Updated for C++ 20

Discovering Modern C++

An Intensive Course for Scientists, Engineers, and Programmers

Peter Gottschling

SECOND EDITION

C++ In-Depth Series   Bjarne Stroustrup

Discovering Modern C++

Second Edition

A Tour of C++
Second Edition
Bjarne Stroustrup
Modern C++ Design
Generic Programming and Design Patterns Applied
Andrei Alexandrescu
C++ Coding Standards
101 Rules, Guidelines, and Best Practices
Herb Sutter
Andrei Alexandrescu
C++ Template Metaprogramming
Concepts, Tools, and Techniques from Boost and Beyond
David Abrahams
Aleksey Gurtovoy
C++ In-Depth Series • Bjarne Stroustrup

Discovering Modern C++

An Intensive Course for Scientists, Engineers, and Programmers

Second Edition

Peter Gottschling

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Library of Congress Control Number: 2021944389

ISBN-13: 978-0-13-667764-2
ISBN-10: 0-13-667764-9

ScoutAutomatedPrintCode

To my wonderful children, Vincent, Daniel, Yanis, and Anissa

This page intentionally left blank

Contents

This page intentionally left blank

Preface

"The world is built on C++ (and its C subset)."

—Herb Sutter

The infrastructures at Google, Amazon, and Facebook are built using components and services designed and implemented in the C++ programming language. A considerable portion of the technology stack of operating systems, networking equipment, and storage systems is implemented in C++. In telecommunication systems, almost all landline and cellular phone connections are orchestrated by C++ software. And key components in industrial and transportation systems, including automated toll collection systems, and autonomous cars, trucks, and autobuses depend on C++.

In science and engineering, most high-quality software packages today are implemented in C++. The strength of the language is evidenced when projects exceed a certain size and data structures and algorithms become non-trivial. It is no wonder that many—if not most—simulation software programs in computational science are realized today in C++; these include FLUENT, Abaqus, deal.II, FEniCS, OpenFOAM, and G+Smo. Even embedded systems are increasingly realized in C++ thanks to more powerful embedded processors and improved compilers. And the new application domains of the Internet of Things (IoT) and embedded edge intelligence are all driven by C++ platforms such as TensorFlow, Caffe2, and CNTK.

Core services you use every day are based on C++: your cell phone, your car, communication and industrial infrastructure, and key elements in media and entertainment services all contain C++ components. C++ services and applications are omnipresent in modern society. The reason is simple. The C++ language has progressed with its demands, and in many ways leads the innovations in programming productivity and execution efficiency. Both attributes make it the language of choice for applications that need to run at scale.

Reasons to Learn C++

Like no other language, C++ masters the full spectrum from programming sufficiently close to the hardware on one end to abstract high-level programming on the other. The lower-level programming—like user-definable memory management—empowers you as a programmer to understand what really happens during execution, which in turn helps you understand the behavior of programs in other languages. In C++ you can write extremely efficient programs that can only be slightly outperformed by code written in machine language with ridiculous effort. However, you should wait a little with the hardcore performance tuning and focus first on clear and expressive software.

This is where the high-level features of C++ come into play. The language supports a wide variety of programming paradigms directly: object-oriented programming (Chapter 6), generic programming (Chapter 3), meta-programming (Chapter 5), concurrent programming (§4.6), and procedural programming (§1.5), among others.

Several programming techniques—like RAII (§2.4.2.1) and expression templates (§5.3)—were invented in and for C++. As the language is so expressive, it was often possible to establish these new techniques without changing the language. And who knows, maybe one day you will invent a new technique.

Reasons to Read This Book

The material in the book has been tested on real humans. The author taught his class, "C++ for Scientists," over three years (i.e., three times two semesters). The students, mostly from the mathematics department, plus some from the physics and engineering departments, often did not know C++ before the class and were able to implement advanced techniques like expression templates (§5.3) by the end of the course. You can read this book at your own pace: straight to the point by following the main path or more thoroughly by reading additional examples and background information in Appendix A.

The Beauty and the Beast

C++ programs can be written in so many ways. In this book, we will lead you smoothly to the more sophisticated styles. This requires the use of advanced features that might be intimidating at first but will become less so once you get used to them. Actually, high-level programming is not only more widely applicable but is usually equally or more efficient and readable.

We will give you a first impression with a simple example: gradient descent with constant step size. The principle is extremely simple: we compute the steepest descent of $f(x)$ with its gradient, say $g(x)$, and follow this direction with fixed-size steps to the next local minimum. Even the algorithmic pseudo-code is as simple as this description:

Algorithm 1: Gradient descent algorithm

```
Input: Start value $x$, step size $s$, termination criterion $\varepsilon$, function $f$, gradient $g$
Output: Local minimum $x$
1 do
2     $x = x - s \cdot g(x)$
3 while $|\Delta f(x)| \geqslant \varepsilon$ ;
```

For this simple algorithm, we wrote two quite different implementations. Please have a look and let it sink in without trying to understand the technical details.

```cpp
void gradient_descent(double* x,
    double* y, double s, double eps,
    double(*f)(double, double),
    double(*gx)(double, double),
    double(*gy)(double, double))
{
    double val= f(*x, *y), delta;
    do {
        *x-= s * gx(*x, *y);
        *y-= s * gy(*x, *y);
        double new_val= f(*x, *y);
        delta= abs(new_val - val);
        val= new_val;
    } while (delta > eps);
}
```

```cpp
template <typename Value, typename P1,
          typename P2, typename F,
          typename G>
Value gradient_descent(Value x, P1 s,
    P2 eps, F f, G g)
{
    auto val= f(x), delta= val;
    do {
        x-= s * g(x);
        auto new_val= f(x);
        delta= abs(new_val - val);
        val= new_val;
    } while (delta > eps);
    return x;
}
```

At first glance, they look pretty similar, and we will tell you soon which one we like more. The first version is in principle pure C, i.e., compilable with a C compiler too. The benefit is that what is optimized is directly visible: a 2D function with `double` values (indicated by the **highlighted** function parameters). We prefer the second version as it is more widely usable: two functions of arbitrary dimension with arbitrary value types (visible by the **marked** type and function parameters). Surprisingly, the versatile implementation is not less efficient. To the contrary, the functions given for `F` and `G` may be inlined (see §1.5.3) so that the function call overhead is saved, whereas the explicit use of (ugly) function pointers in the first version makes this code acceleration difficult for the compiler.

A longer example comparing old and new styles is found in Appendix A (§A.1) for the really patient reader. There, the benefit of modern programming is much more evident than in the toy example here. But we do not want to hold you back too long with preliminary skirmishing.

Languages in Science and Engineering

"It would be nice if every kind of numeric software could be written in C++ without loss of efficiency, but unless something can be found that achieves this without compromising the C++-type system it may be preferable to rely on Fortran, assembler or architecture-specific extensions."

—Bjarne Stroustrup

Scientific and engineering software is written in different languages, and which one is the most appropriate depends on the goals and available resources:

- Math tools like MATLAB, Mathematica, or R are excellent when we can use their existing algorithms. When we implement our own algorithms with fine-grained (e.g., scalar) operations, we will experience a significant decrease in performance. This might not be an issue when the problems are small or the user is infinitely patient; otherwise we should consider alternative languages.

- Python is excellent for rapid software development and already contains scientific libraries like “scipy” and “numpy,” and applications based on these libraries (often implemented in C and C++) are reasonably efficient. Again, user-defined algorithms from fine-grained operations pay a performance penalty. Python is an excellent way to implement small and medium-sized tasks efficiently. When projects grow sufficiently large, it becomes increasingly important that the compiler is stricter (e.g., assignments are rejected when the argument types do not match).

- Fortran is also great when we can rely on existing, well-tuned operations like dense matrix operations. It is well suited to accomplishing old professors’ homework (because they only ask for what is easy in Fortran). Introducing new data structures is, in the author’s experience, quite cumbersome, and writing a large simulation program in Fortran is quite a challenge—today only done voluntarily by a shrinking minority.

- C allows for good performance, and a large amount of software is written in C. The core language is relatively small and easy to learn. The challenge is to write large and bug-free software with the simple and dangerous language features, especially pointers (§1.8.2) and macros (§1.9.2.1). The last standard was released in 2017 and thus is named C17. Most C features—but not all—were sooner or later introduced into C++.

- Languages like Java, C#, and PHP are probably good choices when the main component of the application is a web or graphic interface and not too many calculations are performed.

- C++ shines particularly when we develop large, high-quality software with good performance. Nonetheless, the development process does not need to be slow and painful. With the right abstractions at hand, we can write C++ programs quite rapidly. We are optimistic that in future C++ standards, more scientific libraries will be included.

Evidently, the more languages we know, the more choice we have. Moreover, the better we know those languages, the more educated our choice will be. In addition, large projects often contain components in different languages, whereas in most cases at least the performance-critical kernels are realized in C or C++. All this said, learning C++ is an intriguing journey, and having a deep understanding of it will make you a great programmer in any case.

Typographical Conventions

New terms are set in *clear blue and italic*. C++ sources are printed `blue and monospace`. Important details are marked in **boldface**. Classes, functions, variables, and constants are lowercase, optionally containing underscores. An exception is matrices, which are usually named with a single capital letter. Template parameters and concepts start with a capital letter and may contain further capitals (CamelCase). Program output and commands are in `light blue typewriter font`.

Programs requiring C++11, C++14, C++17, or C++20 features are marked with corresponding margin boxes. Several programs making light use of a C++11 feature that is easily substituted by a C++03 expression are not explicitly marked.

Except for very short code illustrations, all programming examples in this book were tested on at least one compiler and in most cases on three compilers: `g++`, `clang++`, and Visual Studio. All examples are kept as short as possible for easier understanding. To this end, we don't incorporate all features that would have been used in comparable production code. Obviously, we minimize the usage of features not introduced yet. It is probably a good idea to review the examples after finishing reading the book and ask yourself what you would code differently based on all the new knowledge you acquired.

For the C++20 examples, we recommend that you try them to see whether they work on your system. Most of the new features weren't fully supported by all compilers at the time of this writing. Even the existing compiler support might not have been 100 percent correct at that time. For some new libraries, we used prototype implementations when no standard version was available on any compiler (e.g., format library).

⇒ `directory/source_code.cpp`

The location of the program example relevant to the discussed topic is indicated by an arrow and the paths of the complete programs at the beginning of the paragraph or section. All programs are available on GitHub in the public repository `https://github.com/petergottschling/dmc3` and can thus be cloned by:

```
git clone https://github.com/petergottschling/dmc3.git
```

On Windows, it might be more convenient to use TortoiseGit; see `tortoisegit.org`.

Register your copy of *Discovering Modern C++, Second Edition*, on the InformIT site for convenient access to updates and/or corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780136677642) and click Submit. Look on the Registered Products tab for an Access Bonus Content link next to this product, and follow that link to access any available bonus materials. If you would like to be notified of exclusive offers on new editions and updates, please check the box to receive email from us.

Acknowledgments

Starting chronologically, the author would like to thank Karl Meerbergen and his colleagues for the initial 80-page text used as a block lecture at KU Leuven taught by Karl and me in 2008. Over time, most passages have been rewritten, but the original document provided the initial momentum that was essential for the entire writing process. I truly owe Mario Mulansky a great debt for contributing Section 7.1, Implementation of ODE Solvers.

I am tremendously grateful to Jan Christiaan van Winkel and Fabio Fracassi, who back-checked every tiny detail of the manuscript's first edition and gave many suggestions toward standard compliance and comprehensibility. I am equally indebted to the proofreaders of the second edition: Chuck Allison, Sean Parent, and especially Marc Gregoire, who thoroughly checked each and every detail of the book. In addition, Jan Christiaan van Winkel provided invaluable assistance during the final production of this edition and deserves special recognition.

I would especially like to thank Bjarne Stroustrup for giving strategic tips to shape the book, establishing contact with Addison-Wesley, generously allowing me to reuse his well-prepared material, and (lest we forget) for creating C++. All of these people pushed me hard to update my ideas with the latest language features as much as possible.

In addition, I thank Karsten Ahnert for his recommendations as well as Markus Abel and Theodore Omtzigt for helping me eliminate verbosity in the Preface.

When I was looking for an interesting random number application for Section 4.2.2.6, Jan Rudl suggested the share price evolution he used in his class [60].

I am obliged to Technische Universität Dresden, which let me teach C++ in the mathematics department for three years, and I appreciate the constructive feedback I received from all the students in this course. Likewise, I am grateful to those who attended my C++ trainings.

I am deeply indebted to my editor, Greg Doench, for accepting my half-serious, half-casual writing style in this book, for enduring long-lasting discussions about strategic decisions until we were both satisfied, and for providing professional support, without which this book would never have reached publication.

Last but not least, I wholeheartedly thank my children, Yanis, Anissa, Vincent, and Daniel, for sacrificing so much of our time together so I could work on the book instead.

This page intentionally left blank

About the Author

Peter Gottschling's professional passion is writing leading-edge scientific software, and he hopes to infect many readers with this virus. This vocation resulted in writing the Matrix Template Library 4 and 5 as well as coauthoring other libraries including the Boost Graph Library. These programming experiences were shared in several C++ courses at universities and in professional training sessions—finally leading to this book.

He is a member of the ISO C++ standards committee, chair of Germany's programming language standards committee, and founder of the C++ User Group in Dresden. In his young and wild years at TU Dresden, he studied computer science and mathematics in parallel, the latter up to a bachelor's-like degree while finishing the former with a PhD. After an odyssey through academic institutions, he founded his own company, SimuNova, and settled with the younger two of his four children in his hometown of Leipzig.

This page intentionally left blank

Chapter 1

C++ Basics

"To my children:
Never make fun of having to help me with computer stuff.
I taught you how to use a spoon."

—Sue Fitzmaurice

In this first chapter, we will guide you through the fundamental features of C++. As for the entire book, we will look at them from different angles but we will not try to expose every possible detail—which is not feasible anyway. For more detailed questions on specific features, we recommend the online reference at `http://en.cppreference.com`.

1.1 Our First Program

As an introduction to the C++ language, let us look at the following example:

```
#include <iostream>

int main ()
{
    std::cout ≪ "The answer to the Ultimate Question of Life,\n"
              ≪ "the Universe, and Everything is:"
              ≪ std::endl ≪ 6 * 7 ≪ std::endl;
    return 0;
}
```

which yields

```
The answer to the Ultimate Question of Life,
the Universe, and Everything is:
42
```

according to Douglas Adams [2]. This short example already illustrates several features of C++:

- Input and output are not part of the core language but are provided by the library. They must be included explicitly; otherwise we cannot read or write.
- The standard I/O has a stream model and is therefore named `<iostream>`. To enable its functionality, we `#include <iostream>` in the first line.

- Every C++ program starts by calling the function `main`. It does `return` an integer value where 0 represents a successful termination.
- Braces `{}` denote a block/group of code (also called a compound statement).
- `std::cout` and `std::endl` are defined in `<iostream>`. The former is an output stream that allows to print text on the screen. `std::endl` terminates a line. We can also go to a new line with the special character `\n`.
- The operator $\ll$ can be used to pass objects to an output stream such as `std::cout` for performing an output operation. Please note that the operator is denoted as two less-than signs (`<<`) in programs. For a more elegant print image we use a single French guillemet instead in our listings.
- `std::` denotes that the type or function is used from the standard *Namespace*. Namespaces help us organize our names and deal with naming conflicts.
- Many examples in this book use types from the `std` namespace without the qualifying prefix `std::`. To compile such examples, it is assumed that after including the header files the program contains the declaration:

  ```
  using namespace std;
  ```

  The details of namespaces will be discussed in Section 3.2.1.
- String constants (more precisely, literals) are enclosed in double quotes.
- The expression `6 * 7` is evaluated and passed as an integer to `std::cout`. In C++, every expression has a type. Sometimes we as programmers have to declare the type explicitly and other times the compiler can deduce it for us. `6` and `7` are literal constants of type `int` and accordingly their product is `int` as well.

Before you continue reading, we strongly recommend that you compile and run this little program on your computer. Once it compiles and runs, you can play with it a little bit, for example, adding more operations and output (and looking at some error messages). Finally, the only way to really learn a language is to use it. If you already know how to use a compiler or even a C++ IDE, you can skip the remainder of this section.

Linux: Every distribution provides at least the GNU C++ compiler—usually already installed (see the short intro in Section B.1). Say we call our program `hello42.cpp`; it is easily compiled with the command

```
g++ hello42.cpp
```

Following a last-century tradition, the resulting binary is called `a.out` by default. One day we might have more than one program, and then we can use more meaningful names with the output flag:

```
g++ hello42.cpp -o hello42
```

We can also use the build tool `make` (see §7.2.2.1 for an overview) that provides default rules for building binaries. Thus, we could call

```
make hello42
```

and `make` will look in the current directory for a similarly named program source. It will find `hello42.cpp`, and as `.cpp` is a standard file suffix for C++ sources, it will call the system's default C++ compiler. Once we have compiled our program, we can call it on the command line as

```
./hello42
```

Our binary can be executed without needing any other software, and we can copy it to another compatible Linux system[1] and run it there.

Windows: If you are running MinGW, you can compile in the same manner as under Linux. If you use Visual Studio, you will need to create a project first. To begin, the easiest way is to use the project template for a console application, as described, for instance, at `http://www.cplusplus.com/doc/tutorial/introduction/visualstudio`. When you run the program, you might only have a few milliseconds to read the output before the console closes.[2] To extend the reading phase to one second, simply insert the non-portable command `Sleep(1000);` and include `<windows.h>`. With C++11 or higher, the waiting phase can be implemented portably:

```
std::this_thread::sleep_for(std::chrono::seconds(1));
```

after including `<chrono>` and `<thread>`. Microsoft offers free versions of Visual Studio called "Community," which provide the support for the standard language like their professional counterparts. The difference is that the professional editions come with more developer libraries. Since those are not used in this book, you can use the Community version to try our examples.

