other functions ********************************

bool integrate(std::vector<double> &x)
{
    // (re)set initial protein concentration to zero
    x[0] = 0.0;

    const double epsilon = 1.0e-10;

    for(double t = 0.0; t < tMax; ) {
        const double msd = rungeKuttaStepper(t, x, dt);

        // stop when system reaches equilibrium
        if(msd < epsilon)
            return (x[0] >= 0.5 / (a * b));
    }
    throw std::runtime_error("slow convergence to equilibrium.\n");
    return false;
}

double locateThreshold(double ymax)
{
    const double    epsilon = 1.0e-8;   // desired precision
    const int       maxIt   = 100;      // maximum number of bisection steps

    std::vector<double> x(nvar);

    // verify that the active state is reached from y = ymax
    x[1] = ymax;
    if(!integrate(x))
        throw std::logic_error("unable to reach active state.\n");

    // start bisection loop
    double ymin = 0.0;
    for(int it = 0; it < maxIt; ++it) {
        double ymid = (ymin + ymax) / 2;
        if(ymax - ymin < epsilon)
            return ymid;
        x[1] = ymid;
        if(integrate(x))
            ymax = ymid;
        else
            ymin = ymid;
    }
    throw std::runtime_error("too many iterations in locateThreshold().\n");
}

//*** function main() ********************************

int main()
{
    try {
        for(a = 0.2; a <= 1.2; a += 0.05)
            std::cout << a << ' ' << locateThreshold(2.0) << '\n';
    }
    catch(std::exception &error) {
        std::cerr << "error: " << error.what();
```

```
125         exit(EXIT_FAILURE);
126     }
127     return 0;
128 }
```

Exercise 16.5. Making sense of adaptive step-size control.

(a) Using the definitions of the coefficients for Heun's method (a and b correspond, respectively, to `dxdt1` and `dxdt2`)

$$x_{\text{Euler}}(t+h) = x(t) + h\,a$$

and

$$x_{\text{Heun}}(t+h) = x(t) + \frac{h}{2}\,(a+b)$$

Accordingly,

$$\begin{aligned}\epsilon_{\text{Euler}} &\approx x_{\text{Euler}}(t+h) - x_{\text{Heun}}(t+h)\\ &= (x(t) + h\,a) - (x(t) + \frac{h}{2}\,(a+b))\\ &= \frac{h}{2}(a-b)\end{aligned}$$

In the C^{++} code, the error is calculated from the gradients rather than by measuring the difference between the function values x_{Euler} and x_{Heun}, because the latter method is far too sensitive to floating-point roundoff errors. These could easily occur because the gradients can be very small relative to $x(t)$.

(b) This is when the state of the pendulum changes relatively slowly.

(c) Because the average rate of change of the variables decreases towards the end of the simulation.

Exercise 16.6. Runge-Kutta integration with adaptive step-size control.

Code listing 16.4: Adaptive step-size control (Bogacki-Shampine)

```
1  #include <iostream>
2  #include <fstream>
3  #include <vector>
4  #include <cmath>
5  #include <exception>
6  #include <cstdlib>
7
8  //*** parameters of the integration algorithm *********
9
10 const int       nvar        = 2;        // number of variables
11 const double    tEnd        = 1.0;      // max simulation time
12 const double    dt0         = 0.001;    // initial step size
13 const double    dtsav       = 0.01;     // time interval between data points
14 const double    tolerance   = 1.0e-4;   // acceptable local error during numerical integration
15
16 //*** model specification *****************************
17
18 const double    epsilon = 4.0;  // normalised coefficient of friction
19 const double    b = 1000.0;     // normalised spring constant
20
21 void rhs(const double &t, const std::vector<double> &x, std::vector<double> &dxdt)
22 // definition of the righ-hand side of the ODE model
23 // t is the current time
24 // vector x contains the current values of the variables
25 // vector dxdt is filled with the values of the derivatives dx/dt
26 {
27     dxdt[0] = x[1];
28     dxdt[1] = -epsilon * x[1] - b * x[0];
```

```
}

//*** ODE integration routine ************************

const double kdShrinkMax    = 0.1;      // decrease step size by no more than this factor
const double kdGrowMax      = 5.0;      // increase step size by no more than this factor
const double kdSafety       = 0.9;      // safety factor in adaptive stepsize control
const double kdMinH         = 1.0e-6;   // minimum step size

bool bogackiShampineAdaptiveStepper(double &t, std::vector<double> &x, std::vector<double>
    &dxdt, double &h)
{
    // step 1: gradient is passed as an argument to the function

    // step 2
    std::vector<double> xtmp(nvar);
    for(int i = 0; i < nvar; ++i)
        xtmp[i] = x[i] + 0.5 * h * dxdt[i];
    std::vector<double> dxdt2(nvar);
    rhs(t + 0.5 * h, xtmp, dxdt2);

    // step 3
    for(int i = 0; i < nvar; ++i)
        xtmp[i] = x[i] + 0.75 * h * dxdt2[i];
    std::vector<double> dxdt3(nvar);
    rhs(t + 0.75 * h, xtmp, dxdt3);

    // step 4
    for(int i = 0; i < nvar; ++i)
        xtmp[i] = x[i] + h * (2.0 * dxdt[i] + 3.0 * dxdt2[i] + 4.0 * dxdt3[i]) / 9.0;
    std::vector<double> dxdt4(nvar);
    rhs(t + h, xtmp, dxdt4);

    double errMax = 0.0;
    for(int i = 0; i < nvar; ++i) {
        // compute error
        double erri = fabs(h * (5.0 * dxdt[i] / 72.0 - dxdt2[i] / 12.0 - dxdt3[i] / 9.0 + 0.125
    * dxdt4[i])) / tolerance;
        if(erri > errMax)
            errMax = erri;
    }

    // adjust step size
    const double fct = errMax > 0.0 ? kdSafety / pow(errMax, 1.0 / 3.0) : kdGrowMax;
    if(errMax > 1.0) {
        // reduce step size and reject step
        if(fct < kdShrinkMax)
            h *=  kdShrinkMax;
        else
            h *=  fct;
        if(h < kdMinH)
            throw std::runtime_error("step size underflow in eulerHeunAdaptiveStepper().");
        return false;
    }
    else {
        // update solution and increase step size
        x = xtmp;
        dxdt = dxdt4;
        t += h;
        if(fct > kdGrowMax)
            h *=  kdGrowMax;
        else
            h *=  fct;
        return true;
    }
}
```

```
93
94 //*** function main() ********************************
95
96 int main()
97 {
98     try {
99         // open data file
100        std::ofstream ofs("data.csv");
101        if(!ofs.is_open())
102            throw std::runtime_error("unable to open file.\n");
103
104        // set initial conditions and compute initial gradient
105        std::vector<double> x(nvar);
106        x[0] = 1.0;
107        x[1] = 0.0;
108        std::vector<double> dxdt(nvar);
109        rhs(0.0, x, dxdt);
110
111        // start numerical integration
112        int nOK = 0, nStep = 0;
113        double dtMin = dt0, dtMax = kdMinH;
114        for(double t = 0.0, tsav = 0.0, dt = dt0; t < tEnd; ++nStep) {
115            if(bogackiShampineAdaptiveStepper(t, x, dxdt, dt))
116                ++nOK;
117
118            if(dt < dtMin)
119                dtMin = dt;
120            else if(dt > dtMax)
121                dtMax = dt;
122
123            if(t > tsav) {
124                ofs         << t << ',' << x[0] << ',' << x[1] << ',' << dt << '\n';
125                tsav += dtsav;
126            }
127        }
128        ofs.close();
129
130        // report integration data
131        std::cout   << "integration complete.\n"
132                    << "number of steps : " << nStep << '\n'
133                    << "proportion bad steps : " << 1.0 - nOK * 1.0 / nStep << '\n'
134                    << "average step size : " << tEnd / nStep << '\n'
135                    << "min step size : " << dtMin << '\n'
136                    << "max step size : " << dtMax << '\n';
137    }
138    catch(std::exception &error) {
139        std::cerr << "error: " << error.what();
140        exit(EXIT_FAILURE);
141    }
142    return 0;
143 }
```

The screen output (shown below) can be directly compared to that of the Euler-Heun scheme in Listing 16.3. Note the substantial reduction in the number of steps that is needed by the Bogacki-Shampine method to complete the integration.

```
integration complete.
number of steps : 437
proportion bad steps : 0.0686499
average step size : 0.00228833
min step size : 0.001
max step size : 0.00894345
```

The implementation in 16.4 makes use of the fact that the first gradient computed during an integration step is equal to the last gradient computed during the previous step. Therefore, the algorithm must be supplied with the gradient at the initial point to start the integration (see on line 84, 105-109 and 115 how the gradient at the first step is initialised and updated by the algorithm).

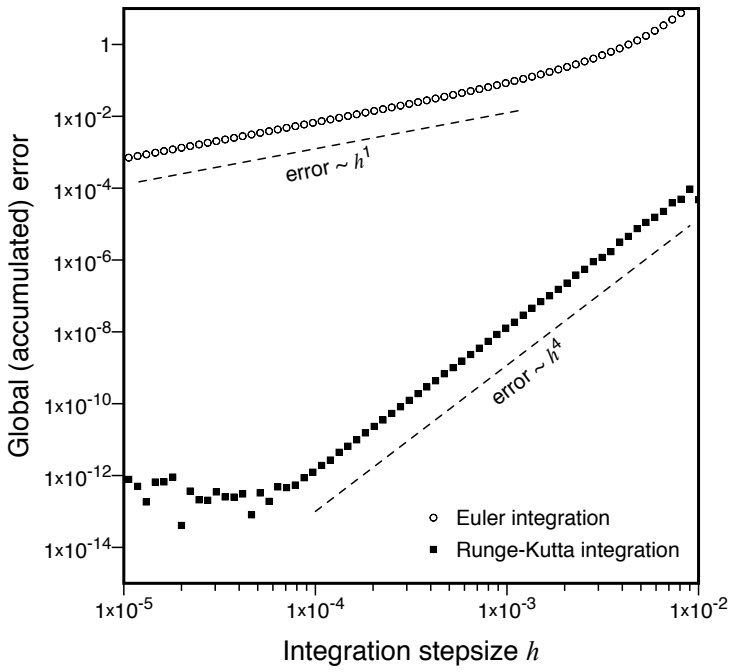


Fig. 16.1: Relationship between integration error and integration stepsize for Euler and Runge-Kutta integration.

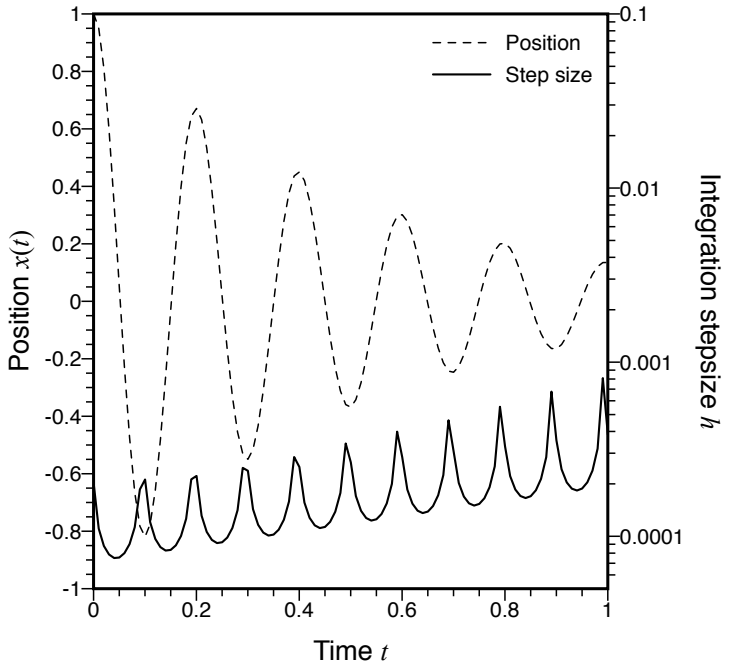


Fig. 16.2: Step size control by an adaptive Heun-Euler integration routine in a simulation of a damped pendulum.

Module 17

Stochastic models

Jamie explains what's on the menu

In this module you will:

- Become familiar with methods for generating pseudo-random numbers.
- Learn how to implement stochastic models based on the Gillespie algorithm.

17.1 Pseudo-random numbers

Computers, which follow a fixed recipe for obtaining a result, are very well suited to simulate deterministic dynamical models. However, not all models are deterministic. Randomness plays a role in many biological processes (e.g., think of gene-expression noise, genetic drift, demographic stochasticity and noise in signalling systems), so we also need computational methods for analysing stochastic models. This requires that we develop algorithms with probabilistic outcomes, in order to capture the effect of chance events.

It is not obvious how to generate real randomness by means of a computer algorithm, but interestingly, it is possible to generate sequences of numbers that, for all practical purposes, look like they have been sampled from a random distribution. The statistical properties of these numbers are indistinguishable from random numbers drawn from a specified distribution, but, unlike real random numbers, they are predictable (if we have complete information about the algorithm used for constructing them). For this reason, computer generated 'random' numbers are called *pseudo-random*, and the routines used for generating them are called *pseudo-random number generators* (PRNGs).

An example of a PRNG is the C-library function `rand()`, which is defined in the header `cstdlib`. This function returns an integer from a discrete (uniform) distribution in the range $[0, \texttt{RAND\_MAX})$, where `RAND_MAX` is a number defined in `cstdlib` that is no smaller than 32767. The code below uses `rand()` to simulate the outcome of rolling a dice thirty times