IDE: Short programs like the examples in this book can be easily handled with an ordinary editor. In larger projects it is advisable to use an *Integrated Development Environment* to see where a function is defined or used, show the in-code documentation, search or replace names project-wide, et cetera. KDevelop is a free IDE from the KDE community written in C++. It is probably the most efficient IDE on Linux and integrates well with `git`, `subversion`, and `CMake`. Eclipse is developed in Java and perceivably slower. However, a lot of effort was recently put into it for improving the C++ support, and many developers are quite productive with it. Visual Studio is a very solid IDE allowing for productive development under Windows and in newer versions also supports an integration of `CMake` projects.

To find the most productive environment takes some time and experimentation and is of course subject to personal and collaborative taste. As such, it will also evolve over time.

1. Often the standard library is linked dynamically (cf. §7.2.1.4) and then its presence in the same version on the other system is part of the compatibility requirements.
2. Since VS 2019 the console will automatically pause.

1.2 Variables

C++ is a strongly typed language (in contrast to many scripting languages). This means that every variable has a type and this type never changes. A variable is declared by a statement beginning with a type followed by a variable name with optional initialization—or a list thereof:

```
int     i1= 2;                // Alignment for readability only
int     i2, i3= 5;            // Note: i2 is not initialized
float   pi= 3.14159;
double  x= -1.5e6;            // -1500000
double  y= -1.5e-6;           // -0.0000015
char    c1= 'a', c2= 35;
bool    cmp= i1 < pi,         // -> true
        happy= true;
```

The two slashes `//` here start a single-line comment; i.e., everything from the double slashes to the end of the line is ignored. In principle, this is all that really matters about comments. So as not to leave you with the feeling that something important on the topic is still missing, we will discuss it a little more in Section 1.9.1.

1.2.1 Intrinsic Types

The most fundamental types in C++ are the *Intrinsic Types* listed in Table 1–1. They are part of the core language and always available.

The first five types are integer numbers of nondecreasing length. For instance, `int` is at least as long as `short`; i.e., it is usually but not necessarily longer. The exact length of each type is implementation-dependent; e.g., `int` could be 16, 32, or 64 bits. All these types can be qualified as `signed` or `unsigned`. The former has no effect on integer numbers (except `char`) since they are `signed` by default.

When we declare an integer type as `unsigned`, we will have no negative values but twice as many positive ones (plus one when we consider zero as neither positive nor negative).

Table 1–1: Intrinsic Types

Name	Semantics
`char`	letter and very short integer number
`short`	rather short integer number
`int`	regular integer number
`long`	long integer number
`long long`	very long integer number
`unsigned`	unsigned versions of all the former
`signed`	signed versions of all the former
`float`	single-precision floating-point number
`double`	double-precision floating-point number
`long double`	long floating-point number
`bool`	boolean

signed and unsigned can be considered adjectives for the noun int with int as the default noun when only the adjective is declared. The same applies for the adjectives short, long, and long long.

The type char can be used in two ways: for letters and rather short numbers. Except for really exotic architectures, it almost always has a length of 8 bits. Thus, we can either represent values from -128 to 127 (signed) or from 0 to 255 (unsigned) and perform all numeric operations on them that are available for integers. When neither signed nor unsigned is declared, it depends on the implementation of the compiler which one is used. Using char or unsigned char for small numbers, however, can be useful when there are large containers of them.

Logic values are best represented as bool. A boolean variable can store true and false.

The non-decreasing length property applies in the same manner to floating-point numbers: float is shorter than or equally as long as double, which in turn is shorter than or equally as long as long double. Typical sizes are 32 bits for float, 64 bits for double, and 128 bits for long double.

1.2.2 Characters and Strings

As mentioned before, the type char can be used to store characters:

```
char c= 'f';
```

We can also represent any letter whose code fits into 8 bits. It can even be mixed with numbers; e.g., 'a' + 7 usually leads to 'h' depending on the underlying coding of the letters. We strongly recommend not playing with this since the potential confusion will likely lead to a perceivable waste of time.

From C we inherited the opportunity to represent strings as arrays of char.

```
char name[8]= "Herbert";
```

These old C strings all end with a binary 0 as a char value. If the 0 is missing, algorithms keep going until the next memory location with a 0-byte is found. Another big danger is appending to strings: name has no extra space and the additional characters overwrite some other data. Getting all string operations right—without corrupting memory or cutting off longer strings—is everything but trivial with these old strings. We therefore strongly recommend not using them except for literal values.

The C++ compiler distinguishes between single and double quotes: 'a' is the character "a" (it has type char) and "a" is an array with a binary 0 as termination (i.e., its type is const char[2]).

The much more convenient fashion to deal with string is by using the class string (which requires that we include <string>):

```
#include <string>

int main()
{
    std::string name= "Herbert";
}
```

C++ strings use dynamic memory and manage it themselves. So if we append more text to a string, we don't need to worry about memory corruption or cutting off strings:

```
name= name + ", our cool anti-hero"; // more on this later
```

Many current implementations also use optimization for short strings (e.g., to 16 bytes) that are not stored in dynamic memory, but directly in the `string` object itself. This optimization can significantly reduce the expensive memory allocation and release.

C++14 Since text in double quotes is interpreted as a `char` array, we need to be able to denote that the text should be considered a `string`. This is done with the suffix `s`, e.g., `"Herbert"s`.[3] Unfortunately, it took us until C++14 to enable this. An explicit conversion like `string("Herbert")` was always possible. A lightweight constant view on strings was added in C++17 that we will show in Section 4.4.5.

1.2.3 Declaring Variables

Advice
Declare variables as late as possible, usually right before using them the first time and whenever possible, but not before you can initialize them.

This makes programs more readable when they grow long. It also allows the compiler to use the memory more efficiently with nested scopes.

C++11 C++11 can deduce the type of a variable for us, e.g.:

```
auto i4= i3 + 7;
```

The type of `i4` is the same as that of `i3 + 7`, which is `int`. Although the type is automatically determined, it remains the same, and whatever is assigned to `i4` afterward will be converted to `int`. We will see later how useful `auto` is in advanced programming. For simple variable declarations like those in this section, it is usually better to declare the type explicitly. `auto` will be discussed thoroughly in Section 3.4.

1.2.4 Constants

Syntactically, constants are like special variables in C++ with the additional attribute of constancy:

```
const int   ci1= 2;
const int   ci3;              // Error: no value
const float pi= 3.14159;
const char  cc= 'a';
const bool  cmp= ci1 < pi;
```

As they cannot be changed, it is mandatory to set their values in the declaration. The second constant declaration violates this rule, and the compiler will not tolerate such misbehavior.

3. As in many other examples, we assume here that the program contains `using namespace std`. It is also possible to import the suffixes only or just specific suffixes. We recommend, however, that you import the entire standard namespace while learning the language.

Constants can be used wherever variables are allowed—as long as they are not modified, of course. On the other hand, constants like those above are already known during compilation. This enables many kinds of optimizations, and the constants can even be used as arguments of types (we will come back to this later in §5.1.4).

1.2.5 Literals

Literals like `2` or `3.14` are typed as well. Simply put, integral numbers are treated as `int`, `long`, or `unsigned long` depending on the magnitude of the number. Every number with a dot or an exponent (e.g., `3e12` $\equiv 3 \cdot 10^{12}$) is considered a `double`.

Literals of other types can be written by adding a suffix from the following table:

Literal	Type
`2`	`int`
`2u`	`unsigned`
`2l`	`long`
`2ul`	`unsigned long`
`2.0`	`double`
`2.0f`	`float`
`2.0l`	`long double`

In most cases, it is not necessary to declare the type of literals explicitly since the implicit conversion (a.k.a. *Coercion*) between built-in numeric types usually sets the values at the programmer's expectation.

There are, however, four major reasons why we should pay attention to the types of literals.

Availability: The standard library provides a type for complex numbers where the type for the real and imaginary parts can be parameterized by the user:

```
std::complex<float> z(1.3, 2.4), z2;
```

Unfortunately, operations are only provided between the type itself and the underlying real type (and arguments are not converted here).[4] As a consequence, we cannot multiply `z` with an `int` or `double` but with `float`:

```
z2= 2 * z;        // Error: no int * complex<float>
z2= 2.0 * z;      // Error: no double * complex<float>
z2= 2.0f * z;     // Okay:  float * complex<float>
```

Ambiguity: When a function is overloaded for different argument types (§1.5.4), an argument like `0` might be ambiguous whereas a unique match may exist for a qualified argument like `0u`.

Accuracy: The accuracy issue comes up when we work with `long double`. Since the non-qualified literal is a `double`, we might lose digits before we assign it to a `long double` variable:

```
long double third1= 0.3333333333333333333;    // may lose digits
long double third2= 0.3333333333333333333l;   // accurate
```

4. Mixed arithmetic is implementable, however, as demonstrated at [19].

Nondecimal Numbers: Integer literals starting with a zero are interpreted as octal numbers, e.g.:

```
int o1= 042;          // int o1= 34;
int o2= 084;          // Error! No 8 or 9 in octals!
```

Hexadecimal literals can be written by prefixing them with `0x` or `0X`:

```
int h1= 0x42;         // int h1= 66;
int h2= 0xfa;         // int h2= 250;
```

C++14 C++14 introduces binary literals, which are prefixed with `0b` or `0B`:

```
int b1= 0b11111010;   // int b1= 250;
```

To improve readability of long literals, C++14 allows us to separate the digits with apostro- C++14 phes:

```
long                d=   6'546'687'616'861'129l;
unsigned long       ulx= 0x139'ae3b'2ab0'94f3;
int                 b=   0b101'1001'0011'1010'1101'1010'0001;
const long double pi=  3.141'592'653'589'793'238'462l;
```

C++17 Since C++17, we can even write hexadecimal floating-point literals:

```
float  f1= 0x10.1p0f; // 16.0625
double d2= 0x1ffp10;   // 523264
```

For these, introduced with the `p` character. The exponent is mandatory—thus we needed `p0` in the first example. Due to the suffix `f`, `f1` is a `float` storing the value $16^1 + 16^{-1} = 16.0625$. These literals involve three bases: the pseudo-mantissa is a hexadecimal scaled with powers of 2 whereby the exponent is given as a decimal number. Thus, `d2` is $511 \times 2^{10} = 523264$. Hexadecimal literals seem, admittedly, a little nerdy at the beginning but they allow us to declare binary floating-point values without rounding errors.

String literals are typed as arrays of `char`:

```
char s1[]= "Old C style"; // better not
```

However, these arrays are everything but convenient, and we are better off with the true `string` type from the library `<string>`. It can be created directly from a string literal:

```
#include <string>
std::string s2= "In C++ better like this";
```

Very long text can be split into multiple sub-strings:

```
std::string s3= "This is a very long and clumsy text "
                "that is too long for one line.";
```

C++14 Although `s2` and `s3` have type `string`, they are still initialized with literals of type `const char[]`. This is not a problem here but might be in other situations where the type is deduced by the compiler. Since C++14, we can directly create literals of type `string` by appending an `s`:

```
f("I'm not a string");     // literal of type const char[]
f("I'm really a string"s); // literal of type string
```

As before, we assume that the namespace `std` is used. To not import the entire standard namespace, we can use sub-spaces thereof, i.e. writing at least one of the following lines:

```
using namespace std::literals;
using namespace std::string_literals;
using namespace std::literals::string_literals;
```

For more details on literals, see for instance [62, §6.2]. We will show how to define your own literals in Section 2.3.6.

1.2.6 Non-narrowing Initialization

C++11

Say we initialize a `long` variable with a long number:

```
long l2= 1234567890123;
```

This compiles just fine and works correctly—when `long` takes 64 bits as on most 64-bit platforms. When `long` is only 32 bits long (we can emulate this by compiling with flags like `-m32`), the value above is too long. However, the program will still compile (maybe with a warning) and runs with another value, e.g., where the leading bits are cut off.

C++11 introduces an initialization that ascertains that no data is lost or, in other words, that the values are not *Narrowed*. This is achieved with the *Uniform Initialization* or *Braced Initialization* that we only touch upon here and expand on in Section 2.3.4. Values in braces cannot be narrowed:

```
long l= {1234567890123};
```

Now the compiler will check whether the variable `l` can hold the value on the target architecture. When using the braces, we can omit the equals sign:

```
long l{1234567890123};
```

The compiler's narrowing protection allows us to verify that values do not lose precision in initializations. Whereas an ordinary initialization of an `int` by a floating-point number is allowed due to implicit conversion:

```
int i1= 3.14;        // compiles despite narrowing (our risk)
int i1n= {3.14};     // Narrowing ERROR: fractional part lost
```

The new initialization form in the second line forbids this because it cuts off the fractional part of the floating-point number. Likewise, assigning negative values to unsigned variables or constants is tolerated with traditional initialization but denounced in the new form:

```
unsigned u2= -3;     // Compiles despite narrowing (our risk)
unsigned u2n= {-3};  // Narrowing ERROR: no negative values
```

In the previous examples, we used literal values in the initializations and the compiler checks whether a specific value is representable with that type:

```
float f1= {3.14};    // okay
```

Well, the value 3.14 cannot be represented with absolute accuracy in any binary floating-point format, but the compiler can set `f1` to the value closest to 3.14. When a float is initialized from a double variable (not a literal), we have to consider all possible double values and whether they are all convertible to float in a loss-free manner.

```
double d;
...
float f2= {d};         // narrowing ERROR
```

Note that the narrowing can be mutual between two types:

```
unsigned u3= {3};
int      i2= {2};

unsigned u4= {i2};   // narrowing ERROR: no negative values
int      i3= {u3};   // narrowing ERROR: not all large values
```

The types signed int and unsigned int have the same size, but not all values of each type are representable in the other.

1.2.7 Scopes

Scopes determine the lifetime and visibility of (nonstatic) variables and constants and contribute to establishing a structure in our programs.

1.2.7.1 Global Definition

Every variable that we intend to use in a program must have been declared with its type specifier at an earlier point in the code. A variable can be located in either the global or local scope. A global variable is declared outside all functions. After their declaration, global variables can be referred to from anywhere in the code, even inside functions. This sounds very handy at first because it makes the variables easily available, but when your software grows, it becomes more difficult and painful to keep track of the global variables' modifications. At some point, every code change bears the potential of triggering an avalanche of errors.

Advice
Do not use global variables.

If you do use them, sooner or later you will regret it because they can be accessed from the entire program and it is therefore extremely tedious to keep track of where global variables are changed—and when and how.

Global constants like

```
const double pi= 3.14159265358979323846264338327950288419716939;
```

are fine because they cannot cause side effects.

1.2.7.2 Local Definition

A local variable is declared within the body of a function. Its visibility/availability is limited to the { }-enclosed block of its declaration. More precisely, the scope of a variable starts with its declaration and ends with the closing brace of the declaration block.

If we define π in the function `main`:

```
int main ()
{
    const double pi= 3.141592653589793238462643383279502884197169399;
    std::cout ≪ "pi is " ≪ pi ≪ ".\n";
}
```

the variable `pi` only exists in the `main` function. We can define blocks within functions and within other blocks:

```
int main ()
{
    {
        const double pi= 3.141592653589793238462643383279502884197169399;
    }
    std::cout ≪ "pi is " ≪ pi ≪ ".\n"; // ERROR: pi is out of scope
}
```

In this example, the definition of `pi` is limited to the block within the function, and an output in the remainder of the function is therefore an error:

```
    ≫pi≪ is not defined in this scope.
```

because π is *Out of Scope*.

1.2.7.3 Hiding

When a variable with the same name exists in nested scopes, only one variable is visible. The variable in the inner scope hides the homonymous variables in the outer scopes (causing a warning in many compilers). For instance:

```
int main ()
{
    int a= 5;              // define a#1
    {
        a= 3;              // assign a#1, a#2 is not defined yet
        int a;             // define a#2
        a= 8;              // assign a#2, a#1 is hidden
        {
            a= 7;          // assign a#2
        }
    }                      // end of a#2's scope
    a= 11;                 // assign to a#1 (a#2 out of scope)

    return 0;
}
```

Due to hiding, we must distinguish the lifetime and the visibility of variables. For instance, `a#1` lives from its declaration until the end of the `main` function. However, it is only visible from its declaration until the declaration of `a#2` and again after closing the block containing `a#2`. In fact, the visibility is the lifetime minus the time when it is hidden. Defining the same variable name twice in one scope is an error.

The advantage of scopes is that we do not need to worry about whether a variable is already defined somewhere outside the scope. It is just hidden but does not create a conflict.[5] Unfortunately, the hiding makes the homonymous variables in the outer scope inaccessible. We can cope with this to some extent with clever renaming. A better solution, however, to manage nesting and accessibility is namespaces; see Section 3.2.1.

`static` variables are the exception that confirms the rule: they live until the end of the execution but are only visible in the scope. We are afraid that their detailed introduction is more distracting than helpful at this stage and have postponed the discussion to Section A.2.1.

1.3 Operators

C++ is rich in built-in operators. There are different kinds of operators:

- Computational:
  - Arithmetic: `++`, `+`, `*`, `%`, ...
  - Boolean:
    - Comparison: `<=`, `!=`, ...
    - Logic: `&&` and `||`
  - Bitwise: $\sim$, $\ll$ and $\gg$, `&`, `^`, and `|`
- Assignment: `=`, `+=`, ...
- Program flow: function call, `?:`, and `,`
- Memory handling: `new` and `delete`
- Access: `.`, `->`, `[ ]`, `*`, ...
- Type handling: `dynamic_cast`, `typeid`, `sizeof`, `alignof`, ...
- Error handling: `throw`

This section will give you an overview of the operators. Some operators are better described elsewhere in the context of the appropriate language feature; e.g., scope resolution is best explained together with namespaces. Most operators can be overloaded for user types; i.e., we can decide which calculations are performed when arguments of our types appear in expressions.

At the end of this section (Table 1–8), you will find a concise table of operator precedence. It might be a good idea to print or copy this page and pin it next to your monitor; many people do so and almost nobody knows the entire priority list by heart. Neither should you hesitate to put parentheses around sub-expressions if you are uncertain about the priorities

5. As opposed to macros, an obsolete and reckless legacy feature from C that should be avoided at any price because it undermines all structure and reliability of the language.

or if you believe it will be more understandable for other programmers working with your sources. If you ask your compiler to be pedantic, it often takes this job too seriously and prompts you to add surplus parentheses assuming you are overwhelmed by the precedence rules. In Section C.2, we will give you a complete list of all operators with brief descriptions and references.

1.3.1 Arithmetic Operators

Table 1–2 lists the arithmetic operators available in C++. We have sorted them by their priorities, but let us look at them one by one.

Table 1–2: Arithmetic Operators

Operation	Expression
Post-increment	`x++`
Post-decrement	`x--`
Pre-increment	`++x`
Pre-decrement	`--x`
Unary plus	`+x`
Unary minus	`-x`
Multiplication	`x * y`
Division	`x / y`
Modulo	`x % y`
Addition	`x + y`
Subtraction	`x - y`

The first kinds of operations are increment and decrement. These operations can be used to increase or decrease a number by 1. As they change the value of the number, they only make sense for variables and not for temporary results, for instance:

```
int i= 3;
i++;                // i is now 4
const int j= 5;
j++;                // Error: j is constant
(3 + 5)++;          // Error: 3 + 5 is only a temporary
```

In short, the increment and decrement operations need something that is modifiable and addressable. The technical term for an addressable data item is *lvalue* (more formally expressed in Definition C–1 in Appendix C). In our code snippet above, this is true for `i` only. In contrast to it, `j` is constant and `3 + 5` is not addressable.

Both notations—prefix and postfix—have the effect on a variable that they add or subtract 1 from it. The value of an increment and decrement expression is different for prefix and postfix operators: the prefix operators return the modified value and postfix the old one, e.g.:

```
int i= 3, j= 3;
int k= ++i + 4;    // i is 4, k is 8
int l= j++ + 4;    // j is 4, l is 7
```

At the end, both `i` and `j` are 4. However in the calculation of `l`, the old value of `j` was used while the first addition used the already incremented value of `i`.

In general, it is better to refrain from using increment and decrement in mathematical expressions and to replace it with `j+1` and the like or to perform the in/decrement separately. It is easier for human readers to understand and for the compiler to optimize when mathematical expressions have no *Side Effects*. We will see quite soon why (§1.3.12).

The unary minus negates the value of a number:

```
int i= 3;
int j= -i;          // j is -3
```

The unary plus has no arithmetic effect on standard types. For user types, we can define the behavior of both unary plus and minus. As shown in Table 1–2, these unary operators have the same priority as pre-increment and pre-decrement.

The operations `*` and `/` are naturally multiplication and division, and both are defined on all numeric types. When both arguments in a division are integers, the fractional part of the result is truncated (rounding toward zero). The operator `%` yields the remainder of the integer division. Thus, both arguments should have an integral type.

Last but not least, the operators `+` and `-` between two variables or expressions symbolize addition and subtraction.

The semantic details of the operations—how results are rounded or how overflow is handled—are not specified in the language. For performance reasons, C++ leaves this typically to the underlying hardware.

In general, unary operators have higher priority than binary. On the rare occasions that both postfix and prefix unary notations have been applied, postfix notations are prioritized over prefix notations.

Among the binary operators, we have the same behavior that we know from math: multiplication and division precede addition and subtraction and the operations are left associative, i.e.:

```
x - y + z
```

is always interpreted as

```
(x - y) + z
```

Something really important to remember: the order of evaluation of the arguments is not defined. For instance:

```
int i= 3, j= 7, k;
k= f(++i) + g(++i) + j;
```

In this example, associativity guarantees that the first addition is performed before the second. But whether the expression `f(++i)` or `g(++i)` is computed first depends on the compiler implementation. Thus, `k` might be either `f(4) + g(5) + 7` or `f(5) + g(4) + 7` (or even `f(5) + g(5) + 7` when both increments are executed before the function call). Furthermore, we cannot assume that the result is the same on a different platform. In general, it is dangerous to modify values within expressions. It works under some conditions, but we always have to test it and pay enormous attention to it. Altogether, our time is better spent by typing some extra letters and doing the modifications separately. More about this topic in Section 1.3.12.

⇒ c++03/num_1.cpp

With these operators, we can write our first (complete) numeric program:

```
#include <iostream>

int main ()
{
    const float r1= 3.5, r2 = 7.3, pi = 3.14159;

    float area1 = pi * r1*r1;
    std::cout ≪ "A circle of radius " ≪ r1 ≪ " has area "
              ≪ area1 ≪ "." ≪ std::endl;

    std::cout ≪ "The average of " ≪ r1 ≪ " and " ≪ r2 ≪ " is "
              ≪ (r1 + r2) / 2 ≪ "." ≪ std::endl;
}
```

When the arguments of a binary operation have different types, one or both arguments are automatically converted (coerced) to a common type according to the rules in §C.3.

The conversion may lead to a loss of precision. Floating-point numbers are preferred over integer numbers, and evidently the conversion of a 64-bit `long` to a 32-bit `float` yields an accuracy loss; even a 32-bit `int` cannot always be represented correctly as a 32-bit `float` since some bits are needed for the exponent. There are also cases where the target variable could hold the correct result but the accuracy was already lost in the intermediate calculations. To illustrate this conversion behavior, let us look at the following example:

```
long l=  1234567890123;
long l2= l + 1.0f - 1.0;   // imprecise
long l3= l + (1.0f - 1.0); // correct
```

This leads on the author's platform to

```
l2 = 1234567954431
l3 = 1234567890123
```

In the case of `l2` we lose accuracy due to the intermediate conversions, whereas `l3` was computed correctly. This is admittedly an artificial example, but you should be aware of the risk of imprecise intermediate results. Especially with large calculations the numerical algorithms must be carefully chosen to prevent the errors from building up. The issue of inaccuracy will fortunately not bother us in the next section.

1.3.2 Boolean Operators

Boolean operators are logical and relational operators. Both return `bool` values as the name suggests. These operators and their meaning are listed in Table 1–3, grouped by precedence.

Binary relational and logical operators are preceded by all arithmetic operators. This means that an expression like `4 >= 1 + 7` is evaluated as if it were written `4 >= (1 + 7)`. Conversely, the unary operator `!` for logic negation is prioritized over all binary operators.

Table 1–3: Boolean Operators

Operation	Expression
Not	`!b`
Three-way comparison (C++20)	`x <=> y`
Greater than	`x > y`
Greater than or equal to	`x >= y`
Less than	`x < y`
Less than or equal to	`x <= y`
Equal to	`x == y`
Not equal to	`x != y`
Logical AND	`b && c`
Logical OR	`b || c`

The boolean operators also have keywords, like `not`, `and`, `or`, and `xor`. There are even keywords for assignments, like `or_eq` for `|=`. We usually don't use them for their paleolithic look but there is one exception: `not` can make expressions far more readable. When negating something that starts with "i" or "l", the exclamation point is easily overseen. A space already helps, but the keyword makes the negation even more visible:

```
big= !little;       // You knew before there's an !
big= not little;    // Much easier to spot, though
```

Although these keywords have been available from the beginning of standard C++, Visual Studio still doesn't support them unless you compile with `/permissive-` or `/Za`.

In old (or old-fashioned) code, you might see logical operations performed on `int` values. Please refrain from this: it is less readable and subject to unexpected behavior.

Advice
Always use `bool` for logical expressions.

Please note that comparisons cannot be chained like this:

```
bool in_bound= min <= x <= y <= max;      // Syntax error
```

Instead we need the more verbose logical reduction:

```
bool in_bound= min <= x && x <= y && y <= max;
```

In the following section, we will see similar operators.

1.3.3 Bitwise Operators

These operators allow us to test or manipulate single bits of integral types. They are important for system programming but less so for modern application development. Table 1–4 lists these operators by precedence.

The operation `x ≪ y` shifts the bits of `x` to the left by `y` positions. Conversely, `x ≫ y` moves `x`'s bits `y` times to the right.[6] In most cases, 0s are moved (except for negative `signed` values) to the right where they are implementation defined.

6. Again we compress the double less-than and larger-than symbols to French guillemets for nicer printing.

Table 1–4: Bitwise Operators

Operation	Expression
One's complement	`~x`
Left shift	`x << y`
Right shift	`x >> y`
Bitwise AND	`x & y`
Bitwise exclusive OR	`x ^ y`
Bitwise inclusive OR	`x | y`

The bitwise AND can be used to test a specific bit of a value. Bitwise inclusive OR can set a bit and exclusive OR flips it. Although these operations are less important in scientific applications, we will use them in Section 3.5.1 for algorithmic entertainment.

1.3.4 Assignment

The value of an object (modifiable `lvalue`) can be set by an assignment:

```
object= expr;
```

When the types do not match, `expr` is converted to the type of `object` if possible. The assignment is right-associative so that a value can be successively assigned to multiple objects in one expression:

```
o3= o2= o1= expr;
```

Speaking of assignments, the author will now explain why he left-justifies the symbol. Most binary operators are symmetric in the sense that both arguments are values. In contrast, an assignment must have a modifiable variable on the left-hand side whereby the right-hand side can be an arbitrary expression (with an appropriate value). While other languages use asymmetric symbols (e.g., `:=` in Pascal), the author uses an asymmetric spacing in C++.

The compound assignment operators apply an arithmetic or bitwise operation to the object on the left side with the argument on the right side; for instance, the following two operations are equivalent:

```
a+= b;          // corresponds to
a=  a + b;
```

All assignment operators have a lower precedence than every arithmetic or bitwise operation so the right-hand side expression is always evaluated before the compound assignment:

```
a*= b + c;      // corresponds to
a=  a * (b + c);
```

The assignment operators are listed in Table 1–5. They are all right-associative and of the same priority.

1.3.5 Program Flow

There are three operators to control the program flow. First, a function call in C++ is handled like an operator. For a detailed description of functions and their calls, see Section 1.5.

Table 1–5: Assignment Operators

Operation	Expression
Simple assignment	`x= y`
Multiply and assign	`x*= y`
Divide and assign	`x/= y`
Modulo and assign	`x%= y`
Add and assign	`x+= y`
Subtract and assign	`x-= y`
Shift left and assign	`x<<= y`
Shift right and assign	`x>>= y`
AND and assign	`x&= y`
Inclusive OR and assign	`x|= y`
Exclusive OR and assign	`x^= y`

The conditional operator `c ? x : y` evaluates the condition `c`, and when it is true the expression has the value of `x`, otherwise `y`. It can be used as an alternative to branches with `if`, especially in places where only an expression is allowed and not a statement; see Section 1.4.3.1.

A very special operator in C++ is the *Comma Operator* that provides a sequential evaluation. The meaning is simply evaluating first the subexpression to the left of the comma and then that to the right of it. The value of the whole expression is that of the right subexpression:

```
3 + 4, 7 * 9.3
```

The result of the expression is 65.1 and the computation of the first subexpression is entirely irrelevant in this case. The subexpressions can contain the comma operator as well so that arbitrarily long sequences can be defined. With the help of the comma operator, one can evaluate multiple expressions in program locations where only one expression is allowed. A typical example is the increment of multiple indices in a `for`-loop (§1.4.4.2):

```
++i, ++j
```

When used as a function argument, the comma expression needs surrounding parentheses; otherwise the comma is interpreted as separation of function arguments.

1.3.6 Memory Handling

The operators `new` and `delete` allocate and deallocate memory, respectively. We postpone their description to Section 1.8.2 since discussing these operators before talking about pointers makes no sense.

1.3.7 Access Operators

C++ provides several operators for accessing substructures, for referring—i.e., taking the address of a variable—and dereferencing—i.e., accessing the memory referred to by an address. They are listed in Table 1–6. We will demonstrate in Section 2.2.3 how to use them, after we introduce pointers and classes.

Table 1–6: Access Operators

Operation	Expression	Reference
Member selection	`x.m`	§2.2.3
Dereferred member selection	`p->m`	§2.2.3
Subscripting	`x[i]`	§1.8.1
Dereference	`*x`	§1.8.2
Address-of	`&x`	§1.8.2
Member dereference	`x.*q`	§2.2.3
Dereferred member dereference	`p->*q`	§2.2.3

1.3.8 Type Handling

The operators for dealing with types will be presented in Chapter 5 when we will write compile-time programs that work on types. Right now we only list them in Table 1–7.

Table 1–7: Type-Handling Operators

Operation	Expression
Run-time type identification	`typeid(x)`
Identification of a type	`typeid(t)`
Size of object	`sizeof(x)` or `sizeof x`
Size of type	`sizeof(t)`
Number of arguments	`sizeof...(p)`
Number of type arguments	`sizeof...(P)`
Alignment of object (C++11)	`alignof(x)`
Alignment of type (C++11)	`alignof(t)`

Note that the `sizeof` operator when used on an expression is the only one that is applicable without parentheses. `alignof` was introduced in C++11; all others have existed since C++98 (at least).

1.3.9 Error Handling

The `throw` operator is used to indicate an exception in the execution (e.g., insufficient memory); see Section 1.6.2.

1.3.10 Overloading

A very powerful aspect of C++ is that the programmer can define operators for new types. This will be explained in Section 2.7. Operators of built-in types cannot be changed. However, we can define how built-in types interact with new types; i.e., we can overload mixed operations like `double` times `matrix`. Most operators can be overloaded. Exceptions are:

`::`	Scope resolution;
`.`	Member selection;
`.*`	Member selection through pointer;
`?:`	Conditional;
`sizeof`	Size of a type or object;
`sizeof...`	Number of arguments;
`alignof`	Memory alignment of a type or object; and
`typeid`	Type identifier.

The operator overloading in C++ gives us a lot of freedom and we have to use this freedom wisely. We come back to this topic in the next chapter when we actually overload operators (in Section 2.7).

1.3.11 Operator Precedence

Table 1–8 gives a concise overview of the operator priorities. For compactness, we combined notations for types and expressions (e.g., `typeid`) and fused the different notations for `new` and `delete`. The symbol `@=` represents all computational assignments like `+=`, `-=`, and so on. A more detailed summary of operators with semantics is given in Appendix C, Table C–1.

Table 1–8: Operator Precedence

Operator Precedence			
class::member	nspace::member	::name	::qualified-name
object.member	pointer->member	expr[expr]	expr(expr_list)
type(expr_list)	lvalue++	lvalue--	typeid(type/expr)
*_cast<type>(expr)			
sizeof expr	sizeof(type)	sizeof...(pack)	alignof(type/expr)
++lvalue	--lvalue	∼expr	!expr
-expr	+expr	&lvalue	*expr
new ... type...	delete $[]_{opt}$ pointer	(type) expr	co_await expr
object.*member_ptr	pointer->*member_ptr		
expr * expr	expr / expr	expr % expr	
expr + expr	expr - expr		
expr ≪ expr	expr ≫ expr		
expr <=> expr			
expr < expr	expr <= expr	expr > expr	expr >= expr
expr == expr	expr != expr		
expr & expr			
expr ^ expr			
expr | expr			
expr && expr			
expr || expr			
expr ? expr: expr			
lvalue = expr	lvalue @= expr	throw expr	co_yield expr
expr , expr			

1.3.12 Avoid Side Effects!

"Insanity: doing the same thing over and over again and expecting different results."

—Unknown[7]

In applications with side effects it is not insane to expect a different result for the same input. On the contrary, it is very difficult to predict the behavior of a program whose components

7. Misattributed to Albert Einstein, Benjamin Franklin, and Mark Twain. It is cited in *Sudden Death* by Rita Mae Brown but the original source seems to be unknown. Maybe the quote itself is beset with some insanity.

interfere massively. Moreover, it is probably better to have a deterministic program with the wrong result than one that occasionally yields the right result since the latter is usually much harder to fix.

An example where the side effects are incorporated correctly is the string copy function `strcpy` from the C standard library. The function takes pointers to the first `char` of the source and the target and copies the subsequent letters until it finds a zero. This can be implemented with one single loop that even has an empty body and performs the copy and the increments as side effects of the continuation test:

```
while (*tgt++= *src++);
```

Looks scary? Well, it somehow is. Nonetheless, this is absolutely legal C++ code, although some compilers might grumble in pedantic mode. It is a good mental exercise to spend some time thinking about operator priorities, types of subexpressions, and evaluation order.

Let us think about something simpler: we assign the value `i` to the `i`-th entry of an array and increment the value `i` for the next iteration:

```
v[i]= i++;    // Undefined behavior before C++17
```

Looks like no problem. Before C++17 it was: the behavior of this expression was undefined. Why? The post-increment of `i` guarantees that we assign the old value of `i` and increment `i` afterward. However, this increment can still be performed before the expression `v[i]` is evaluated so that we possibly assign `i` to `v[i+1]`. Well, this was fixed in C++17 by requiring that the entire expression to the right of the assignment must be finished before the left-hand side is evaluated. This doesn't mean that all undefined behavior disappeared in the meantime. The following—admittedly nasty—expression is still undefined:

```
i = ++i + i++;
```

The last examples should give you an impression that side effects are not always evident at first glance. Some quite tricky stuff might work but much simpler things might not. Even worse, something might work for a while until somebody compiles it on a different compiler or the new release of your compiler changes some implementation details.

The first snippet is an example of excellent programming skills and evidence that the operator precedence makes sense—no parentheses were needed. Nonetheless, such programming style is not appropriate for modern C++. The eagerness to shorten code as much as possible dates back to the times of early C when typing was more demanding, with typewriters that were more mechanical than electrical, and card punchers, all without a monitor. With today's technology, it should not be an issue to type some extra letters.