```
1 #include <iostream>
2 #include <cstdlib>
3
```

```
int main()
{
    for(int i = 0; i < 30; ++i)
        std::cout << 1 + rand() % 6 << ' ';

    return 0;
}
```

The output of the program (when it is compiled on Master Yoda's laptop) is:

```
2 2 6 3 5 3 1 3 6 2 1 6 1 3 4 6 2 2 5 5 4 2 4 6 5 4 4 1 3 2
```

The result appears to mimic the behaviour of a dice quite well, and there is no obvious pattern in the sequence of numbers generated by the program. However, the result appears much less random if the program is executed again, and it clearly does not behave like we would expect when we would throw a dice another 30 times. In particular, the second program run produces:

```
2 2 6 3 5 3 1 3 6 2 1 6 1 3 4 6 2 2 5 5 4 2 4 6 5 4 4 1 3 2
```

That is, exactly the same sequence!

PRNGs are capable of generating many different pseudo-random number sequences, but the algorithm will always produce the same default sequence, if we do not explicitly instruct it to pick another one. The way to select another sequence, is to provide the `rand()` with a seed by passing a value to the function `srand()`. If we change the seed, we will get another sequence. Sometimes we want to set the seed explicitly (for example, to repeat a previous simulation), but often it is more convenient to derive the seed from the system clock, so that the PRNG will automatically generate a different sequence every time the program is started anew. The code below shows how this can be done using the clock functions from the library `chrono`.

```
#include <iostream>
#include <cstdlib>
#include <chrono>

int main()
{
    // obtain seed from system clock
    std::chrono::high_resolution_clock::time_point tp =
    std::chrono::high_resolution_clock::now();
    unsigned seed = static_cast<unsigned>(tp.time_since_epoch().count());

    // seed pseudo-random number generator
    std::clog << "random seed : " << seed << '\n';
    srand(seed);

    // simulate dice throws
    for(int i = 0; i < 30; ++i)
        std::cout << 1 + rand() % 6 << ' ';

    return 0;
}
```

Two different runs of the program now produce the ouput:

```
random seed : 1651618181
4 6 3 1 4 6 6 1 4 6 4 1 2 3 4 5 4 4 4 1 1 5 5 4 1 5 1 4 6 4
```

and

```
random seed : 2248923769
6 6 3 6 2 5 4 6 3 6 4 1 1 2 1 5 4 3 5 3 3 6 3 4 3 2 4 6 2 2
```

***Master Yoda* on random seeds**

When you use the system clock to initialise a PRNG, consider storing the seed in a log file or in an output file with the rest of your simulation results. That way, you can always repeat the same simulation exactly. This can be helpful if you need to collect additional data from a specific simulation or correct errors.

***Jamie's recipes for disaster:* relying on `rand()`**

The C-library function `rand()` is intended to provide a quick-and-dirty implementation of a PRNG. Its implementation is different on different platforms, and some of its versions are of rather poor quality. Therefore, you are *strongly discouraged* to rely on `rand()` for the generation of pseudo-random numbers in scientific applications.

17.2 The library `random`

A proper alternative to `rand()` is provided by the PRNGs in the C++ library `random`. Unfortunately, working with objects from this library can be a bit intimidating at first. In this section, we will therefore illustrate a few common applications to help you get used to the framework and the notation. After studying these examples and experimenting with them, you will be ready to modify them to fit your needs.

The library `random` produces random numbers using combinations of so-called *generator* and *distribution* objects. Generators are objects that generate uniformly distributed numbers; distributions transform sequences of numbers generated by a generator into sequences of numbers that follow a specific random variable distribution (e.g., poisson, normal or binomial). The following code fragment shows how to declare a generator and a distribution object

```
1 #include <random>
2
3 std::mt19937_64 rng;
4 std::uniform_int_distribution<int> dice(1,6);
```

The generator is of type `std::mt19937_64`, which is one of the PRNGs implemented in the library `random`. This particular one represents a 64 bit implementation of the Mersenne Twister algorithm for generating pseudo-random numbers, which is a relatively fast and high-quality algorithm suitable for scientific applications. There are several other PRNGs you can choose from; consult the online documentation of the `random` library for more information.

The distribution object (which we named `dice`) is of type `std::uniform_int_distribution`, which represents a uniform discrete distribution. The template parameter `<int>` is to specify that the numbers drawn from the distribution are normal integers, and the arguments that are supplied at initialisation specify that this particular uniform discrete distribution should contain the integers $i = 1, 2, \ldots, 6$.

To generate a random number, we need to call the distribution object as if it were a function, and supply the generator as an argument. That is, `dice(rng)` will return a random integer between 1 and 6, as illustrated on line 20 in code listing 17.1. Note that this program replicates the earlier implementation using `rand()`:

Code listing 17.1: Drawing integers from a uniform distribution

```
1 #include <iostream>
2 #include <chrono>
3 #include <random>
4
5 int main()
6 {
7     std::mt19937_64 rng;
```

```
8     std::uniform_int_distribution<int> dice(1,6);
9
10    // obtain seed from system clock
11    std::chrono::high_resolution_clock::time_point tp =
      std::chrono::high_resolution_clock::now();
12    unsigned seed = static_cast<unsigned>(tp.time_since_epoch().count());
13
14    // seed pseudo-random number generator
15    std::clog << "random seed : " << seed << '\n';
16    rng.seed(seed);
17
18    // simulate dice throws
19    for(int i = 0; i < 30; ++i)
20        std::cout << dice(rng) << ' ';
21
22    return 0;
23 }
```

The statement on line 16 shows how to seed the PRNG object `rng`. The class `std::mt19937_64` has a member function `seed()` that accepts an **`unsigned int`** to serve as a random seed.

Sampling from standard statistical distributions

Distribution objects of many different types are available to draw random numbers from other statistical distributions. Some of these are illustrated in Listing 17.2. With these examples you should be able to work out how to draw random numbers from other distributions available in `random`. A full list of the available distributions and worked-out examples are available on http://www.cplusplus.com/reference/random/.

Code listing 17.2: Sampling from different standard statistical distributions

```
1  #include <iostream>
2  #include <chrono>
3  #include <random>
4
5  int main()
6  {
7     // obtain seed from system clock
8     std::chrono::high_resolution_clock::time_point tp =
      std::chrono::high_resolution_clock::now();
9     unsigned seed = static_cast<unsigned>(tp.time_since_epoch().count());
10
11    // create and seed pseudo-random number generator
12    std::mt19937_64 rng;
13    std::clog << "random seed : " << seed << '\n';
14    rng.seed(seed);
15
16    //*** examples of discrete distributions ******************************************
17
18    // draw number from Bernoulli distribution
19    const double p = 0.55;
20    std::bernoulli_distribution biasedCoinFlip(p);
21    std::cout   << "(1) flip a biased coin (Pr[heads] = " << p << ")\n"
22                << "    heads : ";
23    if(biasedCoinFlip(rng))
24        std::cout << "yes\n";
25    else
26        std::cout << "no\n";
27
28    // draw number from binomial distribution
29    int n = 100;
30    std::binomial_distribution<int> nBiasedCoinFlips(n, p);
31    std::cout   << "(2) flip a biased coin (Pr[heads] = " << p << ") " << n << " times\n"
32                << "    #heads : " << nBiasedCoinFlips(rng) << '\n';
33
```

```
34      // draw number from poisson distribution
35      double mu = 2.5;
36      std::poisson_distribution<int> poisson(mu);
37      std::cout   << "(3) draw from Poisson distribution with mu = " << mu << '\n'
38                  << "    observation : " << poisson(rng) << '\n';
39
40      //*** examples of continuous distributions ***************************************
41
42      // draw number from uniform distribution
43      double a = 0.0, b = 1.0;
44      std::uniform_real_distribution<double> uniform(0.0, 1.0);
45      std::cout   << "(4) draw from continuous uniform distribution on [" << a << ',' << b <<
        ")\n"
46                  << "    observation : " << uniform(rng) << '\n';
47
48      // draw number from exponential distribution
49      double rate = 10.5;
50      std::exponential_distribution<double> exponential(rate);
51      std::cout   << "(5) draw from exponential distribution with rate = " << rate << '\n'
52                  << "    observation : " << exponential(rng) << '\n';
53
54      // draw number from normal distribution
55      double mean = 0.5;
56      double standardDeviation = 0.1;
57      std::normal_distribution<double> normal(mean, standardDeviation);
58      std::cout   << "(6) draw from normal distribution with mean = " << mean
59                  << " and standard deviation " << standardDeviation << '\n'
60                  << "    observation : " << normal(rng) << '\n';
61
62      return 0;
63  }
```

The output of the program (i.e., one specific realisation of it) is:

```
random seed : 423402986
(1) flip a biased coin (Pr[heads] = 0.55)
    heads : yes
(2) flip a biased coin (Pr[heads] = 0.55) 100 times
    #heads : 54
(3) draw from Poisson distribution with mu = 2.5
    observation : 2
(4) draw from continuous uniform distribution on [0,1)
    observation : 0.611137
(5) draw from exponential distribution with rate = 10.5
    observation : 0.115046
(6) draw from normal distribution with mean = 0.5 and standard deviation 0.1
    observation : 0.515792
```

Implementing a weighted lottery

It also happens regularly that we need to sample from a set of different, mutually exclusive options, that should each occur with a certain probability. The library `random` contains a `std::discrete_distribution` class that allows you to deal with such a 'weighted-lottery'. To sample from a `std::discrete_distribution`, first create a `std::vector` that contains the probability weights $(w_0, w_1, \ldots, w_{n-1})$ for the different options in the weighted lottery. After using this vector to initialise a `std::discrete_distribution` (see line 41 in Listing 17.3 below), you can can sample from the distribution in the usual way (line 45). The probability of sampling option i will then be given by

$$\Pr(i) = \frac{w_i}{\sum_{j=0}^{n-1} w_j}$$

Listing 17.3 shows a worked-out example. The program tracks the densities of three different genotypes in a population of individuals that reproduce clonally. For each genotype, the probability that an individual in the next generation belongs to that genotype is proportional the genotype frequency in the current generation and the fitness of the genotype (see line 36). The simulation continues until one of the genotypes reaches fixation.

Code listing 17.3: Implementation of a weighted lottery

```
#include <iostream>
#include <vector>
#include <chrono>
#include <random>

int main()
{
    void show(int, const std::vector<int>&);

    // obtain seed from system clock
    std::chrono::high_resolution_clock::time_point tp =
    std::chrono::high_resolution_clock::now();
    unsigned seed = static_cast<unsigned>(tp.time_since_epoch().count());

    // create and seed pseudo-random number generator
    std::mt19937_64 rng;
    std::clog << "random seed : " << seed << '\n';
    rng.seed(seed);

    // set relative fitness values
    const int iNrGenotype = 3;
    std::vector<double> vecFitness(iNrGenotype, 1.0);
    vecFitness[2] = 0.8;

    // initialise population
    const int iPopulationSize = 30;
    std::vector<int> vecNrIndividuals(iNrGenotype, iPopulationSize / iNrGenotype);

    // iterate model until one genotype reaches fixation
    for(int it = 0;;) {
        // write output to screen
        show(it, vecNrIndividuals);

        // calculate weights for std::discrete_distribution
        std::vector<double> vecWeights(iNrGenotype);
        for(int i = 0; i < iNrGenotype; ++i) {
            vecWeights[i] = vecNrIndividuals[i] * vecFitness[i];
            vecNrIndividuals[i] = 0;
        }

        // load probability weights into a std::discrete_distribution
        std::discrete_distribution<int> weightedLottery(vecWeights.begin(), vecWeights.end());

        // sample genotype distribution in next generation
        for(int k = 0; k < iPopulationSize; ++k) {
            int j = weightedLottery(rng);
            ++vecNrIndividuals[j];
        }
        ++it;

        // terminate iteration if a genotype reaches fixation
        bool stop = false;
        for(int i = 0; i < iNrGenotype; ++i)
            if(vecNrIndividuals[i] == iPopulationSize) {
                stop = true;
                show(it, vecNrIndividuals);
                break;
            }
        if(stop)
            break;
    }
    return 0;
}

```

```
64 void show(int it, const std::vector<int> &vec)
65 {
66     std::cout << it;
67     for(int i = 0; i < vec.size(); ++i)
68         std::cout << '\t' << vec[i];
69     std::cout << '\n';
70 }
```

17.3 The Gillespie algorithm for stochastic simulations

Stochastic effects are important in all kinds of biological contexts. In meta-populations, for example, local demes can go extinct as a result of demographic stochasticity (i.e., random fluctuations in birth and death rates) or external events. Gene regulation, transcription and protein synthesis can also be highly stochastic, especially when transcription factors and mRNA occur in low copy numbers within a cell. A general method to simulate these and other stochastic biological processes is the Gillespie algorithm, which was originally developed for modelling chemical reactions.

To illustrate an application of the Gillespie algorithm, consider the competition between two species, A and B, in a meta-population consisting of N small demes, or patches. For simplicity, assume that each patch can be in three discrete states: empty, occupied by species A or occupied by species B. Occupied patches can go extinct, and empty patches can be colonised by either species A or species B. Moreover, to make the model more interesting, we assume that species A is a superior competitor: individuals of species A can colonise patches currently occupied by species B, and when this happens, the original occupants of the patch are displaced. The dynamics of the two species can now be described as a process resulting from 5 'reactions':

1. A patch occupied by species A can go extinct. This process can be represented by a reaction

$$A \xrightarrow{e_\mathrm{A}} E$$

   that occurs at rate $v_1 = e_\mathrm{A}\,[A]$. Here $[A]$ is the current density, or 'concentration' of patches currently occupied by species A, and e_A is the extinction rate of species A.

2. A patch occupied by species B can go extinct. This process can be represented by a reaction

$$B \xrightarrow{e_\mathrm{B}} E$$

   that occurs at rate $v_2 = e_\mathrm{B}\,[B]$. Here $[B]$ is the current density, or 'concentration' of patches currently occupied by species B, and e_B is the extinction rate of this species.

3. An empty patch can be colonised by migrants dispersing from a patch occupied by species A. The corresponding reaction

$$E + A \xrightarrow{d_\mathrm{A}} A + A$$

   occurs at rate $v_3 = d_\mathrm{A}\,[A]\,[E]/N$, where $[E]/N$ is the current fraction of empty patches and d_A is the dispersal rate of species A.

4. An empty patch can be colonised by migrants dispersing from a patch occupied by species B. The corresponding reaction

$$E + B \xrightarrow{d_\mathrm{B}} B + B$$

   occurs at rate $v_4 = d_\mathrm{B}\,[B]\,[E]/N$, where d_B is the dispersal rate of species B.

5. Migrants from a patch occupied by species A can arrive at a patch occupied by species B and outcompete the original inhabitants. This reaction,

$$A + B \xrightarrow{d_\mathrm{A}} A + A$$

   occurs at rate $v_5 = d_\mathrm{A}\,[A]\,[B]/N$ (in the worst-case scenario that the establishment of species A is completely unaffected by the presence of species B).

Given the current state of the metapopulation, summarised by the variables $[A]$, $[B]$ and $[E]$, it is now possible to calculate the rates of each of the five events that may occur. What do these rates $v_1, v_2, \ldots, v_5$ tell us exactly? Well, first of all, an event that occurs at a high rate, is likely to occur sooner than another

event that occurs at a low rate. Therefore, the event that occurs next is more likely to be one that occurs at a high rate than one that occurs at a low rate. Second, if the events occur independently of each other, the time that we have to wait for any event to occur simply depends on the sum of the rates of the individual events. If this total rate of events is large, it is likely that one of the events will happen soon; if the total rate of events is small, we may have to wait for a long time for any of the events to occur.

The Gillespie algorithm formalises these ideas in the following recipe for updating the state of the metapopulation:

1. Given the current densities of different patch types ($[E]$, $[A]$ and $[B]$), calculate the five reaction rates $v_1, v_2, \ldots, v_5$ as defined above.
2. Compute the total rate of events $v_{\text{tot}} = \sum_{i=1}^{5} v_i$.
3. Draw the waiting time to the next event from an exponential distribution with rate parameter v_{tot}. Let this waiting time be denoted by dt.
4. The next event could be any of the five possible events; to determine which one of the five it is, draw the event using a weighted lottery scheme with probabilities proportional to v_i. Accordingly, events that occur at a high rate, are more likely to be the next event.
5. Update the state of the population in accordance with the event that occurs next.
6. Increment the time by the time interval dt and return to step 1, unless the end of the simulation has been reached.

Listing 17.4 shows an implementation of these steps in a C^{++} program. Since the above recipe can be generalised to an arbitrary number of reactions, this code can be used as a starting point for developing stochastic models in other contexts. The important model-specific parts of the code that have to be adjusted are line 46-51, where the rates of the different reactions are calculated from the current values of the state variables, and line 68-90, where these value are updated in accordance with the event that was sampled.

Code listing 17.4: The Gillespie algorithm

```
1  #include <iostream>
2  #include <fstream>
3  #include <vector>
4  #include <exception>
5  #include <cstdlib>
6  #include <random>
7  #include <chrono>
8
9  //*** model parameters ******************************
10
11 const int nEvents   =   5;          // number of events
12 const int nTotal    =   200;        // total number of patches
13 const double eA     =   0.2;        // extinction rate for species A
14 const double eB     =   0.2;        // extinction rate for species B
15 const double dA     =   0.3;        // dispersal rate for species A
16 const double dB     =   0.9;        // dispersal rate for species B
17 const double tEnd   =   500.0;      // end of simulation
18 const double dtSav  =   1.0;        // time interval between output data points
19
20 //*** simulation program ****************************
21
22 int main()
23 {
24     try {
25         // obtain seed from system clock
26         std::chrono::high_resolution_clock::time_point tp =
       std::chrono::high_resolution_clock::now();
27         unsigned seed = static_cast<unsigned>(tp.time_since_epoch().count());
28
29         // create and seed pseudo-random number generator
30         std::mt19937_64 rng;
```

```
        std::clog << "random seed : " << seed << '\n';
        rng.seed(seed);

        // open data file
        std::ofstream ofs("data.csv");
        if(!ofs.is_open())
            throw std::runtime_error("unable to open file.\n");

        // initialise metapopulation
        int nA = 10, nB = 10;
        int nE = nTotal - nA - nB;

        // enter simulation loop
        for(double t = 0.0, tsav = 0.0; t < tEnd && nE < nTotal;) {
            // calculate rates for the different events
            std::vector<double> vecRates(nEvents);
            vecRates[0] = eA * nA;                  // patch occupied by A goes extinct
            vecRates[1] = eB * nB;                  // patch occupied by B goes extinct
            vecRates[2] = (dA * nA * nE) / nTotal;  // A colonises empty patch
            vecRates[3] = (dB * nB * nE) / nTotal;  // B colonises empty patch
            vecRates[4] = (dA * nA * nB) / nTotal;  // A outcompetes B

            // determine waiting time to next event
            double sum = 0.0;
            for(int i = 0; i < nEvents; ++i)
                sum += vecRates[i];
            if(sum <= 0.0)
                throw std::runtime_error("unable to draw waiting time.\n");
            std::exponential_distribution<double> waitingTime(sum);
            const double dt = waitingTime(rng);

            // sample next event
            std::discrete_distribution<int> drawEvent(vecRates.begin(), vecRates.end());
            const int event = drawEvent(rng);

            // update population state
            t += dt;
            switch(event) {
                case 0:          // patch occupied by A goes extinct
                    --nA;
                    ++nE;
                    break;
                case 1:          // patch occupied by B goes extinct
                    --nB;
                    ++nE;
                    break;
                case 2:          // A colonises empty patch
                    ++nA;
                    --nE;
                    break;
                case 3:          // B colonises empty patch
                    ++nB;
                    --nE;
                    break;
                case 4:          // A outcompetes B
                    ++nA;
                    --nB;
                    break;
                default: throw std::logic_error("unknown event");
            }

            // generate output
            if(t > tsav) {
                ofs << t << ',' << nA << ',' << nB << ',' << nE << '\n';
                std::cout << t << ' ' << nA << ' ' << nB << ' ' << nE << '\n';
                tsav += dtSav;
```