Another unfavorable aspect of the terse copy implementation is the mingling of different concerns: testing, modification, and traversal. An important concept in software design is *Separation of Concerns*. It contributes to increasing flexibility and decreasing complexity. In this case, we want to decrease the complexity of the mental processes needed to understand the implementation. Applying the principle to the infamous copy one-liner could yield:

```
for (; *src; tgt++, src++)
    *tgt= *src;
*tgt= *src; // copy the final 0
```

Now, we can clearly distinguish the three concerns:

1. Testing: `*src`
2. Modification: `*tgt= *src;`
3. Traversal: `tgt++, src++`

It is also more apparent that the incrementing is performed on the pointers and the testing and assignment on their referred content. The implementation is not as compact as before, but it is much easier to check the correctness. It is also advisable to make the nonzero test more obvious (`*src != 0`).

There is a class of programming languages that are called *Functional Languages*. Values in these languages cannot be changed once they are set. C++ is obviously not that way. But we do ourselves a big favor when we program as much as is reasonable in a functional style. For instance, when we write an assignment, the only thing that should change is the variable to the left of the assignment symbol. To this end, we have to replace mutating with a constant expression: for instance, `++i` with `i+1`. A right-hand side expression without side effects helps us understand the program behavior and makes it easier for the compiler to optimize the code. As a rule of thumb: more comprehensible programs have a better potential for optimization. Speaking of which, `const` declarations not only protect us against accidental modifications, they are also an easy way to enable more optimizations.

1.4 Expressions and Statements

C++ distinguishes between expressions and statements. Very casually, we could say that every expression becomes a statement if a semicolon is appended. However, we would like to discuss this topic a bit more.

1.4.1 Expressions

Let us build this recursively from the bottom up. Any variable name (`x`, `y`, `z`, . . .), constant, or literal is an expression. One or more expressions combined by an operator constitute an expression, e.g., `x + y` or `x * y + z`. In several languages, such as Pascal, the assignment is a statement. In C++, it is an expression, e.g., `x= y + z`. As a consequence, it can be used within another assignment: `x2= x= y + z`. Assignments are evaluated from right to left. Input and output operations such as

```
std::cout ≪ "x is " ≪ x ≪ "\n"
```

are also expressions.

A function call with expressions as arguments is an expression, e.g., `abs(x)` or `abs(x * y + z)`. Therefore, function calls can be nested: `pow(abs(x), y)`. Note that nesting would not be possible if function calls were statements.

Since an assignment is an expression, it can be used as an argument of a function: `abs(x= y)`. Or I/O operations such as those above, e.g.:

```
print(std::cout ≪ "x is " ≪ x ≪ "\n", "I am such a nerd!");
```

Needless to say, this is not particularly readable and it would cause more confusion than doing something useful. An expression surrounded by parentheses is an expression as well, e.g., `(x + y)`. As this grouping by parentheses precedes all operators, we can change the order of evaluation to suit our needs: `x * (y + z)` computes the addition first.

1.4.2 Statements

Any of the expressions above followed by a semicolon is a statement, e.g.:

```
x= y + z;
y= f(x + z) * 3.5;
```

A statement like

```
y + z;
```

is allowed despite having no effect (usually). During program execution, the sum of `y` and `z` is computed and then thrown away. Recent compilers optimize out such useless computations. However, it is not guaranteed that this statement can always be omitted. If `y` or `z` is an object of a user type, then the addition is also user defined and might change `y` or `z` or something else. This is obviously bad programming style (hidden side effect) but legitimate in C++.

A single semicolon is an empty statement, and we can thus put as many semicolons after an expression as we want. Some statements do not end with a semicolon, e.g., function definitions. If a semicolon is appended to such a statement, it is not an error but just an extra empty statement. Nonetheless, some compilers print a warning in pedantic mode. Any sequence of statements surrounded by curly braces is a statement—called a *Compound Statement*.

The variable and constant declarations we have seen before are also statements. As the initial value of a variable or constant, we can use any expression (except another assignment or comma operator). Other statements—to be discussed later—are function and class definitions, as well as control statements that we will introduce in the next sections.

With the exception of the conditional operator, program flow is controlled by statements. Here we will distinguish between branches and loops.

1.4.3 Branching

In this section, we will present the different features that allow us to select a branch in the program execution.

1.4.3.1 `if`-Statement

This is the simplest form of control and its meaning is intuitively clear, for instance in:

```
if (weight > 100.0)
    cout ≪ "This is quite heavy.\n";
else
    cout ≪ "I can carry this.\n";
```

Often, the `else` branch is not needed and can be omitted. Say we have some value in variable `x` and compute something on its magnitude:

```
if (x < 0.0)
    x= -x;
// Now we know that x >= 0.0 (post-condition)
```

The branches of the `if`-statement are scopes, rendering the following statements erroneous:

```
if (x < 0.0)
    double absx= -x;
else
    double absx= x;
cout ≪ "|x| is " ≪ absx ≪ "\n"; // Error: absx out of scope
```

Above, we introduced two new variables, both named `absx`. They are not in conflict because they reside in different scopes. Neither of them exists after the `if`-statement, and accessing `absx` in the last line is an error. In fact, variables declared in a branch can only be used within this branch.

Each branch of `if` consists of one single statement. To perform multiple operations, we can use braces, as in the following example realizing Cardano's method:

```
double D= q*q/4.0 + p*p*p/27.0;
if (D > 0.0) {
    double z1= ...;
    complex<double> z2= ..., z3= ...;
    ...
} else if (D == 0.0) {
    double z1= ..., z2= ..., z3= ...;
    ...
} else {                    // D < 0.0
    complex<double> z1= ..., z2= ..., z3= ...;
    ...
}
```

In the beginning, it is helpful to always write the braces. Many style guides also enforce curly braces on single statements whereas the author prefers them without braces. Irrespective of this, it is highly advisable to indent the branches for better readability.

`if`-statements can be nested whereas each `else` is associated with the last open `if`. If you are interested in examples, have a look at Section A.2.2. Finally, we give you the following:

Advice
Although spaces do not affect compilation in C++, the indentation should reflect the structure of the program. Editors that understand C++ (like Visual Studio's IDE or emacs in C++ mode) and indent automatically are a great help with structured programming. Whenever a line within a language-aware tool is not indented as expected, something is most likely not nested as intended.

⇒ `c++17/if_init.cpp`

C++17 The `if`-statement was extended in C++17 with the possibility to initialize a variable whose scope is limited to the `if`-statement. This helps control the lifetime of variables; for instance, the result of an insertion into a `map` (see Section 4.1.3.5) is a kind of reference to the new entry and a `bool` if the insertion was successful:

```
map<string, double> constants= {{"e", 2.7}, {"pi", 3.14}};
if (auto res= constants.insert({"h", 6.6e-34}); res.second)
    cout ≪ "inserted " ≪ res.first->first ≪ " mapping to "
         ≪ res.first->second ≪ endl;
else
    cout ≪ "entry for " ≪ res.first->first ≪ " already exists.\n";
```

We could have declared `res` before the `if`-statement and it would then exist until the end of the surrounding block—unless we put extra braces around the variable declaration and the `if`-statement.

1.4.3.2 Conditional Expression

Although this section describes statements, we like to talk about the conditional expression here because of its proximity to the `if`-statement. The result of

```
condition ? result_for_true : result_for_false
```

is the second subexpression (i.e., `result_for_true`) when `condition` evaluates to `true` and `result_for_false` otherwise. For instance,

```
min= x <= y ? x : y;
```

corresponds to the following `if`-statement:

```
if (x <= y)
    min= x;
else
    min= y;
```

For a beginner, the second version might be more readable while experienced programmers often prefer the first form for its brevity.

`?:` is an expression and can therefore be used to initialize variables:

```
int x= f(a),
    y= x < 0 ? -x : 2 * x;
```

Calling functions with several selected arguments is easy with the operator:

```
f(a, (x < 0 ? b : c), (y < 0 ? d : e));
```

but quite clumsy with an `if`-statement. If you do not believe us, try it.

In most cases it is not important whether an `if` or a conditional expression is used. So use what feels most convenient to you.

Anecdote: An example where the choice between `if` and `?:` makes a difference is the `replace_copy` operation in the Standard Template Library (STL), §4.1. It used to be implemented with the conditional operator whereas `if` would be slightly more general. This “bug” remained undiscovered for approximately 10 years and was only detected by an automatic analysis in Jeremy Siek's Ph.D. thesis [57].

1.4.3.3 switch Statement

A `switch` is like a special kind of `if`. It provides a concise notation when different computations for different cases of an integral value are performed:

```
switch(op_code) {
  case 0: z= x + y; break;
  case 1: z= x - y; cout ≪ "compute diff\n"; break;
  case 2:
  case 3: z= x * y; break;
  default: z= x / y;
}
```

A somewhat surprising behavior is that the code of the following cases is also performed unless we terminate it with `break`. Thus, the same operations are performed in our example for cases 2 and 3. A compiler warning for (nonempty) cases without `break` is generated with `-Wimplicit-fallthrough` in `g++` and `clang++`.

C++17 To avoid such warnings and to communicate to co-developers that the fall-through is intended, C++17 introduces the attribute `[[fallthrough]]`:

```
switch(op_code) {
  case 0: z= x + y; break;
  case 1: z= x - y; cout ≪ "compute diff\n"; break;
  case 2: x= y; [[fallthrough]];
  case 3: z= x * y; break;
  default: z= x / y;
}
```

C++17 Also added in C++17 is the ability to initialize a variable in the `switch`-statement in the same way as in `if`.

An advanced use of `switch` is found in Appendix A.2.3.

1.4.4 Loops

1.4.4.1 while- and do-while-Loops

As the name suggests, a `while`-loop is repeated as long as the given condition holds. Let us implement as an example the Collatz series that is defined by

Algorithm 1–1: Collatz series

```
  Input: $x_0$
1 while $x_i \neq 1$ do
2     $x_i = \begin{cases} 3\,x_{i-1} + 1 & \text{if } x_{i-1} \text{ is odd} \\ x_{i-1}/2 & \text{if } x_{i-1} \text{ is even} \end{cases}$
```

If we do not worry about overflow, this is easily implemented with a `while`-loop:

```
int x= 19;
while (x != 1) {
    cout ≪ x ≪ '\n';
    if (x % 2 == 1)      // odd
        x= 3 * x + 1;
    else                 // even
        x= x / 2;
}
```

Like the `if`-statement, the loop can be written without braces in case of a single statement.

C++ also offers a `do`-`while`-loop. There the condition for continuation is tested at the end:

```
double eps= 0.001;
do {
    cout ≪ "eps= " ≪ eps ≪ '\n';
    eps/= 2.0;
} while (eps > 0.0001);
```

The loop is performed at least once regardless of the condition.

1.4.4.2 `for`-Loop

The most common loop in C++ is the `for`-loop. As a simple example, we add two vectors[8] and print the result afterward:

```
double v[3], w[]= {2., 4., 6.}, x[]= {6., 5., 4};
for (int i= 0; i < 3; ++i)
    v[i]= w[i] + x[i];

for (int i= 0; i < 3; ++i)
    cout ≪ "v[" ≪ i ≪ "]= " ≪ v[i] ≪ '\n';
```

The loop head consists of three components:

1. The initialization
2. A *Continuation* criterion
3. A step operation

The example above is a typical `for`-loop. In the initialization, we usually declare a new variable and initialize it with 0—this is the start index of most indexed data structures. The condition typically tests whether the loop index is smaller than a certain size while the last operation usually increments the loop index. In the example, we pre-incremented the loop variable `i`. For intrinsic types like `int`, it does not matter whether we write `++i` or `i++`. However, it does for user types where the post-increment causes an unnecessary copy; cf. §3.3.2.5. To be consistent in this book, we always use a pre-increment for loop indices.

It is a very popular beginners' mistake to write conditions like `i <= size(..)`. Since indices are zero based in C++, the index `i == size(..)` is already out of range. People with

8. Later we will introduce real vector classes. For the moment we take simple arrays.

experience in Fortran or MATLAB need some time to get used to zero-based indexing. One-based indexing seems more natural to many and is also used in mathematical literature. However, calculations on indices and addresses are almost always simpler with zero-based indexing.

As another example, we like to compute the Taylor series of the exponential function:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

up to the tenth term:

```
double x= 2.0, xn= 1.0, exp_x= 1.0;
unsigned long fac= 1;
for (unsigned long n= 1; n <= 10; ++n) {
    xn*= x;
    fac*= n;
    exp_x+= xn / fac;
    cout ≪ "e^x is " ≪ exp_x ≪ '\n';
}
```

Here it was simpler to compute term 0 separately and start the loop with term 1. We also used less than or equal to ensure that the term $x^{10}/10!$ is considered.

The `for`-loop in C++ is very flexible. The initialization part can be any expression, a variable declaration, or empty. It is possible to introduce multiple new variables of the same type. This can be used to avoid repeating the same operation in the condition, e.g.:

```
for (int i= begin(xyz), e= end(xyz); i < e; ++i) ...
```

Variables declared in the initialization are only visible within the loop and hide variables of the same names from outside the loop.

The condition can be any expression that is convertible to a `bool`. An empty condition is always `true` and the loop is repeated infinitely. It can still be terminated inside the body, as we will discuss in Section 1.4.4.4. We already mentioned that a loop index is typically incremented in the third subexpression of `for`. In principle, we can modify it within the loop body as well. However, programs are much clearer if it is done in the loop head. On the other hand, there is no limitation that only one variable is increased by 1. We can modify as many variables as we want using the comma operator (§1.3.5) and by any modification desired, such as

```
for (int i= 0, j= 0, p= 1; ...; ++i, j+= 4, p*= 2) ...
```

This is of course more complex than having just one loop index, but it is still more readable than declaring/modifying indices before the loop or inside the loop body.

C++11

1.4.4.3 Range-Based `for`-Loop

A very compact notation is provided by the feature called *Range-Based* `for`*-Loop*. We will tell you more about its background once we come to the iterator concept (§4.1.2).

For now, we will consider it as a concise form to iterate over all entries of an array or other containers:

```
int primes[]= {2, 3, 5, 7, 11, 13, 17, 19};
for (int i : primes)
    std::cout ≪ i ≪ " ";
```

This will print the primes from the array separated by spaces. In C++20 we can initialize `prime` in the range-based loop: C++20

```
for (int primes[]= {2, 3, 5, 7, 11, 13, 17, 19}; int i : primes)
    std::cout ≪ i ≪ " ";
```

1.4.4.4 Loop Control

There are two statements to deviate from the regular loop evaluation:

1. `break`
2. `continue`

A `break` terminates the loop entirely, and `continue` ends only the current iteration and continues the loop with the next iteration, for instance:

```
for (...; ...; ...) {
    ...
    if (dx == 0.0)
        continue;
    x+= dx;
    ...
    if (r < eps)
        break;
    ...
}
```

In the example above, we assumed that the remainder of the iteration is not needed when `dx == 0.0`. In some iterative computations, it might be clear in the middle of an iteration (here when `r < eps`) that all work is already done.

1.4.5 `goto`

All branches and loops are internally realized by jumps. C++ provides explicit jumps called `goto`. However:

Advice

Do not use `goto`! Never! Ever!

The applicability of `goto` is more restrictive in C++ than in C (e.g., we cannot jump over initializations); it still has the power to ruin the structure of our program.

Writing software without `goto` is called *Structured Programming*. However, the term is rarely used nowadays since this is taken for granted in high-quality software.

1.5 Functions

Functions are important building blocks of C++ programs. The first example we have seen is the `main` function in the hello-world program. We will say a little more about `main` in Section 1.5.5.

1.5.1 Arguments

C++ distinguishes two forms of passing arguments: by value and by reference.

1.5.1.1 Call by Value

When we pass an argument to a function, it creates a copy by default. For instance, the following function increments `x` but not visibly to the outside world:

```
void increment(int x)
{
    x++;
}

int main()
{
    int i= 4;
    increment(i);       // Does not increment i
    cout ≪ "i is " ≪ i ≪ '\n';
}
```

The output is 4. The operation `x++` only increments a local copy of `i` within the `increment` function but not `i` itself. This kind of argument transfer is referred to as *Call by Value* or *Pass by Value*.

1.5.1.2 Call by Reference

To modify function parameters, we have to *Pass* the argument *by Reference*:

```
void increment(int& x)
{
    x++;
}
```

Now, the variable itself is incremented and the output will be 5 as expected. We will discuss references in more detail in §1.8.4.

Temporary variables—like the result of an operation—cannot be passed by reference:

```
increment(i + 9); // Error: temporary not referable
```

In order to pass the expression to this function, we must store it to a variable up front and pass that variable. Obviously, modifying functions on temporary variables makes no sense anyway since we never see the impact of the modification.

Larger data structures like vectors and matrices are almost always passed by reference to avoid expensive copy operations:

```
double two_norm(vector& v) { ... }
```

An operation like a norm should not change its argument. But passing the vector by reference bears the risk of accidentally overwriting it. To make sure our vector is not changed (and not copied either), we pass it as a constant reference:

```
double two_norm(const vector& v) { ... }
```

If we tried to change `v` in this function the compiler would emit an error.

Both call-by-value and constant references ascertain that the argument is not altered but by different means:

- Arguments that are passed by value can be changed in the function since the function works with a copy.[9]
- With `const` references we work directly on the passed argument, but all operations that might change the argument are forbidden. In particular, `const`-reference arguments cannot appear on the left-hand side (LHS) of an assignment or be passed as non-`const` references to other functions; in fact, the LHS argument of an assignment is also a non-`const` reference.

In contrast to mutable[10] references, constant ones allow for passing temporaries:

```
alpha= two_norm(v + w);
```

This is admittedly not entirely consequential on the language design side, but it makes the life of programmers much easier.