```
            }
        }
    }
    catch(std::exception &error) {
        std::cerr << error.what();
        exit(EXIT_FAILURE);
    }
    return EXIT_SUCCESS;
}
```

In a well-mixed, deterministic system, species A would quickly outcompete species B, because A is a superior competitor. However, in a meta-population, long-term survival does not only depend on competitive strength, but also on the ability of a species to colonise empty patches. Fig. 17.1, which shows results produced by the simulation program in Listing 17.4, illustrates that differences in dispersal success and a high rate of extinction can actually tip the balance in favour of species B. In summary, if species A is a superior competitor, but species B is a better disperser, then only A will persist in the meta-population if the rate of extinction is low (Fig. 17.1(a)). At very high extinction rates, both species will go extinct, but there is also a range of extinction rates at which B can persist but A cannot (Fig. 17.1(c)). Finally, for a range of intermediate extinction rates, the two species can coexist in the meta-population (Fig. 17.1(b)). Models like this provide theoretical support for the so-called *intermediate disturbance hypothesis*, which posits that local species diversity is maximised when ecological disturbance is neither too rare nor too frequent.

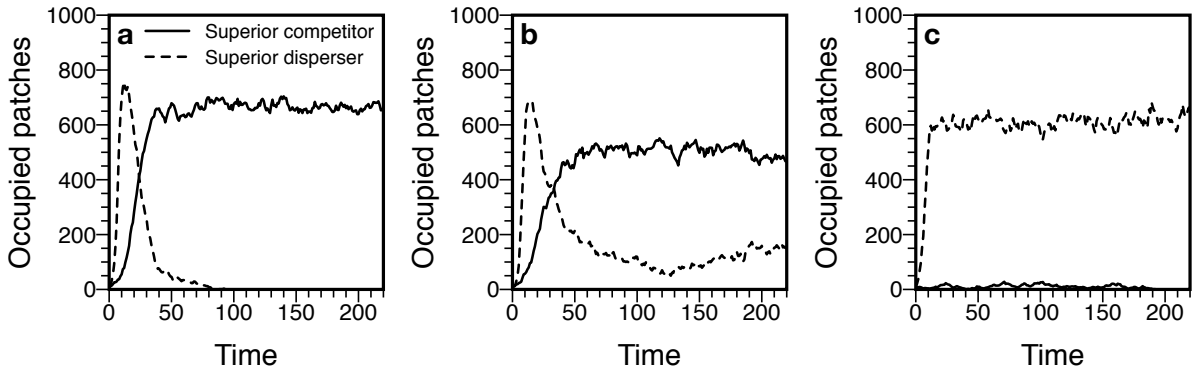


Fig. 17.1: Metapopulation dynamics of two competing species simulated by the Gillespie algorithm of Listing 17.4. (a) $e_A = e_B = 0.1$**; (b)** $e_A = e_B = 0.15$**; (c)** $e_A = e_B = 0.3$**. Other parameters:** $d_A = 0.3$**,** $d_B = 0.8$ **and** $N = 1000$**.**

17.4 Exercises

17.1. Picking a random element from a vector. Write a function that returns the value of a random element of a `std::vector`.

17.2. The shuffle algorithm ($\star$). Write a function that implements Algorithm 4 for rearranging the elements of a vector into a random order.

Data: Vector v with n elements.
Result: A random permutation of v.
for $i \leftarrow n-1$ **to** 1 **do**
  Pick a random element j, such that $0 \leq j \leq i$;
  Swap element v_i with element v_j;
end
return v;

Algorithm 4: Algorithm for rearranging the elements of a vector into a random order.

17.3. Stochastic population growth. Consider the growth of a population as a process that results from three 'reactions':

1. Birth events,
$$A \xrightarrow{b} A + A$$
which occur at rate $v_1 = b\,[A]$. Here, $[A]$ is the current number of individuals, and b is the per-capita birth rate.

2. Externally induced death events,
$$A \xrightarrow{d} \emptyset$$
which occur at rate $v_2 = d\,[A]$, where d is the per-capita death rate.

3. Competition-induced death events,
$$A + A \xrightarrow{d} A$$
which occur at rate $v_3 = c\,[A]^2$, where c is the rate of death due to competitive interaction with another individual.

a Simulate the growth of the population based on the Gillespie algorithm for the parameter values $b = 2$, $d = 1$ and $c = 0.02$. Run the simulation for 50 time units, starting from an initial population of 5 individuals.

b Run several replicate simulations and compare the stochastic growth trajectory of the population to the solution of the logistic growth equation (Eq. 16.2 on p. 224) with $r = b - d$ and $K = (b - d)/c$.

c Repeat the simulation with $c = 0.001$, and compare again with the analytical solution of the logistic equation. Why is the correspondence between the two better than in (b)? What can you say about the relationship between stochastic and deterministic models based on this result?

Test your skill

17.4. Diffusion-limited aggregation (⋆). Many reef corals form intricate branching structures as they grow. This growth occurs slowly, over the course many generations, by the deposition of exoskeleton at the base of each of the polyps in a coral head. The resulting shape of the colony is characteristic of the species, but there are some generalities across species. In particular, the morphology of the colony appears to be shaped by resource competition. This connection has been investigated in more detail using so-called diffusion-limited aggregation (DLA) models. In this exercise, we will implement a stochastic DLA model that can generate structures resembling branching coral colonies (Fig. 17.2).

- To start, create a 150×150 grid of `int` variables, which will indicate whether a particular position is occupied by the coral or if there is a food item there. Think of this grid as a vertical cross-section through the colony. Use cell state = 0 for empty cells, cell state = 1 for cells that contain a food item, and cell state = 2 for cells where the coral has grown. The coral is initially present as a single layer of polyps on the rock surface. To mimic this initial condition, set the lowest row of cells in the grid to cell state 2, and pick 100 other cells at random that are initialised in cell state 1; initialise with cell state 0 everywhere else.

- Write a function to update the state of the grid from one time step to the next. In each step, first collect the current positions of all food items, and shuffle these coordinate pairs, in order to be able to visit the food items in a random sequence (use your solution to Exercise 17.2 to accomplish this). For each food item:

  1. Pick a new position using a probabilistic rule: move down with probability p_{down}; up with probability p_{up}; left with probability p_{left}; right with probability p_{right} and otherwise stay at the same position.
  2. Assume periodic boundary conditions in the horizontal direction (i.e., a food item that leaves the grid on the left, appears again on the right and vice versa), and open boundary conditions at the top (i.e., a food item that leaves the grid at the top, is replaced by another food item that enters at a random empty site somewhere in the top row).
  3. When the cell at the new position of the food item is in state 2 (i.e., a site where coral has grown), the food item is consumed and the coral grows into the cell where the food item used to be.
  4. When the cell at the new position of the food item is empty (state 0), the food item moves to that cell.

5. When the cell at the new position of the food item already contains a food item (state 1), the food item stays at its current position.

- Keep count of how many food items were consumed in each time step, and add new food items in the top row of the system, keeping the total amount of food items constant.
- Continue the simulation until the coral has reached the top row of the grid. Write the state of the grid to a file at the end of the simulation (after replacing all food items by empty cells), and visualise the growth pattern of the coral.
- Experiment with different values for the probabilities of movement.

Fig. 17.2: Coral-like growth patterns generated by diffusion-limited aggregation. Left: $p_{\text{left}} = p_{\text{right}} = 0.2$, $p_{\text{up}} = 0.1$, $p_{\text{down}} = 0.3$; **right :** $p_{\text{left}} = 0.4$, $p_{\text{right}} = 0.3$, $p_{\text{up}} = 0.1$, $p_{\text{down}} = 0.2$.

17.5 Solutions to the exercises

Exercise 17.1. Picking a random element from a vector.

```
#include <random>
#include <vector>

std::mt19937_64 rng;

int pickRandomElement(const std::vector<int> &vec)
{
    std::uniform_int_distribution<int> randomElement(0, vec.size() - 1);
    return vec[randomElement(rng)];
}
```

Exercise 17.2. The shuffle algorithm.

```
#include <random>
#include <vector>

std::mt19937_64 rng;

void shuffle(std::vector<int> &vec)
{
    for(int i = vec.size() - 1; i > 0; --i) {
        std::uniform_int_distribution<int> randomElement(0, i);
        int j = randomElement(rng);

        int tmp = vec[i];
        vec[i] = vec[j];
        vec[j] = tmp;
    }
}
```

Exercise 17.3. Stochastic population growth. Fig. 17.3 shows that the match between the deterministic model and the stochastic simulation improves when the carrying capacity of the population increases. In general, stochastic models tend to converge to their deterministic counterparts when the size of the system increases.

```cpp
#include <iostream>
#include <fstream>
#include <exception>
#include <cstdlib>
#include <random>

//*** model parameters ******************************

const int nEvents   =   3;          // number of events
const double b      =   2.0;        // per-capita birth rate
const double d      =   1.0;        // per-capita death rate
const double c      =   0.001;      // competition-induced death rate
const double tEnd   =   50.0;       // end of simulation
const double dtSav  =   0.5;        // time interval between output data points

//*** simulation program ****************************

int main()
{
    const double r = b - d;
    const double K = r / c;
    try {
        // obtain seed from system clock
        std::chrono::high_resolution_clock::time_point tp =
    std::chrono::high_resolution_clock::now();
        unsigned seed = static_cast<unsigned>(tp.time_since_epoch().count());

        // create and seed pseudo-random number generator
        std::mt19937_64 rng;
        std::clog << "random seed : " << seed << '\n';
        rng.seed(seed);

        // open data file
        std::ofstream ofs("data.csv");
        if(!ofs.is_open())
            throw std::runtime_error("unable to open file.\n");

        // initialise population
        int n = 5;
        double x0 = n;

        // enter simulation loop
        for(double t = 0.0, tsav = 0.0; t < tEnd && n > 0;) {
            // calculate rates for the different events
            std::vector<double> vecRates(nEvents);
            vecRates[0] = b * n;                    // birth
            vecRates[1] = d * n;                    // death by external causes
            vecRates[2] = c * n * n;                // competition-induced death

            // determine waiting time to next event
            double sum = 0.0;
            for(int i = 0; i < nEvents; ++i)
                sum += vecRates[i];
            if(sum <= 0.0)
                throw std::runtime_error("unable to draw waiting time.\n");
            std::exponential_distribution<double> waitingTime(sum);
            const double dt = waitingTime(rng);

            // sample next event
            std::discrete_distribution<int> drawEvent(vecRates.begin(), vecRates.end());
```

```
            const int event = drawEvent(rng);

            // update population state
            t += dt;
            switch(event) {
                case 0:          // birth
                    ++n;
                    break;
                case 1:          // death
                case 2:
                    --n;
                    break;
                default: throw std::logic_error("unknown event");
            }

            // calculate analytical prediction
            const double nPred = x0 * K / ((K - x0) * exp(-r * t) + x0);

            // generate output
            if(t > tsav) {
                ofs << t << ',' << n << ',' << nPred << '\n';
                std::cout << t << ' ' << n << ' ' << nPred << '\n';
                tsav += dtSav;
            }
        }
    }
    catch(std::exception &error) {
        std::cerr << error.what();
        exit(EXIT_FAILURE);
    }
    return EXIT_SUCCESS;
}
```

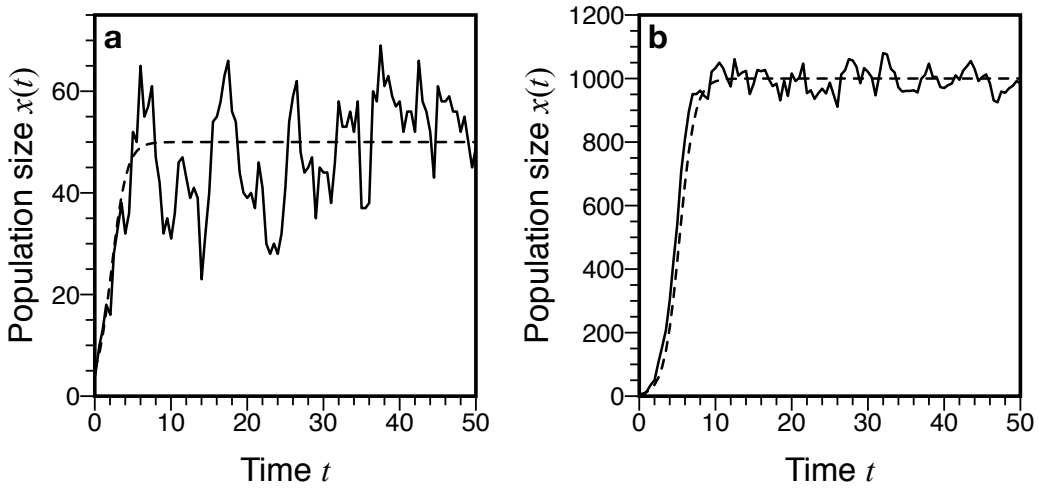


Fig. 17.3: Stochastic simulations (solid) versus a deterministic description of logistic population growth (dashed), for $c = 0.02$ (a) and $c = 0.001$ (b). See exercise 17.3 for details.

Module 18

Elementary individual-based simulations

Jamie explains what's on the menu

In this module you will:

- Be introduced to individual-based simulation.
- See how an individual-based simulation is developed step-by-step, and practice this skill by writing an individual-based simulation program from scratch.

18.1 Individual-Based Simulation (IBS)

In some cases, you may want to model a population of individuals with different ages, who can be in different physiological states, and whose behaviour is governed by several genetic loci. Or perhaps, you are interested in understanding the bio-assembly of chemical compounds that are built from subunits that can combine in multiple ways. In cases like these, it can be difficult to enumerate all possible types of individuals or chemical compounds that may occur, let alone all the possible interactions that can happen between them. Moreover, even if you would be able to describe all these interactions explicitly, the resulting model would be so large that it would be extremely difficult to analyse.

One way to deal with models of this level of complexity (which is not unusual in biology), is to use individual-based simulation (IBS) techniques. Rather than keeping track of the frequencies of all possible types in a population, these models consider a population of finite size, in which each individual is represented explicitly. IBS becomes more efficient than traditional approaches (such as ODE and population-based stochastic models) when the number of potential individual types exceeds the population size. An additional benefit is that developing an IBS can be more intuitive than working with more formal descriptions of dynamical systems. IBSs can often be formulated in terms of simple rules that capture how individuals interact in the same way as a biologist would describe these interactions. It is not uncommon for these simple rules of interaction to give rise to complex emergent phenomena at the population level, which would be difficult to capture directly in a population-based model.

A worked-out example: the stochastic corrector model

Object-oriented programming provides a natural way to implement IBSs. Without going into the details of this programming method, we here give a step-by-step description of the development of an individual-

based simulation program. We will only use basic methods for working with objects, to the extent that they were explained in *Module 11, Home-made objects*. Refer to the Modules in the advanced track *Object-Oriented Programming* for an in-depth coverage of this programming technique.

The biological question that we will focus on as an example take us back to the era of prebiotic evolution and the origin of life. We will study how groups of self-replicating molecules that rely on each other for successful replication can coexist, if those replicators also compete for a common pool of resources and some of them replicate more efficiently than others. Without coexistence of multiple replicators, it is difficult to explain how independently self-replicating molecules can become part of a larger and more complex evolutionary unit such as a protocell.

Theoretical background

Szathmáry and Demeter[1] proposed the following model to describe the interaction between two self-replicating molecules with concentrations x_0 and x_1:

$$\frac{d\,x_0}{dt} = b_0\,M(x_0,x_1) - d\,x_0 - \sigma\,x_0\,(x_0+x_1)$$
$$\frac{d\,x_1}{dt} = b_1\,M(x_0,x_1) - d\,x_1 - \sigma\,x_1\,(x_0+x_1)$$

Three processes are captured by these equations. First, both replicators contribute to the metabolic function of the cell, $M(x_0,x_1)$, which has an effect on both replication rates. However, the two replicators are allowed to have different replication rate constants, b_0 and b_1. Second, the replicators degrade spontaneously at degradation rate d. Both replicators are also equally affected by competition for a limiting resource, which is captured by the third term in the equations. The intensity of competition is determined by the competition parameter σ.

Simulations of the ODE system with $M(x_0,x_1) = \sqrt[4]{x_0\,x_1}$ (as proposed by Szathmáry and Demeter(1987)), paint a bleak picture of the potential for coexistence of the two replicators. In a well-mixed system, the concentrations of the two replicators increase until they start to feel competition. At that point, the most efficient replicators can still increase in frequency, whereas the other replicator becomes rarer. The result is that the metabolic efficiency of the system decreases, so that, eventually, both replicators are driven to extinction (Fig. 18.1).

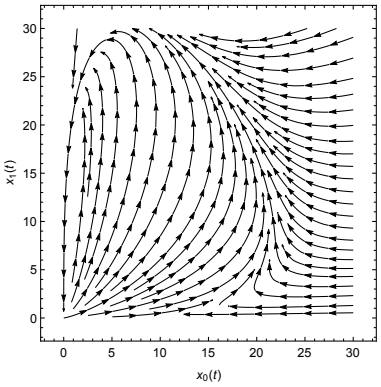


Fig. 18.1: Vectorfield representation of the ODE version of the stochastic corrector model, predicting extinction.

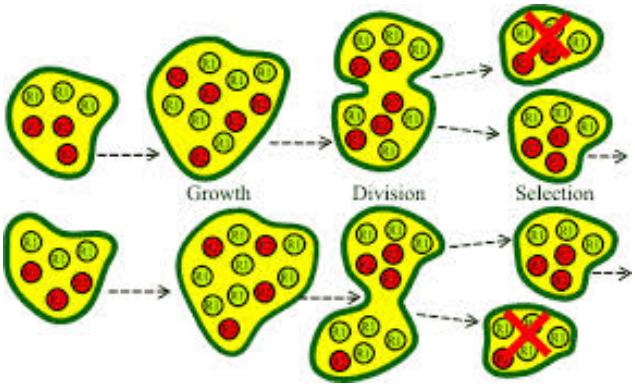


Fig. 18.2: Group selection in the stochastic corrector model: only compartments that contain the two replicators in more or less equal proportions can grow and replicate.

A potential mechanism to avoid competitive exclusion between competing replicators is group selection. Specifically, Szathmáry and Demeter argued that group selection may have been powerful in the early stages of the evolution of life, because self-replicating molecules may have been contained in small compartments, called protocells. Due to the small number of molecules within each compartment, protocell division would lead to daughter compartments with variable composition, creating variation between compartments on which group selection could subsequently act (Fig. 18.2). Group selection would eliminate compartments with an unbalanced composition, because such compartments cannot grow efficiently. Hence, by favouring compartments with roughly equal numbers of the two replicators, group selection is expected to prevent or delay the extinction of the two replicators.

[1] Szathmáry, E., and Demeter L. (1987). Group selection of early replicators and the origin of life. *J. Theor. Biol.* **128**: 463 Ð 486.

Individual-based implementation of the stochastic corrector model

The first step in the development of an IBS for Szathmáry and Demeter's 'stochastic corrector model' is the design of a class object that represents the 'individuals' in the simulation. In our case, the 'individuals' of the IBS are protocell compartments that may contain any number of replicators of each of the two types. One way to start programming these compartments would be like this:

```
1 const int kiMaxNrReplicators = 10;
2
3 class Compartment {
4 public:
5     Compartment() {x0 = x1 = kiMaxNrReplicators / 2;}
6     std::pair<int, int> getNrReplicators() const {return std::pair<int, int>(x0, x1);}
7 private:
8     int x0, x1;
9 };
```

So far, the only **private** members of the class are the integers `x0` and `x1`, which represent the copy numbers of the two replicators in the compartment. In addition, the class has two **public** member functions. The first one is a so-called constructor. This is a special function that has the same name as the class and that will be called automatically whenever an instance of the class is created. Accordingly, constructors are used to set the initial values of the member variables of a class. The constructor of the class `Compartment` initialises a new compartment with `kiMaxNrReplicators / 2` replicator molecules of each type (line 5), so that the compartment contains `kiMaxNrReplicators` initially. The value of the model parameter `kiMaxNrReplicators` is set as a global constant. The second **public** member of the class is the member function `getNrReplicators()`, which returns a `std::pair` containing the number of replicators of each type in the compartment. This function has **const** behind the argument list, to indicate that the function will not change any of the member variables of the class.

Both members functions have been defined inside the class definition, which is common practice for member functions that have very short function bodies. You can also define the member functions elsewhere, as `Compartment::Compartment()` and `Compartment::getNrReplicators()`, and specify only the function prototypes inside the class, as we have done so far. The two methods are equivalent, except for the fact that functions defined within a class are treated as if they were declared with the keyword **inline** (see p. **??**).

Already at this point, we can start testing the functionality of the class. For example, we can try to create an instance of `Compartment` and check if the number of replicators has been initialised correctly:

```
1 int main()
2 {
3     Compartment myCompartment;
4     std::pair<int, int> x = myCompartment.getNrReplicators();
5     std::cout << x.first << ' ' << x.second << '\n';
6     return 0;
7 }
```

We continue with the next stage only when we are sure that everything works as intended. From here on, we will alternate between phases in which new functionality is added to the class, and a testing phase to check if the newly added pieces fit in seamlessly with the rest of the class.

Modelling the dynamic of the replicators within a compartment

As a next step, we model how the number of replicators within a compartment changes as a result of replication, decay and competition. As the number of replicators within a compartment is small, it makes sense to simulate the dynamics of replicators within a compartment by means of the Gillespie algorithm. Therefore, we add

`double tNextEvent;` and

`std::vector<double> rates;`

as data members in the **private** part of class, as well as a **private** member function `reset()` that is used to calculate the values of these variables, depending on the current state. In addition, we write a **public** function `update()` to update the number of replicators according to the procedure outlined for the

Gillespie algorithm. Below are the definitions of these functions, together with a new definition of the constructor, which had to be updated in order to properly initialise the new member variables.

```
1  Compartment::Compartment()
2  // constructor, called at initialisation
3  {
4      x0 = x1 = kiMaxNrReplicators / 2;
5      rates = std::vector<double>(kiNEvent);
6      tNextEvent = 0.0;
7      reset();
8  }
9
10 void Compartment::reset()
11 // Recalculates the rates of the different
12 // events that can occur in a compartment
13 {
14     // birth rates of replicators
15     const double M = pow(x0 * x1, 0.25) / sqrt(kiMaxNrReplicators);
16     rates[0] = kdBirthRate0 * M * x0;      // replicator 0
17     rates[1] = kdBirthRate1 * M * x1;      // replicator 1
18
19     // death rate of replicators
20     const double aux = kdDeathRate + kdSigma * (x0 + x1);
21     rates[2] = aux * x0;                   // replicator 0
22     rates[3] = aux * x1;                   // replicator 1
23
24     // draw waiting time to the next event in this compartment
25     double sum = 0.0;
26     for(int i = 0; i < kiNEvent; ++i)
27         sum += rates[i];
28
29     if(sum > 0.0) {
30         std::exponential_distribution<double> waitingTime(sum);
31         tNextEvent += waitingTime(rng);
32     }
33     else
34         throw std::runtime_error("unable to draw waiting time.\n");
35 }
36
37 void Compartment::update()
38 // Implements a single birth or death event
39 // of a replicator in the compartment
40 {
41     // draw event and update state
42     std::discrete_distribution<int> event(rates.begin(), rates.end());
43     switch(event(rng))
44     {
45         case 0 :  ++x0; break;
46         case 1 :  ++x1; break;
47         case 2 :  --x0; break;
48         case 3 :  --x1; break;
49         default:  throw std::logic_error("Error: invalid event code in
       Compartment::update().\n");
50     }
51     //recalculate transitions rates
52     reset();
53 }
```

In what follows, we will use the parameter `kiMaxNrReplicators` to determine at what size protocell compartments are large enough to divide into two. This parameter appears on line 15 in the function `Compartment::reset()`, as a normalisation term of the metabolic efficiency function, which was included to make our model comparable to treatments of the stochastic corrector model in the scientific literature.

After adding definitions for the random number generator `rng` and the model parameters `kiNEvent`, `kdBirthRate0` (corresponding to b_0 in the ODE model), `kdBirthRate1` (b_1), `kdDeathRate` (d) and `kdSigma`

(σ), we can test what happens when the state of a compartment is repeatedly updated in the following way:

```
1  int main()
2  {
3      try {
4          // obtain seed from system clock and seed pseudo-random number generator
5          unsigned seed =
       static_cast<unsigned>(std::chrono::high_resolution_clock::now().time_since_epoch().count());
6          std::clog << "random seed : " << seed << '\n';
7          rng.seed(seed);
8
9          // create a Compartment and update it repeatedly
10         Compartment myCompartment;
11         for(;;) {
12             myCompartment.update();
13             std::pair<int, int> x = myCompartment.getNrReplicators();
14             std::cout << '(' << x.first << ',' << x.second << ")->";
15         }
16     }
17     catch(std::exception &error) {
18         std::cerr << error.what();
19         exit(EXIT_FAILURE);
20     }
21
22     return 0;
23 }
```

The output of the program is consistent with the prediction of the ODE model. Within a single compartment, the number of replicators first increases, but one replicator becomes more common than the other. This causes the metabolic efficiency to collapse, inducing the extinction of first one and then the other replicator.

```
random seed : 1007025759
(5,6)->(5,7)->(5,8)->(5,9)->(5,10)->(5,11)->(4,11)->(4,10)->(3,10)->(3,9)->(4,9)->(5,9)->
(5,8)->(5,9)->(5,10)->(5,11)->(4,11)->(5,11)->(5,12)->(5,13)->(5,14)->(5,15)->(5,14)->
(6,14)->(6,13)->(6,14)->(7,14)->(7,15)->(7,16)->(7,17)->(7,18)->(7,19)->(8,19)->(8,18)->
(8,17)->(8,18)->(9,18)->(8,18)->(9,18)->(9,19)->(9,20)->(9,19)->(8,19)->(7,19)->(6,19)->
(5,19)->(5,20)->(5,21)->(5,22)->(5,23)->(4,23)->(4,22)->(4,23)->(3,23)->(3,22)->(3,23)->
(3,24)->(3,25)->(3,26)->(3,27)->(2,27)->(2,26)->(2,27)->(2,28)->(2,27)->(2,28)->(2,27)->
(2,28)->(2,27)->(2,26)->(2,25)->(2,26)->(2,25)->(2,26)->(2,25)->(2,26)->(2,27)->(2,28)->
(2,29)->(2,28)->(3,28)->(3,29)->(3,28)->(3,29)->(3,28)->(3,27)->(3,26)->(3,27)->(3,26)->
(3,27)->(3,28)->(3,29)->(3,28)->(3,27)->(3,28)->(3,27)->(2,27)->(2,26)->(2,25)->(2,26)->
(2,25)->(2,26)->(2,27)->(2,26)->(2,27)->(2,28)->(2,29)->(2,30)->(2,29)->(2,28)->(2,27)->
(2,26)->(2,27)->(2,28)->(2,27)->(1,27)->(1,26)->(1,27)->(1,26)->(1,27)->(1,26)->(1,25)->
(1,26)->(1,27)->(1,28)->(0,28)->(0,27)->(0,26)->(0,25)->(0,24)->(0,23)->(0,22)->(0,21)->
(0,20)->(0,19)->(0,18)->(0,17)->(0,16)->(0,15)->(0,14)->(0,13)->(0,12)->(0,11)->(0,10)->
(0,9)->(0,8)->(0,7)->(0,6)->(0,5)->(0,4)->(0,3)->(0,2)->(0,1)->unable to draw waiting time.
```

As you can see, the simulation terminated after an exception was thrown from line 34. This was triggered by the fact that there were no replicators left in the compartment, so that the sum of all rates became zero. More detailed information can be gathered in this second phase of testing the program, but we will not show this here and move on to the next stage of developing the simulation.

Fission of protocells

When the number of replicators in a compartment grows to `kiMaxNrReplicators`, the protocell divides into two, and its replicators are randomly distributed over the two daughter cells. To implement the division of compartments, we define a function `divide()`, which distributes the replicators in a protocell between the current `Compartment` (this will become one of the daughter cells) and another object that will become the other daughter cell. In the implementation shown below, the function `Compartment::divide()` calls another function that determines how the replicators are distributed among the daughter cells by sampling from a binomial distribution.

```
1  int divideRandom(const int x)
2  // returns the number of replicators that end up in one daughtercell,
3  // assuming that each element in a set of x replicators ends up there
4  // with probability 0.5
5  {
6      if(x == 0)
7          return 0;
8      else {
9          std::binomial_distribution<int> sample(x, 0.5);
10         return sample(rng);
11     }
12 }
13
14 void Compartment::divide(Compartment &obj)
15 // Implements division of the compartment. The mother compartment
16 // will become one of the daughter cells. Compartment obj will
17 // be replaced by the other daughter cell.
18 {
19     obj.x0 = obj.x1 = 0;
20     for(int tot0 = x0, tot1 = x1; x0 + x1 == 0 || obj.x0 + obj.x1 == 0; ) {
21         x0 = divideRandom(tot0);
22         x1 = divideRandom(tot1);
23         obj.x0 = tot0 - x0;
24         obj.x1 = tot1 - x1;
25     }
26     obj.tNextEvent = tNextEvent;
27     reset();
28     obj.reset();
29 }
```

The function of the **for**-loop in `Compartment::divide()` (line 20-25) is to prevent outcomes where one daughter cell is an exact copy of the mother cell, which would occur if the other daughter cell inherits no replicators. After the division has occurred, the daughter cells are synchronised and both reset their rates in accordance with their new state.

To check whether compartment division is implemented properly we write a simple test program. After declaring an object `myCompartment` and showing its state on the screen, the program calls the function `divide()` which distributes the replicators between `myCompartment` and another `Compartment` object named `otherCompartment`.

```
1  int main()
2  {
3      try {
4          // obtain seed from system clock and seed pseudo-random number generator
5          unsigned seed =
       static_cast<unsigned>(std::chrono::high_resolution_clock::now().time_since_epoch().count());
6          std::clog << "random seed : " << seed << '\n';
7          rng.seed(seed);
8
9          // create a protocell
10         Compartment myCompartment;
11         std::pair<int, int> x = myCompartment.getNrReplicators();
12         std::cout << "before division : (" << x.first << ',' << x.second << ")\n";
13
14         // divide the protocell into two daughter cells
15         Compartment otherCompartment;
16         myCompartment.divide(otherCompartment);
17         x = myCompartment.getNrReplicators();
18         std::pair<int, int> y = otherCompartment.getNrReplicators();
19         std::cout   << "after division  : (" << x.first << ',' << x.second << ") + "
20                     << '(' << y.first << ',' << y.second << ")\n";
21
22         // update both daughter cells
23         myCompartment.update();
24         otherCompartment.update();
```

```
        x = myCompartment.getNrReplicators();
        y = otherCompartment.getNrReplicators();
        std::cout   << "after update    : (" << x.first << ',' << x.second << ") + "
                    << '(' << y.first << ',' << y.second << ")\n";
    }
    catch(std::exception &error) {
        std::cerr << error.what();
        exit(EXIT_FAILURE);
    }

    return 0;
}
```

The screen output of the program,

```
random seed : 18326758
before division : (5,5)
after division  : (4,4) + (1,1)
after update    : (3,4) + (2,1)
```

enables us to verify that the total number of replicators is the same before and after the division. Moreover, it appears that the new daughter cells can also be updated without problems.

Implementing the simulation

Now that the protocells can update their state and divide into two daughter compartments, we are ready to start working on the simulation program itself, which will simulate a whole population of compartments. At this stage, one often realises that small modifications to the implementation of the class are necessary, or that additional member functions are needed. For example, while writing the code below, we simplified the member function `Compartment::getNrReplicators()` to

```
int getNrReplicators() const {return x0 + x1;}
```

since access to the total number of replicators was all that was needed in the simulation program. We also added a member function `Compartment::getTimeOfNextEvent()`, defined as

```
double getTimeOfNextEvent() const {return tNextEvent;}
```

in order to gain read-access to the private data member `tNextEvent`. Listing 18.1 outlines the simulation program:

Code listing 18.1: Simulation of the stochastic corrector model

```
/*************************************************************
** Individual -based implementation of Szathmary and Demeter's **
** stochastic corrector model.                                 **
** written on 26-09-2014 by Master Yoda                        **
*************************************************************/

#include <iostream>
#include <fstream>
#include <utility>
#include <vector>
#include <cmath>
#include <exception>
#include <cstdlib>
#include <random>

/*************************************************************
***                  model parameters                       ***
*************************************************************/

const int kiPopulationSize      = 1000; // number of compartments
const double tEnd               = 100.0;// simulation time
const int kiMaxNrReplicators    = 10;   // max number of replicators per cell
const int kiNEvent              = 4;    // number of event types in Gillespie algorithm
const double kdBirthRate0       = 50.0; // per capita birth rate of first replicator
const double kdBirthRate1       = 60.0; // per capita birth rate of second replicator
const double kdDeathRate        = 0.5;  // per capita death rate of first replicator
```

```
const double kdSigma             = 1.75; // resource competition coefficient

/*************************************************************
***                    global  variables                   ***
*************************************************************/

std::mt19937_64 rng;
std::vector<Compartment> compartments(kiPopulationSize);

/*************************************************************
***                    simulation  code                    ***
*************************************************************/

void initialise()
// obtains seed from system clock and seed pseudo-random number generator
{
    unsigned seed =
    static_cast<unsigned>(std::chrono::high_resolution_clock::now().time_since_epoch().count());
    std::clog << "random seed : " << seed << '\n';
    rng.seed(seed);
}

void simulate()
{
    for(double t = 0.0, tSav = 0.0; t < tEnd;) {
        //select the compartment that will change state first
        int i = -1;
        double tnext = 0.0;
        for(int j = 0; j < kiPopulationSize; ++j) {
            if(compartments[j].getNrReplicators() == 0) continue;
            double tj = compartments[j].getTimeOfNextEvent();
            if(i < 0 || tj < tnext) {
                i = j;
                tnext = tj;
            }
        }
        // stop if all compartments are empty
        if(i < 0)
            return;

        // update the selected compartment
        compartments[i].update();
        t = tnext;

        // compartment division
        if(compartments[i].getNrReplicators() > kiMaxNrReplicators) {
            // try to find an empty compartment
            int j;
            for(j = 0 ; j < kiPopulationSize; ++j)
                if(compartments[j].getNrReplicators() == 0) break;
            // select a random compartment (other than i) if there is no empty one
            while(j == kiPopulationSize || j == i) {
                std::uniform_int_distribution<int> sample(0, kiPopulationSize - 1);
                j = sample(rng);
            }
            // implement compartment division
            compartments[i].divide(compartments[j]);
        }
        if(t > tSav) {
            std::cout << t << '\n';
            tSav += 1.0;
        }
    }
}

void exportDistribution()
```

```
92 {
93     // analyse distribution of compartment sizes
94     std::vector<int> histogram(kiMaxNrReplicators + 1, 0);
95     for(int i = 0; i < kiPopulationSize; ++i)
96         ++histogram[compartments[i].getNrReplicators()];
97 
98     // store data in file
99     std::ofstream ofs("data.csv");
100    if(!ofs.is_open())
101        throw std::runtime_error("unable to open file.\n");
102 
103    for(int k = 0; k <= kiMaxNrReplicators; ++k)
104        ofs << k << ',' << histogram[k] * 1.0 / kiPopulationSize << '\n';
105 }
106 
107 /*************************************************************
108 ***                    main()                                 ***
109 *************************************************************/
110 
111 int main()
112 {
113     try {
114         initialise();
115         simulate();
116         exportDistribution();
117     }
118     catch(std::exception &error) {
119         std::cerr << error.what();
120         exit(EXIT_FAILURE);
121     }
122 
123     return 0;
124 }
```

The final version of the definition and implementation of the **class** `Compartment` is shown in Listing 18.2. Apart from the adjustments to the class definition that were mentioned already, the function `Compartment::update()` was modified to prevent the simulation from terminating prematurely when the replicators in one of the compartments go extinct. In the new version, the reaction rates are recalculated only if a compartment is not empty (line 77-78 of Listing 18.2). This prevents that `reset()` will throw the exception on line 56 when a compartment lost all of its replicators. We should now make sure that the program never calls the function `Compartment::update()` for cells that have become empty. This was ensured by adding the **throw** statement on line 63-64.

To create a functional program, combine the code in Listing 18.1 with that in 18.2. Insert the class definition between the section `*** model parameters ***` and the section `*** global variables ***`, and move the class implementation below `*** global variables ***`.

Code listing 18.2: Definition and implementation of the class `Compartment`

```
1  /*************************************************************
2  ***          definition  of  the  class  Compartment          ***
3  *************************************************************/
4 
5  class Compartment {
6  public:
7      Compartment();
8      void update();
9      void divide(Compartment&);
10     int getNrReplicators() const {return x0 + x1;}
11     double getTimeOfNextEvent() const {return tNextEvent;}
12 private:
13     int x0, x1;
14     double tNextEvent;
15     std::vector<double> rates;
```

```
16      void reset();
17  };
18
19  /*************************************************************
20  ***          implementation of the class Compartment              ***
21  *************************************************************/
22
23  Compartment::Compartment()
24  // constructor, called at initialisation
25  {
26      x0 = x1 = kiMaxNrReplicators / 2;
27      rates = std::vector<double>(kiNEvent);
28      tNextEvent = 0.0;
29      reset();
30  }
31
32  void Compartment::reset()
33  // Recalculates the rates of the different
34  // events that can occur in a compartment
35  {
36      // birth rates of replicators
37      const double M = pow(x0 * x1, 0.25) / sqrt(kiMaxNrReplicators);
38      rates[0] = kdBirthRate0 * M * x0;      // replicator 0
39      rates[1] = kdBirthRate1 * M * x1;      // replicator 1
40
41      // death rate of replicators
42      const double aux = kdDeathRate + kdSigma * (x0 + x1);
43      rates[2] = aux * x0;                // replicator 0
44      rates[3] = aux * x1;                // replicator 1
45
46      // draw waiting time to the next event in this compartment
47      double sum = 0.0;
48      for(int i = 0; i < kiNEvent; ++i)
49          sum += rates[i];
50
51      if(sum > 0.0) {
52          std::exponential_distribution<double> waitingTime(sum);
53          tNextEvent += waitingTime(rng);
54      }
55      else
56          throw std::runtime_error("unable to draw waiting time.\n");
57  }
58
59  void Compartment::update()
60  // Implements a single birth or death event
61  // of a replicator in the compartment
62  {
63      if(x0 + x1 == 0)
64          throw std::logic_error("Error: Compartment::update() cannot be called for empty
        compartment.\n");
65
66      // draw event and update state
67      std::discrete_distribution<int> event(rates.begin(), rates.end());
68      switch(event(rng))
69      {
70          case 0 :  ++x0; break;
71          case 1 :  ++x1; break;
72          case 2 :  --x0; break;
73          case 3 :  --x1; break;
74          default:  throw std::logic_error("Error: invalid event code in
        Compartment::update().\n");
75      }
76      //recalculate transitions rates
77      if(x0 + x1 > 0)
78          reset();
79  }
```