1.5.1.3 Default Arguments

If an argument usually has the same value, we can declare it with a default value. Say we implement a function that computes the n-th root and mostly the square root, then we can write:

```
double root(double x, int degree= 2) { ... }
```

This function can be called with one or two arguments:

```
x= root(3.5, 3);
y= root(7.0);     // like root(7.0, 2)
```

We can declare multiple defaults but only at the end of the parameter list. In other words, after an argument with a default value we cannot have one without.

9. Assuming the argument is properly copied. User types with broken copy implementations can undermine the integrity of the passed-in data.
10. Note that we use the word *mutable* for linguistic reasons as a synonym for non-`const` in this book. In C++, we also have the keyword `mutable` (§2.6.3), which we do not use very often.

Default values are also helpful when extra parameters are added. Let us assume that we have a function that draws circles:

```
draw_circle(int x, int y, float radius);
```

These circles are all black. Later, we add a color:

```
draw_circle(int x, int y, float radius, color c= black);
```

Thanks to the default argument, we do not need to refactor our application since the calls of `draw_circle` with three arguments still work.

1.5.2 Returning Results

In the earlier examples, we only returned `double` or `int`. These are well-behaved `return` types. Now we will look at the extremes: large or no data.

1.5.2.1 Returning Large Data Structures

Functions that compute new values of large data structures are more difficult. For the details, we will put you off until later and only mention the options here. The good news is that compilers are smart enough to elide the copy of the return value in many cases; see Section 2.3.5.3. In addition, the move semantics (Section 2.3.5) where data of temporaries is stolen avoids copies when the aforementioned elision does not apply. Advanced libraries avoid returning large data structures altogether with a technique called expression templates and delay the computation until it is known where to store the result (Section 5.3.2). In any case, we must not return references to local function variables (Section 1.8.6).

1.5.2.2 Returning Nothing

Syntactically, each function must return something even if there is nothing to return. This dilemma is solved by the void type named `void`. For instance, a function that just prints `x` has no result:

```
void print_x(int x)
{
    std::cout ≪ "The value x is " ≪ x ≪ '\n';
}
```

`void` is not a real type but more of a placeholder that enables us to omit returning a value. We cannot define `void` objects:

```
void nothing;     // Error: no void objects
```

A `void` function can be terminated earlier:

```
void heavy_compute(const vector& x, double eps, vector& y)
{
    for (...) {
        ...
        if (two_norm(y) < eps)
            return;
    }
}
```

with a no-argument `return`. Returning something in a `void` function would be an error. The only thing that can appear in its `return` statement is the call of another `void` function (as a shortcut of the call plus an empty `return`).

1.5.3 Inlining

Calling a function is relatively expensive: registers must be stored, arguments copied on the stack, and so on. To avoid this overhead, the compiler can `inline` function calls. In this case, the function call is substituted with the operations contained in the function. The programmer can ask the compiler to do so with the appropriate keyword:

```
inline double square(double x) { return x*x; }
```

However, the compiler is not obliged to `inline`. Conversely, it can `inline` functions without the keyword if this seems promising for performance. The `inline` declaration still has its use: for including a function in multiple compile units, which we will discuss in Section 7.2.3.2.

1.5.4 Overloading

In C++, functions can share the same name as long as their parameter declarations are sufficiently different. This is called *Function Overloading*. Let us first look at an example:

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

int divide(int a, int b) {
    return a / b ;
}

float divide(float a, float b) {
    return std::floor( a / b ) ;
}

int main() {
    int   x= 5, y= 2;
    float n= 5.0, m= 2.0;
    std::cout << divide(x, y) << std::endl;
    std::cout << divide(n, m) << std::endl;
    std::cout << divide(x, m) << std::endl; // Error: ambiguous
}
```

Here we defined the function `divide` twice: with `int` and `float` parameters. When we call `divide`, the compiler performs an *Overload Resolution*:

1. Is there an overload that matches the argument type(s) exactly? Take it; otherwise:
2. Are there overloads that match after conversion? How many?
   - 0: Error: No matching function found.
   - 1: Take it.
   - >1: Error: ambiguous call.

How does this apply to our example? The calls `divide(x, y)` and `divide(n, m)` are exact matches. For `divide(x, m)`, no overload matches exactly and both by *Implicit Conversion* so that it's ambiguous.

The term *implicit conversion* requires some explanation. We have already seen that the language's numeric types can be converted one to another. These are implicit conversions as demonstrated in the example. When we later define our own types, we can implement a conversion from another type to it or conversely from our new type to an existing one.

⇒ `c++11/overload_testing.cpp`

More formally phrased, function overloads must differ in their *Signature*. In C++, the signature consists of

- The function name;
- The number of arguments, called *Arity*; and
- The types of the arguments (in their respective order).

In contrast, overloads varying only in the `return` type or the argument names have the same signature and are considered as (forbidden) redefinitions:

```
void f(int x) {}
void f(int y) {} // Redefinition: only argument name different
long f(int x) {} // Redefinition: only return type different
```

That functions with different names or arity are distinct goes without saying. The presence of a reference symbol turns the argument type into another argument type (thus, `f(int)` and `f(int&)` can coexist). The following three overloads have different signatures:

```
void f(int x) {}              // #1
void f(int& x) {}             // #2
void f(const int& x) {}       // #3
```

This code snippet compiles. Problems will arise, however, when we call `f`:

```
int       i= 3;
const int ci= 4;

f(i);
f(ci);
f(3);
```

All three function calls are ambiguous: for the first call `#1` and `#2` are equal matches, and for the other calls `#1` and `#3`. Mixing overloads of reference and value arguments almost always fails. Thus, when one overload has a reference qualified argument, the corresponding argument of the other overloads should be reference qualified as well. We can achieve this in our toy example by omitting the value-argument overload. Then `f(3)` and `f(ci)` will resolve to the overload with the constant reference and `f(i)` to that with the mutable one.

1.5.5 main Function

The main function is not fundamentally different from any other function. There are two signatures allowed in the standard:

```
int main()
```

or:

```
int main(int argc, char* argv[])
```

The latter is equivalent to:

```
int main(int argc, char** argv)
```

The parameter argv contains the list of arguments and argc its length. The first argument (argv[0]) is on most systems the name of the called executable (which may be different from the source code name). To play with the arguments, we can write a short program called argc_argv_test:

```
int main (int argc, char* argv[])
{
    for (int i= 0; i < argc; ++i)
        cout ≪ argv[i] ≪ '\n';
    return 0;
}
```

Calling this program with the following options:

```
argc_argv_test first second third fourth
```

yields:

```
argc_argv_test
first
second
third
fourth
```

As you can see, each space in the command splits the arguments. The main function returns an integer as exit code that states whether the program finished correctly or not. Returning 0 (or the macro EXIT_SUCCESS from <cstdlib>) represents success and every other value a failure. It is standards compliant to omit the return statement in the main function. In this case, return 0; is automatically inserted. Some extra details are found in Section A.2.4.

1.6 Error Handling

"An error doesn't become a mistake until you refuse to correct it."

—Orlando Aloysius Battista

The two principal ways to deal with unexpected behavior in C++ are assertions and exceptions. The former is intended for detecting programming errors and the latter for exceptional situations that prevent proper continuation of the program. To be honest, the distinction is not always obvious.

1.6.1 Assertions

The macro `assert` from header `<cassert>` is inherited from C but still useful. It evaluates an expression, and when the result is `false` the program is terminated immediately. It should be used to detect programming errors. Say we implement a cool algorithm computing a square root of a non-negative real number. Then we know from mathematics that the result is non-negative. Otherwise something is wrong in our calculation:

```
#include <cassert>

double square_root(double x)
{
    check_somehow(x >= 0);
    ...
    assert(result >= 0.0);
    return result;
}
```

How to implement the initial check is left open for the moment. When our result is negative, the program execution will print an error like this:

```
assert_test: assert_test.cpp:10: double square_root(double):
Assertion 'result >= 0.0' failed.
```

The assertion requires that our result must be greater than or equal to zero; otherwise our implementation contains a bug and we must fix it before we use this function for serious applications.

After we fixed the bug we might be tempted to remove the assertion(s). We should not do so. Maybe one day we will change the implementation; then we still have all our sanity tests working. Actually, assertions on post-conditions are somehow like mini-unit tests.

A great advantage of `assert` is that we can let it disappear entirely by a simple macro declaration. Before including `<cassert>` we can define `NDEBUG`:

```
#define NDEBUG
#include <cassert>
```

and all assertions are disabled; i.e., they do not cause any operation in the executable. Instead of changing our program sources each time we switch between debug and release mode, it is better and cleaner to declare `NDEBUG` in the compiler flags (usually `-D` on Linux and `/D` on Windows):

```
g++ my_app.cpp -o my_app -O3 -DNDEBUG
```

Software with assertions in critical kernels can be slowed down by a factor of two or more when the assertions are not disabled in the release mode. Good build systems like `CMake` activate `-DNDEBUG` automatically in the release mode's compile flags.

Since assertions can be disabled so easily, we should follow this advice:

Defensive Programming
Test as many properties as you can.

Even if you are sure that a property obviously holds for your implementation, write an assertion. Sometimes the system does not behave precisely as we assumed, or the compiler might be buggy (extremely rare but not impossible), or we did something slightly different from what we intended originally. No matter how much we reason and how carefully we implement, sooner or later one assertion may be raised. If there are so many properties to check that the actual functionality is no longer clearly visible in the code, the tests can be outsourced to another function.

Responsible programmers implement large sets of tests. Nonetheless, this is no guarantee that the program works under all circumstances. An application can run for years like a charm and one day it crashes. In this situation, we can run the application in debug mode with all the assertions enabled, and in most cases they will be a great help to find the reason for the crash. However, this requires that the crashing situation is reproducible and that the program in slower debug mode reaches the critical section in reasonable time.

1.6.2 Exceptions

In the preceding section, we looked at how assertions help us detect programming errors. However, there are many critical situations that we cannot prevent even with the smartest programming, like files that we need to read but which are deleted. Or our program needs more memory than is available on the actual machine. Other problems are preventable in theory but the practical effort is disproportionally high, e.g., to check whether a matrix is regular is feasible but might be as much or even more work than the actual task. In such cases, it is usually more efficient trying to accomplish the task and checking for *Exceptions* along the way.

1.6.2.1 Motivation

Before illustrating the old-style error handling, we introduce our anti-hero Herbert[11] who is an ingenious mathematician and considers programming a necessary evil for demonstrating how magnificently his algorithms work. He is immune to the newfangled nonsense of modern programming.

His favorite approach to deal with computational problems is to return an error code (like the `main` function does). Say we want to read a matrix from a file and check whether the file is really there. If not, we return an error code of 1:

```
int read_matrix_file(const char* fname, matrix& A)
{
    fstream f(fname);
    if (!f.is_open())
        return 1;
        ...
    return 0;
}
```

So, he checked for everything that can go wrong and informed the caller with the appropriate error code. This is fine when the caller evaluated the error and reacted appropriately. But

11. To all readers named Herbert: Please accept our honest apology for having picked your name.

what happens when the caller simply ignores his return code? Nothing! The program keeps going and might crash later on absurd data, or even worse might produce nonsensical results that careless people might use to build cars or planes. Of course, car and plane builders are not that careless, but in more realistic software even careful people cannot have an eye on each tiny detail.

Nonetheless, bringing this reasoning across to programming dinosaurs like Herbert might not convince them: "Not only are you dumb enough to pass in a nonexisting file to my perfectly implemented function, then you do not even check the return code. You do everything wrong, not me."

C++17

We get a little bit more security with C++17. It introduces the attribute `[[nodiscard]]` to state that the return value should not be discarded:

```
[[nodiscard]] int read_matrix_file(const char* fname, matrix& A)
```

As a consequence, each call that ignores the return value will cause a warning, and with an additional compiler flag we can turn each warning into an error. Conversely, we can also suppress this warning with another compiler flag. Thus, the attribute doesn't guarantee us that the return code is used. Furthermore, merely storing the return value into a variable already counts as usage, regardless of whether we touch this variable ever again.

Another disadvantage of the error codes is that we cannot return our computational results and have to pass them as reference arguments. This prevents us from building expressions with the result. The other way around is to return the result and pass the error code as a (referred) function argument, which is not much less cumbersome. So much for messing around with error codes. Let us now see how exceptions work.

1.6.2.2 Throwing

The better approach to deal with problems is to `throw` an exception:

```
matrix read_matrix_file(const std::string& fname)
{
    fstream f(fname);
    if (!f.is_open())
        throw "Cannot open file.";
    ...
}
```

C++ allows us to `throw` everything as an exception: strings, numbers, user types, et cetera. However, for dealing with the exceptions properly it is better to define exception types or to use those from the standard library:

```
struct cannot_open_file {};

matrix read_matrix_file(const std::string& fname)
{
    fstream f(fname);
    if (!f.is_open())
        throw cannot_open_file{};
    matrix A;
    // populate A with data (possibly throw exception)
    return A;
}
```

Here, we introduced our own exception type. In Chapter 2, we will explain in detail how classes can be defined. In the preceding example, we defined an empty class that only requires opening and closing braces followed by a semicolon. Larger projects usually establish an entire hierarchy of exception types that are usually derived (Chapter 6) from `std::exception`.

1.6.2.3 Catching

To react to an exception, we have to `catch` it. This is done in a `try`-`catch`-block. In our example we threw an exception for a file we were unable to open, which we can catch now:

```
try {
    A= read_matrix_file("does_not_exist.dat");
} catch (const cannot_open_file& e) {
    // Here we can deal with it, hopefully.
}
```

We can write multiple catch clauses in the block to deal with different exception types in one location. Discussing this in detail makes much more sense after introducing classes and inheritance. Therefore we postpone it to Section 6.1.5.

1.6.2.4 Handling Exceptions

The easiest handling is delegating it to the caller. This is achieved by doing nothing (i.e., no `try`-`catch`-block).

We can also catch the exception, provide an informative error message, and terminate the program:

```
try {
    A= read_matrix_file("does_not_exist.dat");
} catch (const cannot_open_file& e) {
    cerr ≪ "Hey guys, your file does not exist! I'm out.\n";
    exit(EXIT_FAILURE);
}
```

Once the exception is caught, the problem is considered to be solved and the execution continues after the `catch`-block(s). To terminate the execution, we used `exit` from the header `<cstdlib>`. The function `exit` ends the execution even when we are not in the `main` function. It should only be used when further execution is too dangerous and there is no hope that the calling functions have any cure for the exception either.

Alternatively, we can continue after the complaint or a partial rescue action by rethrowing the exception, which might be dealt with later:

```
try {
    A= read_matrix_file("does_not_exist.dat");
} catch (const cannot_open_file& e) {
    cerr ≪ "O my gosh, the file is not there! Please caller help me.\n";
    throw;
}
```

In our case, we are already in the `main` function and there is no other function on the call stack to catch our exception. Ignoring an exception is easily implemented by an empty block:

```
} catch (cannot_open_file&) {} // File is rubbish, don't care
```

So far, our exception handling did not really solve our problem of missing a file. If the filename is provided by a user, we can pester him/her until we get one that makes us happy:

```
bool keep_trying= true;
do {
    std::string fname;
    cout ≪ "Please enter the filename: ";
    cin ≫ fname;
    try {
        A= read_matrix_file(fname);
        ...
        keep_trying= false;
    } catch (const cannot_open_file& e) {
        cout ≪ "Could not open the file. Try another one!\n";
    }
} while (keep_trying);
```

When we reach the end of the `try`-block, we know that no exception was thrown and we can call it a day. Otherwise we land in one of the `catch`-blocks and `keep_trying` remains `true`.

1.6.2.5 Advantages of Exceptions

Error handling is necessary when our program runs into a problem that cannot be solved where it is detected (otherwise we'd do so, obviously). Thus, we must communicate this to the calling function with the hope that the detected problem can be solved or at least treated in a manner that is acceptable to the user. It is possible that the direct caller of the problem-detecting function is not able to handle the either, and the issue must be communicated further up the call stack over several functions, possibly up to the main function. By taking this into consideration, exceptions have the following advantages over error codes:

- Function interfaces are clearer.
- Returning results instead of error codes allows for nesting function calls.
- Untreated errors immediately abandon the application instead of silently continuing with corrupt data.
- Exceptions are automatically propagated up the call stack.
- The explicit communication of error codes obfuscates the program structure.

An example from the author's practice concerned an LU factorization. It cannot be computed for a singular matrix. There is nothing we can do about it. However, in the case that the factorization was part of an iterative computation, we were able to continue the iteration somehow without that factorization. Although this would be possible with traditional error handling as well, exceptions allow us to implement it much more readably and elegantly.

We can program the factorization for the regular case and when we detect the singularity, we throw an exception. Then it is up to the caller how to deal with the singularity in the respective context—if possible.

1.6.2.6 Who Throws?

C++11

C++03 allowed specifying which types of exceptions can be thrown from a function. Without going into details, these specifications turned out not to be very useful and were deprecated since C++11 and removed since C++17.

C++11 added a new qualification for specifying that no exceptions must be thrown out of the function, e.g.:

```
double square_root(double x) noexcept { ... }
```

The benefit of this qualification is that the calling code never needs to check for thrown exceptions after `square_root`. If an exception is thrown despite the qualification, the program is terminated.

In templated functions, it can depend on the argument type(s) whether an exception is thrown. To handle this properly, `noexcept` can depend on a compile-time condition; see Section 5.2.2.

Whether an assertion or an exception is preferable is not an easy question and we have no short answer to it. The question will probably not bother you now. We therefore postpone the discussion to Section A.2.5 and leave it to you when you read it.

1.6.3 Static Assertions

C++11

Program errors that can already be detected during compilation can raise a `static_assert`. In this case, an error message is emitted and the compilation stopped.