```
80
81 int divideRandom(const int x)
82 // returns the number of replicators that end up in one daughtercell,
83 // assuming that each element in a set of x replicators ends up there
84 // with probability 0.5
85 {
86     if(x == 0)
87         return 0;
88     else {
89         std::binomial_distribution<int> sample(x, 0.5);
90         return sample(rng);
91     }
92 }
93
94 void Compartment::divide(Compartment &obj)
95 // Implements division of the compartment. The mother compartment
96 // will become one of the daughter cells. Compartment obj will
97 // be replaced by the other daughter cell.
98 {
99     obj.x0 = obj.x1 = 0;
100    for(int tot0 = x0, tot1 = x1; x0 + x1 == 0 || obj.x0 + obj.x1 == 0; ) {
101        x0 = divideRandom(tot0);
102        x1 = divideRandom(tot1);
103        obj.x0 = tot0 - x0;
104        obj.x1 = tot1 - x1;
105    }
106    obj.tNextEvent = tNextEvent;
107    reset();
108    obj.reset();
109 }
```

The simulation program was run with the same parameter set as the one used for drawing the ODE vectorfield in Fig. 18.1 (`kdBirthRate0` $= b_0 = 50$, `kdBirthRate1` $= b_1 = 60$, `kdDeathRate` $= d = 0.5$ and `kdSigma` $= \sigma = 1.75$), using different values of the threshold compartment size for protocell division `kiMaxNrReplicators`. At the end of each simulation, we recorded the final distribution of compartment sizes. Fig. 18.3 shows the results. As predicted by the ODE model, the two replicators go extinct if the variation between daughter cells is small (i.e., when `kiMaxNrReplicators` is sufficiently large). However, for low values of the division threshold `kiMaxNrReplicators`, stochastic fluctuations in the number of replicators that are passed on to the daughter compartments produce so much variation that group selection can maintain a population of viable compartments. This is evidenced by the presence of a stable distribution of compartment sizes > 0. Fig. 18.3 indicates that the maximum compartment size that allows for the maintenance of the two replicators (given the other parameter values) lies just above 20 replicator molecules.

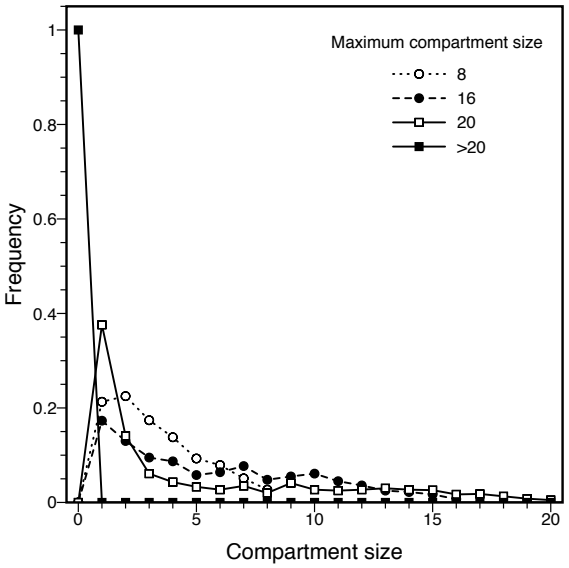


Fig. 18.3: Frequency distribution of the number of replicators per compartment (compartment size) in the stochastic corrector model for different values of the threshold for cell division.

18.2 Exercises

Test your skill

18.1. Self-organised division of labour ($\star$). Insects that live in large eusocial colonies often show a high level of task specialisation but may switch between performing different tasks (e.g., brood care, foraging or nest defence) a few times during their life. It is not entirely clear what cues govern this division of labour, nor how an individual decides which task to perform in a huge colony that appears to lack a detailed mechanism of central control.

One hypothesis, proposed by Theraulaz *et al.*[2] is that division of labour in insect colonies is an emergent phenomenon that arises due to the reinforcement of individual response thresholds for performing certain tasks. The key idea of their model is that individuals posses thresholds for responding to different task stimuli. If an individual performs a task often, the threshold for performing this task becomes lower, so that the individual can be induced more easily to perform the same task once again. On the other hand, if an individual has not performed a task for a while, its threshold for that task will have increased, making it even less likely that the individual will perform that particular task in the future. These processes are meant to reflect the ability of individuals to boost their performance by learning, and their tendency to lose behaviours that are not reinforced.

Your task in this Exercise is to implement a variant of the response threshold reinforcement model. Proceed in the following steps (make sure to test your program after you have completed each step):

1. Create a `class Individual` that contains a `std::vector<double>` of task preferences for a number of different tasks.

2. Add a member function `void Individual::performTask(int j)` that accepts an integer argument j representing the task that is performed by the individual at the current moment. This function should update the task preferences in the following way:

$$p_i \longleftarrow (1 - \beta_i)\, p_i + \delta_{ij}\, \alpha_i$$

Here, p_i is the individual's behavioural preference for task i (which is inversely related to its response threshold), β_i is the task-specific rate of forgetting and α_i is the learning rate for task i. Learning occurs only for the task that is currently executed, so δ_{ij} is equal to one if $i = j$ and zero otherwise ($i \neq j$). The parameter vectors α_i and β_i can be defined in your program as global variables, but make sure that they are properly initialised somewhere (see item 5 for a suggested list of parameter values).

[2] Theraulaz, G., Bonabeau, E., and Deneubourg, J.L. (1989). Response threshold reinforcement and division of labour in insect societies. *Proc. R. Soc. Lond. B* **265**: 327-332.

If all tasks are performed by an individual equally often, p_i is expected to converge to the equilibrium value

$$p_i^* = \frac{\alpha_i}{n\,\beta_i}$$

where n is the number of different tasks. Use these equilibrium values to initialise the task preference values for an `Individual` at the start of its life.

3. Add a member function `void Individual::pickTask(std::vector<int> &jobs)` to implement how an individual selects the task it will perform next, given its preferences and the availability of `jobs` for each task. Assume, that at a given point in time, k_i individuals are needed to perform task i. (In your program, the numbers k_i will be contained in the `std::vector<int> jobs` that is passed by reference to the function `pickTask`.) Assume, furthermore, that the probability that the individual selects task i is proportional to the number of individuals that are needed to perform the task (i.e., the number of available jobs, k_i) and the preference of the individual for performing the task, p_i, i.e.,

$$\Pr(\text{task } i) \propto p_i \cdot k_i$$

When the function `void Individual::pickTask(std::vector<int> &jobs)` is called for an `Individual`, the following steps should be executed:

   (a) The `Individual` selects a task based on a weighted lottery with probability weights $p_i \cdot k_i$. (In the unlikely event that all probability weights are zero, assume that the `Individual` selects one of the available jobs at random.)

   (b) The `Individual` performs the selected task, by means of a call to the member function `performTask`.

   (c) The number of available jobs for the selected task is decreased by one.

4. Create a colony of `Individuals` of size $N = 100$. Write a function to simulate the actions of the individuals during a single time step in which all individuals are allowed to pick an available task. Start each time step by initialising the vector `jobs` with equal numbers of jobs for each task, and a total number of N jobs (so that there is exactly one job to do for each individual). Then, allow each individual in the population to pick one of the available jobs, one individual after the other in a random sequence, until all individuals have picked a task, and no more jobs are left.

5. Simulate the behaviour of individuals for 100 steps. Assume that there are two tasks with $\alpha_1 = 0.8$, $\alpha_2 = 0.5$, $\beta_1 = 0.3$ and $\beta_2 = 0.5$ (i.e., the second task is less rewarding and is forgotten more easily than the first). Extend the class `Individual` to be able to keep track of the sequence of tasks that was performed by an individual during the 100 steps of the simulation. Write this information to a data file for a subset of the individuals. Also use the task sequence data to determine how often individuals switched between tasks, and write the population average of this switching index to the data file. Do the individuals in your simulation show division of labour?

Part III:

Appendices

Projects

Bachelor

BSc students use the last week of the course to work on a mini-project. The list of available topics for this year is provided below. We strongly recommend that you work on the project in a team of 2 or 3 students, but you are allowed to work individually as well.

Each project requires that you learn some background theory, which you can find in Module 13, 16, 17, 18 of the manual. The lectures for Module 16-18 are available as a prerecorded lecture; Module 13 is a review chapter for which there is no separate lecture.

During the mini-project, you will develop a C++ program, and use it to run simulations or collect some results. These results have to be documented in a brief written report, accompanied by some explanation or interpretation and summarising conclusions of the biological insights that are generated by your analysis. The report has to be submitted with a clean and annotated version of the C++code of your self-written simulation program, which will be evaluated based on the same criteria as the weekly programming assignments. The deadline for submitting the results of the mini-project is on Friday 01/12, 17:00 o' clock.

Do not start coding without a clear plan; think of a way to build your program in small, incremental steps. Make sure to test your program after each step and save your code often, so that you can always fall back on an earlier working version of your program when you run into unanticipated problems. Do not wait too long to ask for help in the event that you get stuck. The following suggested time-planning can help you keep track of your progress during the mini-project week:

Monday: study necessary background theory and reading material

Tuesday - Wednesday: develop initial implementation, optimize and streamline program code

Thursday: run simulations and visualize results

Friday: write brief report, clean-up code and finish code annotation

You can find a list of mini-project options below. The report should detail your used approach, some background information on the biological information of your project, details on hurdles and programming issues that you had to overcome, and finally show results based on your programming code. Examples of good quality BSc mini-project reports are available on Brightspace.

Associated chapters:

project 3, 4, 7, 8 = Module 16 (ODE models)
project 12 = Module 17 (stochastic models)
project 2, 6, 9, 10, 13 = Module 18 (individual-based simulation)
project 1, 5 = Module 13 (review chapter; without lecture) or Module 14 (STL; without lecture)

Also the Test-Your-Skill exercises of Module 17 and 18 are suitable as mini-project options.

List of projects

1. **Bio-informatics:** write a program to align two DNA sequences by means of the Needleman-Wunsch algorithm.
   Technique: dynamic programming.
   Reference: http://en.wikipedia.org/wiki/Needleman-Wunsch algorithm

2. **Microbiology:** compare how efficiently bacteria with different movement patterns can migrate towards a source of nutrients.
   Technique: individual-based simulation.
   Reference: Taktikos, J., Stark, H. and Zaburdaev, V. (2013). How the motility pattern of bacteria affects their dispersal and chemotaxis. *PLoS One* **8**: e81936.

3. **Physiology:** investigate how reduced coupling between cell populations in the heart can lead to emergence of cardiac arrhythmia.
   Technique: simulation of discontinuous ODE systems.
   Reference: Sheheitli, H., and Rand, R. (2009). Origin of arrhythmias in a heart model. *Commun. Nonlinear Sci. Numer. Simulat.* **14**: 3707-3714.

4. **Development:** study the dynamic of pattern formation during organismal growth and development.
   Technique: numerical integration of partial differential equations.
   Reference: Kondo, S., and Asai, R. (1995). A reaction-diffusion wave on the skin of the marine angelfish *Pomacanthus*. *Nature* **376**: 765-768.

5. **Neural networks:** implement a Hopfield network to study how a collection of simple neurons can be trained to remember and reconstruct complex information.
   Technique:: neural network programming.
   Reference: http://en.wikipedia.org/wiki/Hopfield network

6. **Behavioural biology:** model the emergence of dominance hierarchies in social groups.
   Technique: individual-based simulation.
   Reference: Hemelrijk, C.K. (1999). An individual-oriented model of the emergence of despotic and egalitarian societies. *Proc. R. Soc. Lond. B* **266**: 361-369.

7. **Evolutionary genetics:** simulate evolutionary transitions between genetic sex-determining systems triggered by sexually antagonistic selection.
   Technique: iteration of a large system of coupled difference equations.
   Reference: Van Doorn, G.S., and Kirkpatrick, M. (2007). Turnover of sex chromosomes induced by sexual conflict. *Nature* **449**: 909-912.

8. **Population dynamics:** model the chaotic dynamics of populations competing for ecological resources using state-of-the-art numerical integration methods.
   Technique: Runge-Kutta integration of ordinary differential equations with adaptive step size control.
   Reference: Huisman, J., and Weissing, F.J. (1999). Biodiversity of plankton by species oscillations and chaos. *Nature* **402**: 407-410.

9. **Cultural evolution:** compare the evolutionary dynamics of cooperation in a population with and without conformism.
   Technique: individual-based simulation based on evolutionary game theory.
   Reference: Pena, J. (2008). Conformist transmission and the evolution of cooperation. *Proc. Artificial Life* **11**: 458-465.

10. **Bio-informatics:** write a program to construct a phylogenetic tree in Newick format from a genetic distance matrix.
    Technique: UPGMA (Unweighted Pair Group Method with Arithmetic Mean) clustering.
    References: (1) Fitch, W.M., and Margoliash, E. (1967). Construction of phylogenetic trees. *Science* **155**: 279-284. (2) Romesburg, H.C. (1984). Cluster Analysis for Researchers. Lifetime Learning Publications, Belmont CA, pages 14-23.

11. **Evolutionary ecology:** model the emergence of species by sympatric speciation.
    Technique: Individual-based simulation.
    Reference: Dieckmann, U. and Doebeli, M. (1999). On the origin of species by sympatric speciation. *Nature* **400**: 354-357.

12. **Phylogenetics:** simulate a phylogenetic tree using a diversity dependent diversification model.
    Technique stochastic simulation based on the Gillespie algorithm.
    Reference: Etienne Rampal S., Haegeman Bart, Stadler Tanja, Aze Tracy, Pearson Paul N., Purvis Andy and Phillimore Albert B. (2012) Diversity-dependence brings molecular phylogenies closer to agreement with the fossil record *Proc. R. Soc. B.* 2791300–1309

13. **Population genetics:** write a program to simulate the admixture of genomes after hybridization *Technique:* individual-based simulation.
Reference: Thijs Janzen, Arne W. Nolte, Arne Traulsen (2018) The breakdown of genomic ancestry blocks in hybrid lineages given a finite number of recombination sites. *Evolution* Volume 72, Issue 4, 1 April 2018, Pages 735–750.

Useful pieces of code

4.1 The if-else-statement 38
4.3 The conditional operator 42
4.4 The switch statement 43
5.1 The for-statement 50
5.2 Iterating over elements of a std::vector 52
5.3 Creating a matrix 53
5.4 Some member functions of a std::vector 54
5.5 The while-statement 58
5.6 The do-while-statement 59
6.1 Function prototypes 66
8.1 Calling functions by reference and const-reference 80
8.2 Minimal working example of a reference 84
9.1 String manipulation functions 92
10.1 File I/O 98
10.2 Applying stringstreams to generating filenames during a simulation 101
11.4 Data encapsulation 112
13.2 Redirecting a standard output stream 169
14.4 `std::array` 190
14.5 `std::bitset` 190
14.7 `std::list` 195
15.2 Verifying assumptions using the `assert` statement 209
15.3 Numerical analysis of an age-structured population model 212
16.1 Euler integration 225
16.2 Runge-Kutta integration stepper 227
16.4 Adaptive step-size control (Bogacki-Shampine) 240
17.2 Sampling from different standard statistical distributions 248
17.3 Implementation of a weighted lottery 250
17.4 The Gillespie algorithm 252

List of exercises

Exercise 1.1. tHello world *13*
Exercise 1.2. tBasic calculations in C++ *13*
Exercise 1.3. tInteger division *13*
Exercise 1.4. tPrime number machine *13*
Exercise 2.1. tInteger versus floating-point arithmetic *23*
Exercise 2.2. tFloating point accuracy *23*
Exercise 2.3. tImplicit type conversion I *23*
Exercise 2.4. tImplicit type conversion II *23*
Exercise 2.5. tThe Quetelet index *23*
Exercise 2.6. tMasterMix *24*
Exercise 3.1. tC++ operators *35*
Exercise 3.2. tToo many brackets *35*
Exercise 3.3. tCorrect scoping *35*
Exercise 3.4. tGompertz law for tumour growth *35*
Exercise 3.5. tThe abc formula *35*
Exercise 4.1. tWhat kind of critter is this? *41*
Exercise 4.2. tExclusive OR *41*
Exercise 4.3. tThe optimist/pessimist quiz *41*
Exercise 4.4. tTemperature converter and weather advisory *41*
Exercise 4.5. tThe conditional operator *45*
Exercise 4.6. tThe switch-statement *45*
Exercise 5.1. tFind the bugs *55*
Exercise 5.2. tThe Ricker model of population growth *55*
Exercise 5.3. tCreating a histogram *55*
Exercise 5.4. tEstimating rates of mutation *55*
Exercise 5.5. tNasty nested for-statements *56*
Exercise 5.6. tDescriptive statistics *56*
Exercise 5.7. tAll loop statements lead to Rome *59*
Exercise 6.1. tScope of a function *67*
Exercise 6.2. tDefining a function *67*
Exercise 6.3. tFinding the largest element in a vector *67*
Exercise 6.4. tCounting the number of matches *67*
Exercise 6.5. tCreate a calculator *67*
Exercise 8.1. tReferences *83*
Exercise 8.2. tOverloading and calling by reference *84*
Exercise 8.3. tCalling by reference *84*
Exercise 8.4. tComputing the factorial of a number *84*
Exercise 8.5. tPredator Prey dynamics *84*
Exercise 8.6. tAdvanced version *85*
Exercise 9.1. tLooping through the alphabet *92*
Exercise 9.2. tConverting between upper- and lowercase letters *93*
Exercise 9.3. tDNA transcription *93*
Exercise 9.4. tRNA translation *93*
Exercise 10.1. tPractising file I/O - part 1 *101*
Exercise 10.2. tPractising file I/O - part 2 *101*
Exercise 10.3. tPractising file I/O - part 3 *102*
Exercise 11.1. tElementary operations on a std::pair and a user-defined structure *114*
Exercise 11.2. tElementary operations on a class object *115*
Exercise 11.3. tThe PGCPPPHD phone directory *115*
Exercise 11.4. tComplex number arithmetic *115*
Exercise 11.5. tSimulating a farm *118*
Exercise 12.1. tThe sequential assessment game *145*
Exercise 12.2. tThe sequential assessment game with imperfect information *147*
Exercise 12.3. tComplex number arithmetic - continued *148*
Exercise 12.4. tImplementing mathematical set objects *148*
Exercise 13.1. tThe hangman game *172*
Exercise 13.2. tThe hangman game – continued *173*
Exercise 13.3. tThe neighbour-joining method for phylogenetic tree reconstruction *173*
Exercise 15.1. tThe bisection algorithm for locating the root of a function *215*

Exercise 15.2. tPractise your bug-hunting skills *216*
Exercise 15.3. tThe efficiency/accuracy trade-off *217*
Exercise 15.4. tModelling sexual selection *218*
Exercise 16.1. tOscillations in a predator-prey model *227*
Exercise 16.2. tLower- versus higher-order integration schemes *228*
Exercise 16.3. tBistability in gene-regulation *228*
Exercise 16.4. tThe Hodgkin-Huxley model for the action potential *229*
Exercise 16.5. tMaking sense of adaptive step-size control *233*
Exercise 16.6. tRunge-Kutta integration with adaptive step-size control *234*
Exercise 17.1. tPicking a random element from a vector *254*
Exercise 17.2. tThe shuffle algorithm *254*
Exercise 17.3. tStochastic population growth *254*
Exercise 17.4. tDiffusion-limited aggregation *255*
Exercise 18.1. tSelf-organised division of labour *270*

Index

inline, 130
signed, 90
std::, 11
using, 12

    =, 39
' ', 90
++, 27
-, 27
/*...*/, 10
//, 10
;, 10
«, 11
==, 39
», 11
?:, 42
[], 51
#include, 10
    cctype, 93
    chrono, 246
    cmath, 35
    cstdlib, 99, 245
    fstream, 97
    iomanip, 102
    iostream, 10, 97
    random, 247
    sstream, 100
    string, 90
    utility, 107
    vector, 51
&, 79
bool, 39
break, 43, 51
case, 43
chrono, 246
class, 111
    private, 111
    public, 111
cmath, 35
const
    reference, 79
continue, 57
cstdlib, 99, 245
default, 43
double, 15
do, 58
else, 38
false, 39
float, 21
for, 49
fstream, 97
if, 38
inline, 261
iomanip, 102
iostream, 10, 97
ostringstream, 100
private, 111
public, 111
rand(), 245
random, 247
srand(), 246
static_cast, 19
std::string::npos, 92
std::
    cerr, 97
    cin, 11
    clog, 97
    cout, 11
    endl, 11
    ifstream, 98
    ofstream, 97
    ostringstream, 100
    string, 90
    vector, 51
string, 90
struct, 109
switch, 43
true, 39
unsigned, 89
utility, 107
vector, 51
while, 58

algorithm, 205
arithmetic operators, 13, 27
assignment
    compound, 27
assignment operator, 17

bool, 39
brackets, 29
break, 43, 51, 57
bug, 208

case, 43
checking I/O status, 100
cin, 10
class, 111
    implementation, 111
    initialisation, 129
    interface, 111
comma operator, 29
comment, 11
compound assignment, 27
compound data type, 109
compound statement, 30
conditional operator, 42
conditional statement, 37
const, 10
constructor, 128
cout, 10

debugging, 208
declaration, 15
declaration statement, 15, 17
decrement operator, 27
default, 43
division operator, 18
do, 59
double, 16

else, 38
encapsulation, 111
end of file, 99
endl, 10
error
    exit, 99
    ios::eof(), 99
    ios::fail(), 100
    compile-time, 208
    division, 18
    overflow, 20, 90
    run-time, 208
    underflow, 21
explicit type conversion, 19
expression, 28
    constant, 17
    logical, 39
expression statement, 29

file stream, 97
float, 21, 22
floating-point arithmetic, 21
floating-point division, 18
floating-point underflow, 21
flowchart, 206

for, 50
function, 63
  argument, 64
  call by reference, 79
  call by value, 77
  declaration, 65
  definition, 64, 65
  prototype, 65
  recursive, 83
functions continued, 77

hexadecimal notation, 17

IDE, 8
if, 38
implementation, 111
implicit type conversion, 18
increment operator, 27
  prefix vs. postfix, 28
individual-based simulation, 259
initialisation statement, 17
inline, 130
int, 9
integer division, 13, 18
interface, 111
is equal to operator, 39
is not equal to operator, 39
iterative statement, 49

jump statement, 57

library
  cctype, 93
  chrono, 246
  cmath, 35
  cstdlib, 99, 245
  fstrream, 97
  iomanip, 102
  iostream, 97
  random, 247
  string, 90
  utility, 107
  vector, 51
literal, 17
  char, 90
  string, 90
logical expression, 39
logical operators, 39
lvalue, 17

main, 9
matrix, 53
modulo operator, 13, 18

namespace, 11

object
  class, 111
  struct, 109
  data member, 110
  instance, 110
  member function, 111
  member variable, 110
octal notation, 17
ODE, 223
operator, 27
  [], 51
  arithmetic, 13
  assignment, 17
  comma, 29, 50
  compound assignment, 27
  conditional, 42
  decrement, 27
  division, 18
  increment, 27
  is equal to, 39
  is not equal to, 39
  logical, 39
  modulo, 13, 18
  precedence, 29
  relational, 39
ordinary differential equation, 223
overflow, 20, 90

precedence rules, 29
precision, 20
  floating point, 21
  underflow, 21
pseudo-random number, 245
pseudocode, 206

random number, 245
random seed, 246
recursive function, 83
reference, 79
  const, 79
reference parameter, 79
relational operators, 39
return, 9

scientific notation, 18
scope, 30
seed random number generator, 246
semicolon, 10
statement
  break, 51
  continue, 57
  do...while, 58
  for, 49
  if...else, 37
  switch, 43
  compound, 30
  conditional, 37
  declaration, 15, 17
  expression, 29
  initialisation, 17
  iterative, 49
  jump, 57
statistical distributions, 248
STL, 51
  vector, 51
stream object, 97, 100
stringstream, 100
struct, 109, 110
structure, 109
switch, 43
system clock, 246

terminate program, 99
textttstd::
  pair, 107
type, 15
  bool, 39
  char, 89
  double, 15
  float, 21
  int, 15
  void, 64
  cast, 19
  conversion, 19
  modifier, 16

underflow, 21

variable
  declaration, 11, 15, 17
  global, 30
  initialisation, 17
  names, 11
  scope, 30
  type, 15
vector, 51
  initialisation, 53
  member functions, 54
  multi-dimensional, 53
Visual Studio
  build configurations, 10
  compiling, 9
  create new project, 8
  run options, 10

weighted lottery, 249
while, 58, 59