```
static_assert(sizeof(int) >= 4,
              "int is too small on this platform for 70000");
const int capacity= 70000;
```

In this example, we store the literal value `70000` to an `int`. Before we do this, we verify that the size of `int` is large enough on the platform this code snippet is compiled for to hold the value correctly. The complete power of `static_assert` is unleashed with meta-programming (Chapter 5) and we will show more examples there.

1.7 I/O

C++ uses a convenient abstraction called *streams* to perform I/O operations in sequential media such as screens or keyboards. A stream is an object where a program can either insert characters or extract them. The standard C++ library contains the header `<iostream>` where the standard input and output stream objects are declared.

1.7.1 Standard Output

By default, the standard output of a program is written to the screen, and we can access it with the C++ stream named `cout`. It is used with the insertion operator, which is denoted by

`<<` (like left shift). We have already seen that it may be used more than once within a single statement. This is especially useful when we want to print a combination of text, variables, and constants, e.g.:

```
cout << "The square root of " << x << " is " << sqrt(x) << endl;
```

with an output like

```
The square root of 5 is 2.23607
```

`endl` produces a newline character. An alternative representation of `endl` is the character `\n`. For the sake of efficiency, the output may be buffered. In this regard, `endl` and `\n` differ: the former flushes the buffer while the latter does not. Flushing can help us when we are debugging (without a debugger) to find out between which outputs the program crashes. In contrast, when a large amount of text is written to files, flushing after every line slows down I/O considerably.

Fortunately, the insertion operator has a relatively low priority so that arithmetic operations can be written directly:

```
std::cout << "11 * 19 = " << 11 * 19 << std::endl;
```

All comparisons and logical and bitwise operations must be grouped by surrounding parentheses. Likewise the conditional operator:

```
std::cout << (age > 65 ? "I'm a wise guy\n" : "I am still half-baked.\n");
```

When we forget the parentheses, the compiler will remind us (offering us an enigmatic message to decipher).

1.7.2 Standard Input

Handling the standard input in C++ is done by applying the overloaded operator of extraction `>>`. The standard input device is usually the keyboard as stream name `cin`:

```
int age;
std::cin >> age;
```

This command reads characters from the input device and interprets them as a value of the variable type (here `int`) it is stored to (here `age`). The input from the keyboard is processed once the RETURN key has been pressed. We can also use `cin` to request more than one data input from the user:

```
std::cin >> width >> length;
```

which is equivalent to

```
std::cin >> width;
std::cin >> length;
```

In both cases the user must provide two values: one for `width` and another for `length`. They can be separated by any valid blank separator: a space, a tab character, or a newline.

1.7.3 Input/Output with Files

C++ provides the following classes to perform input and output of characters from/to files:

`ofstream`	write to files
`ifstream`	read from files
`fstream`	both read and write from/to files

We can use file streams in the same fashion as `cin` and `cout`, with the only difference that we have to associate these streams with physical files. Here is an example:

```
#include <fstream>

int main ()
{
    std::ofstream square_file;
    square_file.open("squares.txt");
    for (int i= 0; i < 10; ++i)
        square_file ≪ i ≪ "^2 = " ≪ i*i ≪ '\n';
    square_file.close();
}
```

This code creates a file named `squares.txt` (or overwrites it if it already exists) and writes some lines to it—like we write to `cout`. C++ establishes a general stream concept that is satisfied by an output file and by `std::cout`. This means we can write everything to a file that we can write to `std::cout` and vice versa. When we define `operator≪` for a new type, we do this once for `ostream` (Section 2.7.3) and it will work with the console, with files, and with any other output stream.

Alternatively, we can pass the filename as an argument to the constructor of the stream to open the file implicitly. The file is also implicitly closed when `square_file` goes out of scope,[12] in this case at the end of the `main` function. The short version of the preceding program is:

```
#include <fstream>

int main ()
{
    std::ofstream square_file{"squares.txt"};
    for (int i= 0; i < 10; ++i)
        square_file ≪ i ≪ "^2 = " ≪ i*i ≪ '\n';
}
```

We prefer the short form (as usual). The explicit form is only necessary when the file is first declared and opened later for some reason. Likewise, the explicit `close` is only needed when the file should be closed before it goes out of scope.

1.7.4 Generic Stream Concept

Streams are not limited to screens, keyboards, and files; every class can be used as a stream when it is derived[13] from `istream`, `ostream`, or `iostream` and provides implementations for the

12. Thanks to the powerful technique named RAII, which we will discuss in Section 2.4.2.1.
13. How classes are derived is shown in Chapter 6. Let us here just take notice that being an output stream is technically realized by deriving it from `std::ostream`.

functions of those classes. For instance, Boost.Asio offers streams for TCP/IP and Boost. IOStream provides alternatives to the I/O above. The standard library contains a `stringstream` that can be used to create a string from any kind of printable type. `stringstream`'s method `str()` returns the stream's internal `string`.

We can write output functions that accept every kind of output stream by using a mutable reference to `ostream` as an argument:

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

void write_something(std::ostream& os)
{
    os << "Hi stream, did you know that 3 * 3 = " << 3 * 3 << '\n';
}

int main (int argc, char* argv[])
{
    std::ofstream myfile{"example.txt"};
    std::stringstream mysstream;

    write_something(std::cout);
    write_something(myfile);
    write_something(mysstream);

    std::cout << "mysstream is: " << mysstream.str(); // newline contained
}
```

Likewise, generic input can be implemented with `istream` and read/write I/O with `iostream`.

1.7.5 Formatting

⇒ `c++03/formatting.cpp`

I/O streams are formatted by so-called I/O manipulators which are found in the header file `<iomanip>`. By default, C++ only prints a few digits of floating-point numbers. Thus, we increase the precision:

```
double pi= M_PI;
cout << "pi is " << pi << '\n';
cout << "pi is " << setprecision(16) << pi << '\n';
```

and yield a more accurate number:

```
pi is 3.14159
pi is 3.141592653589793
```

C++20

In Section 4.3.1, we will show how the precision can be adjusted to the type's representable number of digits. Instead of the macro `M_PI` or a literal value, in C++20 we can use the `double` constant `std::number::pi` from `<numbers>`.

When we write a table, vector, or matrix, we need to align values for readability. Therefore, we next set the width of the output:

```
cout << "pi is " << setw(30) << pi << '\n';
```

This results in

```
pi is               3.141592653589793
```

setw changes only the next output while setprecision affects all following (numerical) outputs, like the other manipulators. The provided width is understood as a minimum, and if the printed value needs more space, our tables will get ugly.

We can further request that the values be left aligned, and the empty space be filled with a character of our choice, say, -:

```
cout << "pi is " << setfill('-') << left
     << setw(30) << pi << '\n';
```

yielding

```
pi is 3.141592653589793-------------
```

Another way of formatting is setting the flags directly. Furthermore, we force the “scientific” notation in the normalized exponential representation:

```
cout.setf(ios_base::showpos);
cout << "pi is " << scientific << pi << '\n';
```

resulting in:

```
pi is +3.1415926535897931e+00
```

Integer numbers can be represented in octal and hexadecimal base by:

```
cout << "63 octal is " << oct << 63 << ".\n";
cout << "63 hexadecimal is " << hex << 63 << ".\n";
cout << "63 decimal is " << dec << 63 << ".\n";
```

with the expected output:

```
63 octal is 77.
63 hexadecimal is 3f.
63 decimal is 63.
```

Boolean values are by default printed as integers 0 and 1. On demand, we can present them as true and false:

```
cout << "pi < 3 is " << (pi < 3) << '\n';
cout << "pi < 3 is " << boolalpha << (pi < 3) << '\n';
```

Finally, we can reset all the format options that we changed:

```
int old_precision= cout.precision();
cout << setprecision(16)
...
cout.unsetf(ios_base::adjustfield | ios_base::basefield
        | ios_base::floatfield | ios_base::showpos | ios_base::boolalpha);
cout.precision(old_precision);
```

Each option is represented by a bit in a status variable. To enable multiple options, we can combine their bit patterns with a binary OR.

C++20

1.7.6 New Formatting

⇒ c++20/fmt_example.cpp

As we have seen in the preceding section, traditional stream formatting requires a fair amount of typing. Alternatively, we can use the output from C with the printf function and format strings. This allows us to declare with few symbols what we've written with multiple I/O manipulators before.

Nonetheless, we advise against using printf, for two reasons: it can't be used with user types and it is not type safe. The format string is parsed at run time and the following arguments are treated with an obscure macro mechanism. If the arguments don't match the format string, the behavior is undefined and can cause program crashes. For instance, a string is passed as a pointer, and from the pointed address on, the bytes are read and printed as char until a binary 0 is found in memory. If we accidentally try printing an int as a string, the int value is misinterpreted as an address from which a sequence of char shall be printed. This will result in either absolute nonsensical output or (more likely) in a memory error if the address is inaccessible. We have to admit that recent compilers parse format strings (when known at compile time) and warn about argument mismatches.

The new <format> library in C++20 combines the expressibility of the format string with the type safety and the user extensibility of stream I/O and adds the opportunity to reorder the arguments in the output. Unfortunately, not even the latest compilers by the time of writing (GCC 12.0, Clang 13, and Visual Studio 16.9.6[14]) support the <format> library. Therefore, we used the prototype library <fmt> and refrained from wild-guessing how this would translate to the final standard interface. We strongly encourage you to migrate these examples yourself to <format> as soon as it's available to you. The syntax is different but the principles are the same.

Instead of a formal specification, we port some printf examples from cppreference.com to the new format:

```
print("Decimal:\t{} {} {:06} {} {:0} {:+} {:d}\n",
      1, 2, 3, 0, 0, 4, -1);
print("Hexadecimal:\t{:x} {:x} {:X} {:#x}\n", 5, 10, 10, 6);
print("Octal:\t\t{:o} {:#o} {:#o}\n", 10, 10, 4);
print("Binary:\t\t{:b} {:#b} {:#b}\n", 10, 10, 4);
```

This snippet prints:

```
Decimal:        1 2 000003 0 0 +4 -1
Hexadecimal:    5 a A 0x6
Octal:          12 012 04
Binary:         1010 0b1010 0b100
```

The first two numbers were just printed without giving any format information. The same output is generated when we ask for a decimal number with the format specifier "{:d}". The

14. We read, however, the announcement that VS 16.10 will be library complete.

third number will be printed (minimally) 6 characters wide and filled with leading `0`s. The specifier `+` allows us to force printing the sign for all numbers. `printf` allows for specifying `unsigned` output of numbers. That leads to incorrect large numbers when the value to print is negative. The `<format>` library refrains from user declarations of `unsigned` output since this information is already contained in the type of the according argument. If somebody feels the urge to print a negative value as a large positive one, they must convert it explicitly.

The second line demonstrates that we can print values hexadecimally—both with lower- and uppercase for the digits larger than 9. The specifier `"#"` generates the prefix `"0x"` used in hexadecimal literals. Likewise, we can print the values as octals and binaries, optionally with the according literal prefix.

With floating-point numbers we have more formatting options:

```
print("Default:\t{} {:g} {:g}\n", 1.5, 1.5, 1e20);
print("Rounding:\t{:f} {:.0f} {:.22f}\n", 1.5, 1.5, 1.3);
print("Padding:\t{:05.2f} {:.2f} {:5.2f}\n", 1.5, 1.5, 1.5);
print("Scientific:\t{:E} {:e}\n", 1.5, 1.5);
print("Hexadecimal:\t{:a} {:A}\n\n", 1.5, 1.3);
```

Then we get:

```
Default:        1.5 1.5 1e+20
Rounding:       1.500000 2 1.3000000000000000444089
Padding:        01.50 1.50  1.50
Scientific:     1.500000E+00 1.500000e+00
Hexadecimal:    0x1.8p+0 0X1.4CCCCCCCCCCCDP+0
```

With empty braces or only containing a colon, we get the default output. This corresponds to the format specifier `"{:g}"` and yields the same output as streams without the manipulators. The number of fractional digits can be given between a dot and the format specifier `"f"`. Then the value is rounded to that precision. If the requested number is larger than what is representable by the value's type, the last digits aren't very meaningful. A digit in front of the dot specifies the (minimal) width of the output. As with integers, we can request leading `0`s. Floating-point numbers can be printed in the scientific notation with either an upper- or lowercase `"e"` to start the exponential part. The hexadecimal output can be used to initialize a variable in another program with precisely the same bits.

The output can be redirected to any other `std::ostream`:[15]

```
print(std::cerr, "System error code = {}\n", 7);

ofstream error_file("error_file.txt");
print(error_file, "System error code = {}\n", 7);
```

In contrast to `printf`, arguments can now be reordered:

```
print("I'd rather be {1} than {0}.\n", "right", "happy");
```

In addition to referring the arguments by their positions, we can give them names:

```
print("Hello, {name}! The answer is {number}. Goodbye, {name}.\n",
      arg("name", name), arg("number", number));
```

15. Requires including `ostream.h` with the `<fmt>` library.

Or more concisely:

```
print("Hello, {name}! The answer is {number}. Goodbye, {name}.\n",
      "name"_a=name, "number"_a=number);
```

The example also demonstrates that we can print an argument multiple times.

Reordering arguments is very important in multilingual software to provide a natural phrasing. In Section 1.3.1 we printed the average of two values, and now we want to extend this example to five languages:

```
void print_average(float v1, float v2, int language)
{
    using namespace fmt;
    string formats[]= {
        "The average of {v1} and {v2} is {result}.\n",
        "{result:.6f} ist der Durchschnitt von {v1} und {v2}.\n",
        "La moyenne de {v1} et {v2} est {result}.\n",
        "El promedio de {v1} y {v2} es {result}.\n",
        "{result} corrisponde alla media di {v1} e {v2}.\n"};
    print (formats[language], "v1"_a= v1, "v2"_a= v2,
           "result"_a= (v1+v2)/2.0f);
}
```

Of course, the German version is the most pedantic one, requesting six decimal digits no matter what:

```
The average of 3.5 and 7.3 is 5.4.
5.400000 ist der Durchschnitt von 3.5 und 7.3.
La moyenne de 3.5 et 7.3 est 5.4.
El promedio de 3.5 y 7.3 es 5.4.
5.4 corrisponde alla media di 3.5 e 7.3.
```

Admittedly, this example would have worked without reordering the arguments but it nicely demonstrates the important possibility to separate the text and the formatting from the values. To store formatted text in a string we don't need a `stringstream` any longer but can do it directly with the function `format`.

Altogether, the new formatting is:

- **Compact**, as demonstrated in the examples above
- **Adaptable** to various output orders
- **Type-safe**, as an exception is thrown when an argument doesn't match
- **Extensible**, which we will see in Section 3.5.6

For those reasons, it is superior to the preceding techniques, and we therefore strongly advise using it as soon as sufficient compiler support is available.

1.7.7 Dealing with I/O Errors

To make one thing clear from the beginning: I/O in C++ is not fail-safe. Errors can be reported in different ways and our error handling must comply to them. Let us try the following example program:

```
int main ()
{
    std::ifstream infile("some_missing_file.xyz");

    int i;
    double d;
    infile >> i >> d;

    std::cout << "i is " << i << ", d is " << d << '\n';
    infile.close();
}
```

Although the file does not exist, the opening operation does not fail. We can even read from the nonexisting file and the program goes on. It is needless to say that the values in `i` and `d` are nonsense:

```
i is 1, d is 2.3452e-310
```

By default, the streams do not throw exceptions. The reason is historical: they are older than the exceptions, and later the behavior was kept to not break software written in the meantime. Another argument is that failing I/O is nothing exceptional but quite common, and checking errors (after each operation) would be natural.

To be sure that everything went well, we have to check error flags, in principle, after each I/O operation. The following program asks the user for new filenames until a file can be opened. After reading its content, we check again for success:

```
int main ()
{
    std::ifstream infile;
    std::string filename{"some_missing_file.xyz"};
    bool opened= false;
    while (!opened) {
        infile.open(filename);
        if (infile.good()) {
            opened= true;
        } else {
            std::cout << "The file '" << filename
                      << "' doesn't exist (or can't be opened),"
                      << "please give a new filename: ";
            std::cin >> filename;
        }
    }
    int i;
    double d;
    infile >> i >> d;
    if (infile.good())
        std::cout << "i is " << i << ", d is " << d << '\n';
    else
        std::cout << "Could not correctly read the content.\n";
    infile.close();
}
```

You can see from this simple example that writing robust applications with file I/O can create some work. If we want to use exceptions, we have to enable them during run time for each stream:

```
cin.exceptions(ios_base::badbit | ios_base::failbit);
cout.exceptions(ios_base::badbit | ios_base::failbit);

std::ifstream infile("f.txt");
infile.exceptions(ios_base::badbit | ios_base::failbit);
```

The streams throw an exception every time an operation fails or when they are in a "bad" state. Exceptions could be thrown at (unexpected) file end as well. However, the end of file is more conveniently handled by testing (e.g., `while (!f.eof())`).

In the preceding example, the exceptions for `infile` are only enabled after opening the file (or the attempt thereof). For checking the opening operation, we have to create the stream first, then turn on the exceptions, and finally open the file explicitly. Enabling the exceptions gives us at least the guarantee that all I/O operations went well when the program terminates properly. We can make our program more robust by catching possible exceptions.

The exceptions in file I/O only protect us partially from making errors. For instance, the following small program is obviously wrong (types don't match and numbers aren't separated):

```
void with_io_exceptions(ios& io)
{    io.exceptions(ios_base::badbit | ios_base::failbit); }

int main ()
{
    std::ofstream outfile;
    with_io_exceptions(outfile);
    outfile.open("f.txt");

    double o1= 5.2, o2= 6.2;
    outfile << o1 << o2 << std::endl;  // no separation
    outfile.close();

    std::ifstream infile;
    with_io_exceptions(infile);
    infile.open("f.txt");

    int  i1, i2;
    char c;
    infile >> i1 >> c >> i2;           // mismatching types
    std::cout << "i1 = " << i1 << ", i2 = " << i2 << "\n";
}
```

Nonetheless, it does not throw exceptions and fabricates the following output:

```
i1 = 5, i2 = 26
```

As we all know, testing does not prove the correctness of a program. This is even more obvious when I/O is involved. Stream input reads the incoming characters and passes them as values of the corresponding variable type, e.g., `int` when setting `i1`. It stops at the first character that cannot be part of the value, first at the dot for the `int` value `i1`. If we read another `int` afterward, it would fail because an empty string cannot be interpreted as an `int` value. But we do not; instead we read a `char` next to which the dot is assigned. When parsing the input for `i2` we find first the fractional part from `o1` and then the integer part from `o2` before we get a character that cannot belong to an `int` value.

Unfortunately, not every violation of the grammatical rules causes an exception in practice: .3 parsed as an `int` yields zero (while the next input probably fails); -5 parsed as an `unsigned` results in 4294967291 (when `unsigned` is 32 bits long). The narrowing principle apparently has not found its way into I/O streams yet (if it ever will for backward compatibility's sake).

At any rate, the I/O part of an application needs close attention. Numbers must be separated properly (e.g., by spaces) and read with the same type as they were written. Floating-point values can also vary in their local representation and it is therefore recommended to store and read them without internationalization (i.e., with neutral C locale) when the program will run on different systems. Another challenge can be branches in the output so that the file format can vary. The input code is considerably more complicated and might even be ambiguous.

There are two other forms of I/O we want to mention: binary and C-style I/O. The interested reader will find them in Sections A.2.6 and A.2.7, respectively. You can also read this later when you need it.

1.7.8 Filesystem

C++17

⇒ `c++17/filesystem_example.cpp`

A library that was overdue in C++ is `<filesystem>`. Now we can list all files in a directory and ask for their type, for instance:[16]

```
namespace fs = std::filesystems;
for (auto & p : fs::directory_iterator("."))
    if (is_regular_file(p))
        cout ≪ p ≪ " is a regular file.\n"; // Error in Visual Studio
    else if(is_directory(p))
        cout ≪ p ≪ " is a directory.\n";
    else
        cout ≪ p ≪ " is neither regular file nor directory.\n";
```

printed out for a directory containing one executable and a subdirectory:

```
.\\cpp17_vector_any.exe is a regular file.
.\\sub is a directory.
```

16. The example doesn't compile with Visual Studio at the moment since the output operator for `directory_entry` isn't found.

The filesystem library also allows us to copy files, create symbolic and hard links, and rename files directly within a C++ program in a portable manner. Boost.Filesystem is a reasonable alternative if your compiler is not handling file operations properly or you are obliged to hold back to older standards.

1.8 Arrays, Pointers, and References

1.8.1 Arrays

The intrinsic array support of C++ has certain limitations and some strange behaviors. Nonetheless, we feel that every C++ programmer should know it and be aware of its problems. An array is declared as follows:

```
int x[10];
```

The variable `x` is an array with 10 `int` entries. In standard C++, the size of the array must be constant and known at compile time. Some compilers (e.g., `g++`) support run-time sizes.

Arrays are accessed by square brackets: `x[i]` is a reference to the i-th element of `x`. The first element is `x[0]`; the last one is `x[9]`. Arrays can be initialized at the definition:

```
float v[]= {1.0, 2.0, 3.0}, w[]= {7.0, 8.0, 9.0};
```

In this case, the array size is deduced.

C++11

The list initialization in C++11 forbids the narrowing of the values. This will rarely make a difference in practice. For instance, the following:

```
int v[]= {1.0, 2.0, 3.0};    // Error in C++11: narrowing
```

was legal in C++03 but not in C++11 since the conversion from a floating-point literal to `int` potentially loses precision. However, we would not write such ugly code anyway.

Operations on arrays are typically performed in loops; e.g., to compute $x = v - 3w$ as a vector operation is realized by

```
float x[3];
for (int i= 0; i < 3; ++i)
    x[i]= v[i] - 3.0 * w[i];
```

We can also define arrays of higher dimensions:

```
float A[7][9];      // a 7 by 9 matrix
int   q[3][2][3];   // a 3 by 2 by 3 array
```

The language does not provide linear algebra operations upon the arrays. Implementations based on arrays are inelegant and error prone. For instance, a function for a vector addition would look like this:

```
void vector_add(unsigned size, const double v1[], const double v2[],
                double s[])
{
    for (unsigned i= 0; i < size; ++i)
        s[i]= v1[i] + v2[i];
}
```

Note that we passed the size of the arrays as the first function parameter whereas array parameters don't contain size information.[17] In this case, the function's caller is responsible for passing the correct size of the arrays:

```
int main ()
{
    double x[]= {2, 3, 4}, y[]= {4, 2, 0}, sum[3];
    vector_add(3, x, y, sum);
    ...
}
```

Since the array size is known during compilation, we can compute it by dividing the byte size of the array by that of a single entry:

```
vector_add(sizeof x / sizeof x[0], x, y, sum);
```

With this old-fashioned interface, we are also unable to test whether our arrays match in size. Sadly enough, C and Fortran libraries with such interfaces where size information is passed as function arguments are still realized today. They crash at the slightest user mistake, and it can take enormous effort to trace back the reasons for crashing. For that reason, we will show in this book how we can realize our own math software that is easier to use and less prone to errors. Hopefully, future C++ standards will come with more higher mathematics, especially a linear-algebra library.

Arrays have the following two disadvantages:

1. Indices are not checked before accessing an array, and we can find ourselves outside the array when the program crashes with segmentation fault/violation. This is not even the worst case; at least we see that something goes wrong. The false access can also mess up our data; the program keeps running and produces entirely wrong results with whatever consequence you can imagine. We could even overwrite the program code. Then our data is interpreted as machine operations leading to any possible nonsense.

2. The size of the array must be known at compile time.[18] This is a serious problem when we fill an array with data from a file:

   ```
   ifstream ifs("some_array.dat");
   ifs ≫ size;
   float v[size];   // Error: size not known at compile time
   ```

   This does not work when the number of entries in the file varies.

The first problem can only be solved with new array types and the second one with dynamic allocation. This leads us to pointers.

17. When passing arrays of higher dimensions, only the first dimension can be open while the others must be known during compilation. However, such programs can easily get nasty and we have better techniques for it in C++.

18. Some compilers support run-time values as array sizes. Since this is not guaranteed with other compilers, one should avoid this in portable software. This feature was considered for C++14, but its inclusion was postponed (maybe forever) as not all subtle issues could be solved on each and every platform.

1.8.2 Pointers

A pointer is a variable that contains a memory address. This address can be that of another variable provided by the address operator (e.g., `&x`) or dynamically allocated memory. Let's start with the latter as we were looking for arrays of dynamic size.

```
int* y= new int[10];
```

This allocates an array of 10 `int`. The size can now be chosen at run time. We can also implement the vector-reading example from the preceding section:

```
ifstream ifs("some_array.dat");
int size;
ifs ≫ size;
float* v= new float[size];
for (int i= 0; i < size; ++i)
    ifs ≫ v[i];
```

Pointers bear the same danger as arrays: accessing data out of range, which can cause program crashes or silent data invalidation. When dealing with dynamically allocated arrays, it is the programmer's responsibility to store the array size.

Furthermore, the programmer is responsible for releasing the memory when it is not needed anymore. This is done by:

```
delete[] v;
```

Since arrays as function parameters are compatible with pointers, the `vector_add` function from Section 1.8.1 works with pointers as well:

```
int main (int argc, char* argv[])
{
    const int size= 3;
    double *x= new double[size], *y= new double[size],
           *sum= new double[3];
    for (unsigned i= 0; i < size; ++i)
        x[i]= i+2, y[i]= 4-2*i;
    vector_add(size, x, y, sum);
    ...
}
```

With pointers, we cannot use the `sizeof` trick; it would only give us the byte size of the pointer itself, which is of course independent of the number of entries. Other than that, pointers and arrays are interchangeable in most situations: a pointer can be passed as an array argument (as in the preceding listing) and an array as a pointer argument. The only place where they are really different is the definition: whereas defining an array of size `n` reserves space for `n` entries, defining a pointer only reserves the space to hold an address.

Since we started with arrays, we took the second step before the first one regarding pointer usage. The simple use of pointers is allocating one single data item:

```
int* ip= new int;
```

Releasing this memory is performed by:

```
delete ip;
```

Note the duality of allocation and release: the single-object allocation requires a single-object release and the array allocation demands an array release. Otherwise the run-time system might handle the deallocation incorrectly and crash at this point or corrupt some data. Pointers can also refer to other variables:

```
int  i= 3;
int* ip2= &i;
```

The operator `&` takes an object and returns its address. The opposite operator is `*` which takes an address and returns an object:

```
int  j= *ip2;
```

This is called *Dereferencing*. Given the operator priorities and the grammar rules, the meaning of the symbol `*` as dereference or multiplication cannot be confused—at least not by the compiler.

Pointers that are not initialized contain a random value (whatever bits are set in the corresponding memory). Using uninitialized pointers can cause any kind of error. To say explicitly that a pointer is not pointing to something, we should set it to C++11

```
int* ip3= nullptr;    // >= C++11
int* ip4{};           // ditto
```

or in old compilers:

```
int* ip3= 0;          // better not in C++11 and later
int* ip4= NULL;       // ditto
```

C++11 The address 0 is guaranteed never to be used for applications, so it is safe to indicate this way that the pointer is empty (not referring to something). Nonetheless, the literal 0 does not clearly convey its intention and can cause ambiguities in function overloading. The macro `NULL` is not better: it just evaluates to `0`. C++11 introduces `nullptr` as a keyword for a pointer literal. It can be assigned to or compared with all pointer types. As `nullptr` cannot be confused with types that aren't pointers and it is self-explanatory, this is the preferred notation. The initialization with an empty braced list also sets a `nullptr`.

Very frequent errors with pointers are *Memory Leaks*. For instance, our array `y` became too small and we want to assign a new array:

```
int* y= new int[10];
// some stuff
y= new int[15];
```

Initially we allocated space for 10 `int` values. Later we needed more and allocated 15 `int` locations. But what happened to the memory that we allocated before? It is still there but we have no access to it anymore. We cannot even release it because this requires the address of that memory block. This memory is lost for the rest of our program execution. Only when the program is finished will the operating system be able to free it. In our example, we only lost 40 bytes out of several gigabytes that we might have. But if this happens in an iterative process, the unused memory grows continuously until at some point the whole (virtual) memory is exhausted.

Even if the wasted memory is not critical for the application at hand, when we write high-quality scientific software, memory leaks are unacceptable. When many people are using our software, sooner or later somebody will criticize us for it and eventually discourage other people from using it. Fortunately, tools are available to help us find memory leaks, as demonstrated in Section B.3.

The demonstrated issues with pointers are not intended to be "fun killers.". And we do not discourage the use of pointers. Many things can only be achieved with pointers: lists, queues, trees, graphs, et cetera. But pointers must be used with care to avoid all the really severe problems mentioned above. There are three strategies to minimize pointer-related errors:

1. Use standard containers from the standard library or other validated libraries. `std::vector` from the standard library provides us all the functionality of dynamic arrays, including resizing and range check, and the memory is released automatically.

2. Encapsulate dynamic memory management in classes. Then we have to deal with it only once per class.[19] When all memory allocated by an object is released when the object is destroyed, it does not matter how often we allocate memory. If we have 738 objects with dynamic memory, then it will be released 738 times. The memory should be allocated in the object construction and deallocated in its destruction. This principle is called *Resource Acquisition Is Initialization* (RAII). In contrast, if we called `new` 738 times, partly in loops and branches, can we be sure that we have called `delete` exactly 738 times? We know that there are tools for this, but these are errors that are better prevented than fixed.[20] Of course, the encapsulation idea is not idiot-proof, but it is much less work to get it right than sprinkling (raw) pointers all over our program. We will discuss RAII in more detail in Section 2.4.2.1.

3. Use smart pointers, which we will introduce in the next section (§1.8.3).

Pointers serve two purposes:

1. Referring to objects

2. Managing dynamic memory

The problem with so-called *Raw Pointers* is that we have no notion whether a pointer is only referring to data or also in charge of releasing the memory when it is not needed any longer. To make this distinction explicit at the type level, we can use *Smart Pointers*.

19. It is safe to assume that there are many more objects than classes; otherwise there is something wrong with the entire program design.

20. In addition, a tool only shows that the current run had no errors, but this might be different with other input.

1.8.3 Smart Pointers

C++11

Three new smart-pointer types are introduced in C++11: `unique_ptr`, `shared_ptr`, and `weak_ptr`. The already-existing smart pointer from C++03 named `auto_ptr` is generally considered as a failed attempt on the way to `unique_ptr` since the language was not ready at the time. It was therefore removed in C++17. All smart pointers are defined in the header `<memory>`. If you cannot use C++11 features on your platform (e.g., in embedded programming), the smart pointers in Boost are a decent replacement.

1.8.3.1 Unique Pointer

C++11

This pointer's name indicates *Unique Ownership* of the referred data. It can be used essentially like an ordinary pointer:

```
#include <memory>

int main ()
{
    unique_ptr<double> dp{new double};
    *dp= 7;
    ...
    cout ≪ "The value of *dp is " ≪ *dp ≪ endl;
}
```

The main difference from a raw pointer is that the memory is automatically released when the pointer expires. Therefore, it is a bug to assign addresses that are not allocated dynamically:

```
double d= 7.2;
unique_ptr<double> dd{&d}; // Error: causes illegal deletion
```

The destructor of pointer `dd` will try to delete `d`. To guarantee the uniqueness of the memory ownership, a `unique_ptr` cannot be copied:

```
unique_ptr<double> dp2{dp}; // Error: no copy allowed
dp2= dp;                    // ditto
```

However, we can transfer the memory location to another `unique_ptr`:

```
unique_ptr<double> dp2{move(dp)}, dp3;
dp3= move(dp2);
```

using `move`. The details of move semantics will be discussed in Section 2.3.5. In our example, the ownership of the referred memory is first passed from `dp` to `dp2` and then to `dp3`. `dp` and `dp2` are `nullptr` afterward, and the destructor of `dp3` will release the memory. In the same manner, the memory's ownership is passed when a `unique_ptr` is returned from a function. In the following example, `dp3` takes over the memory allocated in `f()`:

```
std::unique_ptr<double> f()
{   return std::unique_ptr<double>{new double}; }

int main ()
{
        unique_ptr<double> dp3;
        dp3= f();
    }
```

In this case, `move()` is not needed since the function result is a temporary that will be moved (again, details in §2.3.5).

Unique pointer has a special implementation[21] for arrays. This is necessary for properly releasing the memory (with `delete[]`). In addition, the specialization provides array-like access to the elements:

```
    unique_ptr<double[]> da{new double[3]};
    for (unsigned i= 0; i < 3; ++i)
        da[i]= i+2;
```

In return, the `operator*` is not available for arrays.

An important benefit of `unique_ptr` is that it has absolutely no overhead over raw pointers: neither in time nor in memory.

Further reading: An advanced feature of unique pointers is to provide its own *Deleter*; for details see [40, §5.2.5f], [62, §34.3.1], or an online reference (e.g., `cppreference.com`).

C++11

1.8.3.2 Shared Pointer

As its name indicates, a `shared_ptr` manages memory that is shared between multiple parties (each holding a pointer to it). The memory is automatically released as soon as no `shared_ptr` is referring the data any longer. This can simplify a program considerably, especially with complicated data structures. An extremely important application area is concurrency: the memory is automatically freed when all threads have terminated their access to it. In contrast to a `unique_ptr`, a `shared_ptr` can be copied as often as desired, e.g.:

```
    shared_ptr<double> f()
    {
        shared_ptr<double> p1{new double};
        shared_ptr<double> p2{new double}, p3= p1;
        cout ≪ "p3.use_count() = " ≪ p3.use_count() ≪ endl;
        return p3;
    }

    int main ()
    {
        shared_ptr<double> p= f();
        cout ≪ "p.use_count() = " ≪ p.use_count() ≪ endl;
    }
```

In this example, we allocated memory for two `double` values: in `p1` and in `p2`. The pointer `p1` is copied into `p3` so that both point to the same memory, as illustrated in Figure 1–1.

21. Specialization will be discussed in §3.5.1 and §3.5.3.

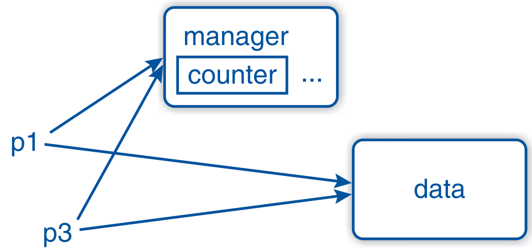

Figure 1–1: Shared pointer in memory

We can see this from the output of `use_count`:

```
p3.use_count() = 2
p.use_count() = 1
```

When `f` returns, the pointers are destroyed and the memory referred to by `p2` is released (without ever being used). The second allocated memory block still exists since `p` from the `main` function is still referring to it.

If possible, a `shared_ptr` should be created with `make_shared`:

```
shared_ptr<double> p1= make_shared<double>();
```

Then the internal and the user data are stored together in memory—as shown in Figure 1–2—and the memory caching is more efficient. `make_shared` also provides better exception safety because we have only one memory allocation. As it is obvious that `make_shared` returns a shared pointer, we can use automatic type detection (§3.4.1) for simplicity:

```
auto p1= make_shared<double>();
```

We have to admit that a `shared_ptr` has some overhead in memory and run time. On the other hand, the simplification of our programs thanks to `shared_ptr` is in most cases worth some small overhead.

Further reading: For deleters and other details of `shared_ptr` see the library reference [40, §5.2], [62, §34.3.2], or an online reference.

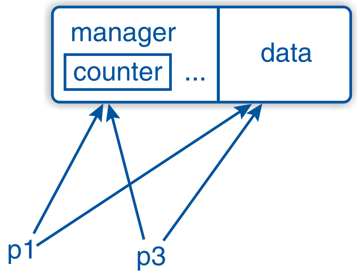

Figure 1–2: Shared pointer in memory after `make_shared`

C++11

1.8.3.3 Weak Pointer

A problem that can occur with shared pointers is *Cyclic References* that impede the memory to be released. Such cycles can be broken by `weak_ptr`s. They do not claim ownership of the memory, not even a shared one. At this point, we only mention them for completeness and suggest that you read appropriate references when their need is established: [40, §5.2.2], [62, §34.3.3], or `cppreference.com`.

For managing memory dynamically, there is no alternative to pointers. To only refer to other objects, we can use another language feature called *Reference* (surprise, surprise), which we introduce in the next section.

1.8.4 References

The following code introduces a reference:

```
int  i= 5;
int& j= i;
j= 4;
std::cout ≪ "i = " ≪ i ≪ '\n';
```

The variable `j` is referring to `i`. Changing `j` will also alter `i` and vice versa, as in the example. `i` and `j` will always have the same value. One can think of a reference as an alias: it introduces a new name for an existing object or subobject. Whenever we define a reference, we must directly declare what it refers to (other than pointers). It is not possible to refer to another variable later.

References are even more useful for function arguments (§1.5), for referring to parts of other objects (e.g., the seventh entry of a vector), and for building views (e.g., §5.2.3).

As a compromise between pointers and references, C++11 offers a `reference_wrapper` class which behaves similarly to references but avoids some of their limitations. For instance, it can be used within containers; see §4.4.8.

C++11

1.8.5 Comparison between Pointers and References

The main advantage of pointers over references is the ability of dynamic memory management and address calculation. On the other hand, references are forced to refer to existing locations.[22] Thus, they do not leave memory leaks (unless you play really evil tricks), and they have the same notation in usage as the referred object. Unfortunately, it is almost impossible to construct containers of references (use `reference_wrapper` instead).

In short, references are not fail-safe but are much less error-prone than pointers. Pointers should only be used when dealing with dynamic memory, for instance, when we create data structures like lists or trees dynamically. Even then we should do this via well-tested types or encapsulate the pointer(s) within a class whenever possible. Smart pointers take care of memory allocation and should be preferred over raw pointers, even within classes. The pointer–reference comparison is summarized in Table 1-9.

22. References can also refer to arbitrary addresses but you must work harder to achieve this. For your own safety, we will not show you how to make references behave as badly as pointers.

Table 1–9: Comparison between Pointers and References

Feature	Pointers	References
Referring to defined location		✓
Mandatory initialization		✓
Avoidance of memory leaks		✓
Object-like notation		✓
Memory management	✓	
Address calculation	✓	
Build containers thereof	✓	

1.8.6 Do Not Refer to Outdated Data!

Function-local variables are only valid within the function's scope, for instance:

```
double& square_ref(double d) // Error: returns stale reference
{
    double s= d * d;
    return s;                // Error: s will be out of scope
}
```

Here, our function result refers the local variable `s` which does not exist after the function end. The memory where it was stored is still there and we might be lucky (mistakenly) that it is not overwritten yet. But this is nothing we can count on. Finally, such situation-dependent errors are worse than permanent ones: on the one hand, they are harder to debug, and on the other, they can go unnoticed despite extensive testing and cause greater damage later in real usage.

References to variables that no longer exist are called *Stale References*. Sadly enough, we have seen such examples even in some web tutorials.

The same applies to pointers:

```
double* square_ptr(double d) // Error: returns dangling pointer
{
    double s= d * d;
    return &s;               // Error: s will be out of scope
}
```

This pointer holds the address of a local variable that has already gone out of scope. This is called a *Dangling Pointer*.

Returning references or pointers can be correct in member functions when referring to member data (see Section 2.6) or to `static` variables.

Advice

Only return pointers, references, and objects with reference semantic that point to dynamically allocated data, to data that existed before the function was called, or to `static` data.

By "objects with reference semantic" we mean objects that don't contain all their data but refer to external data that is not replicated when the object is copied; in other words, objects that have, at least partially, the behavior of a pointer and thus the risk of referring to something that doesn't exist anymore. In Section 4.1.2 we will introduce iterators, which are classes from the standard library or from users with a pointer-like behavior, and the danger of referring to already destroyed objects.

Compilers are fortunately getting better and better at detecting such errors, and all current compilers should warn us of obviously stale references or dangling pointers as in the examples above. But the situation is not always so obvious, especially when we have a user class with reference semantics.

1.8.7 Containers for Arrays

As alternatives to the traditional C arrays, we want to introduce two container types that can be used in similar ways but cause fewer problems.

1.8.7.1 Standard Vector

Arrays and pointers are part of the C++ core language. In contrast, `std::vector` belongs to the standard library and is implemented as a class template. Nonetheless, it can be used very similarly to arrays. For instance, the example from Section 1.8.1 of setting up two arrays `v` and `w` looks for vectors as follows:

```
#include <vector>

int main ()
{
    std::vector<float> v(3), w(3);
    v[0]= 1; v[1]= 2; v[2]= 3;
    w[0]= 7; w[1]= 8; w[2]= 9;
}
```

The size of the vector does not need to be known at compile time. Vectors can even be resized during their lifetime, as will be shown in Section 4.1.3.1.

C++11 The element-wise setting is not particularly concise. C++11 allows the initialization with initializer lists:

```
std::vector<float> v= {1, 2, 3}, w= {7, 8, 9};
```

In this case, the size of the vector is implied by the length of the list. The vector addition shown before can be implemented more reliably:

```
void vector_add(const vector<float>& v1, const vector<float>& v2,
                vector<float>& s)
{
    assert(v1.size() == v2.size());
    assert(v1.size() == s.size());
    for (unsigned i= 0; i < v1.size(); ++i)
        s[i]= v1[i] + v2[i];
}
```

In contrast to C arrays and pointers, the `vector` arguments know their sizes and we can now check whether they match. Note that the array size can be deduced with templates, which we'll show later in Section 3.3.2.1.

Vectors are copyable and can be returned by functions. This allows us to use a more natural notation:

```
vector<float> add(const vector<float>& v1, const vector<float>& v2)
{
    assert(v1.size() == v2.size());
    vector<float> s(v1.size());
    for (unsigned i= 0; i < v1.size(); ++i)
        s[i]= v1[i] + v2[i];
    return s;
}

int main ()
{
    std::vector<float> v= {1, 2, 3}, w= {7, 8, 9}, s= add(v, w);
}
```

This implementation is potentially more expensive than the preceding one where the target vector is passed in as a reference. We will later discuss the possibilities of optimization: on both the compiler and user sides. In our experience, it is more important to start with a productive interface and deal with performance later. It is easier to make a correct program fast than to make a fast program correct. Thus, aim first for a good program design. In almost all cases, the favorable interface can be realized with sufficient performance.

The container `std::vector` is not a vector in the mathematical sense. There are no arithmetic operations. Nonetheless, the container proved very useful in scientific applications to handle nonscalar intermediate results.

1.8.7.2 valarray

A `valarray` is a one-dimensional array with element-wise operations; even the multiplication is performed element-wise. Operations with a scalar value are performed respectively with each element of the `valarray`. Thus, the `valarray` of a floating-point number is a vector space.

The following example demonstrates some operations:

```
#include <iostream>
#include <valarray>

int main ()
{
    std::valarray<float> v= {1, 2, 3}, w= {7, 8, 9},
                         s= v + 2.0f * w;
    v= sin(s);
    for (float x : v)
        std::cout << x << ' ';
    std::cout << '\n';
}
```

Note that a `valarray<float>` can only operate with itself or `float`. For instance, `2 * w` would fail since it is an unsupported multiplication of `int` with `valarray<float>`.

A strength of `valarray` is the ability to access slices of it. This allows us to *Emulate* matrices and higher-order tensors, including their respective operations. Nonetheless, due to the lack of direct support of most linear-algebra operations, `valarray` is not widely used in the numeric community. We also recommend using established C++ libraries for linear algebra. Hopefully, future standards will contain one.

To complete the topic of dynamic memory management, we refer to Section A.2.8 where *Garbage Collection* is briefly described. The bottom line is that a C++ programmer can pretty well live without it and no compiler supports it (yet?) anyway.

1.9 Structuring Software Projects

A big problem of large projects is name conflicts. For this reason, we will discuss how macros aggravate this problem. On the other hand, we will show later in Section 3.2.1 how namespaces help us master name conflicts.

In order to understand how the files in a C++ software project interact, it is necessary to understand the build process, i.e., how an executable is generated from the sources. This will be the subject of our second subsection. In this light, we will present the macro mechanism and other language features.

First, we will briefly discuss a feature that contributes to structuring a program: comments.

1.9.1 Comments

The primary purpose of a comment is evidently to describe in plain language what is not obvious to everybody from the program sources, like this:

```
// Transmogrification of the anti-binoxe in O(n log n)
while (cryptographic(trans_thingy) < end_of(whatever)) {
    ....
```

Often, the comment is a clarifying pseudo-code of an obfuscated implementation:

```
// A= B * C
for ( ... ) {
    int x78zy97= yo6954fq, y89haf= q6843, ...
    for ( ... ) {
        y89haf+= ab6899(fa69f) + omygosh(fdab); ...
        for ( ... ) {
             A(dyoa929, oa9978)+= ...
```

In such a case, we should ask ourselves whether we can restructure our software so that such obscure implementations are realized once, in a dark corner of a library, and everywhere else we write clear and simple statements such as:

```
A= B * C;
```

as program and not as pseudo-code. This is one of the main goals of this book: to show you how to write the short expression you want while the implementation under the hood squeezes out the maximal performance.

Another frequent usage of comments is to remove snippets of code temporarily to experiment with alternative implementations, e.g.:

```
for ( ... ) {
    // int x= a + b + c
    int x = a + d + e;
    for ( ... ) {
        ...
```

Like C, C++ provides a form of block comments, surrounded by /* and */. They can be used to render an arbitrary part of a code line or multiple lines into a comment. Unfortunately, they cannot be nested: no matter how many levels of comments are opened with /*, the first */ ends all block comments. Many programmers run into this trap sometimes: they want to comment out a longer fraction of code that already contains a block comment so that the comment ends earlier than intended, for instance:

```
for ( ... ) {
    /* int x78zy97= yo6954fq;      // start new comment
    int x78zy98= yo6953fq;
    /* int x78zy99= yo6952fq;      // start old comment
    int x78zy9a= yo6951fq;    */   // end old comment
    int x78zy9b= yo6950fq;    */   // end new comment (presumably)
    int x78zy9c= yo6949fq;
    for ( ... ) {
```

Here, the line for setting x78zy9b should have been disabled, but the preceeding */ terminated the comment prematurely.

Nested comments can be realized (correctly) with the preprocessor directive #if as we will illustrate in Section 1.9.2.4. Another way to deactivate multiple lines conveniently is by using the appropriate function of IDEs and language-aware editors. At any rate, commenting out code fractions should only be a temporary solution during development when we investigate various approaches. Once we settle for a certain option, we can delete all the unused code and rely on our version control system that it will remain available for possible later modifications.

1.9.2 Preprocessor Directives

In this section, we will present the commands (directives) that can be used in preprocessing. As they are mostly language independent, we recommend limiting their usage to an absolute minimum, especially macros.

1.9.2.1 Macros

"Almost every macro demonstrates a flaw in the programming language, in the program, or the programmer."

—Bjarne Stroustrup

This is an old technique of code reuse by expanding macro names to their text definition, potentially with arguments. The use of macros gives a lot of possibilities to empower your program but much more for ruining it. Macros are resistant against namespaces, scopes, or any other language feature because they are reckless text substitution without any notion of types. Unfortunately, some libraries define macros with common names like `major`. We uncompromisingly undefine such macros, e.g., `#undef major`, without mercy for people who might want to use those macros. With Visual Studio we have—even today!!!—`min` and `max` as macros, and we strongly advise you to disable this by compiling with `/DNOMINMAX`.[23] Almost all macros can be replaced by other techniques (constants, templates, inline functions). But if you really do not find another way of implementing something:

Macro Names

Use `LONG_AND_UGLY_NAMES_IN_CAPITALS` for macros!

Macros can create weird problems in almost every thinkable and unthinkable way. To give you a general idea, we look at a few examples in Appendix A.2.9 and give some tips for how to deal with them. Feel free to postpone the reading until you run into an issue.

Macro Use

Replace every macro with another language feature whenever possible and use macros only when nothing else works.

As you will see throughout this book, C++ provides better alternatives like constants, `inline` functions, templates, and `constexpr`.

1.9.2.2 Inclusion

To keep the C language simple, many features such as I/O were excluded from the core language and realized by the library instead. C++ follows this design and realizes new features whenever possible by the standard library, and yet nobody would call C++ a simple language.

As a consequence, almost every program needs to include one or more headers. The most frequent one is that for I/O, as seen before:

```
#include <iostream>
```

The preprocessor searches that file in standard include directories like `/usr/include`, and `/usr/local/include` on Unix-like systems. We can add more directories to this search path with a compiler flag—usually `-I` in the Unix/Linux/Mac OS world and `/I` in Windows.

When we write the filename within double quotes, e.g.:

```
#include "herberts_math_functions.hpp"
```

the compiler usually searches first in the current directory and then in the standard paths.[24] This is equivalent to quoting with angle brackets and adding the current directory to the

23. Technically, the macros come from `Windows.h` and not from Visual Studio itself, but this difference matters only a little once we run into trouble with it, and this is not so rare as this header is frequently used.

24. However, which directories are searched with double-quoted filenames is implementation dependent and not stipulated by the standard.

search path. Some people argue that angle brackets should only be used for system headers and user headers should use double quotes (but we do not agree with them).

To avoid name clashes, often the include's parent directory is added to the search path and a relative path is used in the directive:

```
#include "herberts_includes/math_functions.hpp"
#include <another_project/math_functions.h>
```

The slashes are portable and also work under Windows (where both regular slashes and backslashes can be used for subdirectories).

Include guards: Frequently used header files may be included multiple times in one source file due to indirect inclusion. To avoid forbidden repetitions and to limit the text expansion, so-called *Include Guards* ensure that only the first inclusion is performed. These guards are ordinary macros that state the inclusion of a certain file. A typical include file looks like this:

```
// Author: me
// License: Pay me $100 every time you read this

#ifndef HERBERTS_MATH_FUNCTIONS_INCLUDE
#define HERBERTS_MATH_FUNCTIONS_INCLUDE

#include <cmath>

double sine(double x);
...

#endif // HERBERTS_MATH_FUNCTIONS_INCLUDE
```

Thus, the content of the file is only included when the guard is not yet defined. Within the content, we define the guard to suppress further inclusions.

As with all macros, we have to pay close attention that the name is unique, not only in our project but also within all other headers that we include directly or indirectly. Ideally the name should represent the project and filename. It can also contain project-relative paths or namespaces (§3.2.1). It is a common practice to terminate it with `_INCLUDE` or `_HEADER`.

Accidentally reusing a guard can produce a multitude of different error messages. In our experience it can take an unpleasantly long time to discover the root of that evil. Advanced developers generate them automatically from the aforementioned information or by using random generators.

A convenient alternative is `#pragma once`. The preceding example simplifies to:

```
// Author: me
// License: Pay me $100 every time you read this

#pragma once

#include <cmath>

double sine(double x);
...
```

Pragmas are compiler-specific extensions, which is why we cannot count on them being portable. `#pragma once` is, however, supported by all major compilers and certainly the pragma with the highest portability. In addition to the shorter notation, we can also delegate responsibility to avoid double inclusions to the compiler.

The advanced technology for organizing code in files is introduced in C++20 with modules. We will present them in Section 7.3.

1.9.2.3 Conditional Compilation

An important and necessary usage of preprocessor directives is the control of conditional compilation. The preprocessor provides the directives `#if`, `#else`, `#elif`, and `#endif` for branching. Conditions can be comparisons, checking for definitions, or logical expressions thereof. The directives `#ifdef` and `#ifndef` are shortcuts for, respectively:

```
#if defined(MACRO_NAME)

#if !defined(MACRO_NAME)
```

The long form must be used when the definition check is combined with other conditions. Likewise, `#elif` is a shortcut for `#else` and `#if`.

In a perfect world, we would only write portable standard-compliant C++ programs. In reality, we sometimes have to use nonportable libraries. Say we have a library that is only available on Windows, and more precisely only with Visual Studio (where the macro `_MSC_VER` is predefined). For all other relevant compilers, we have an alternative library. The simplest way for the platform-dependent implementation is to provide alternative code fragments for different compilers:

```
#ifdef _MSC_VER
    ... Windows code
#else
    ... Linux/Unix code
#endif
```

Similarly, we need conditional compilation when we want to use a new language feature that is not available on all target platforms, say, modules (§7.3):

```
#ifdef MY_LIBRARY_WITH_MODULES
   ... well-structured library as modules
#else
   ... portable library in old-fashioned way
#endif
```

Here we can use the feature when available and still keep the portability to compilers without this feature. Of course, we need reliable tools that define the macro only when the feature is really available.

C++20

Alternatively, we can rely on compiler developers' opinions as to whether this feature is properly supported. To this end, C++20 introduced a macro for each feature introduced since C++11—for instance, `__cpp_modules` for module support—so that our example now reads:

```
#ifdef __cpp_modules
   ... well-structured library as modules
```

```
#else
   ... portable library in old-fashioned way
#endif
```

The value of this macro is the year and month when the feature was added to the standard (draft). For evolving core language and library features, this allows us to find out which version is actually supported. For instance, the `<chrono>` library (Section 4.5) grew over time, and to check whether its C++20 functionality is available on our system, we can use the value of `__cpp_lib_chrono`:

```
#if __cpp_lib_chrono >= 201907L
```

Conditional compilation is quite powerful but comes at a price: the maintenance of the sources and the testing are more laborious and error-prone. These disadvantages can be lessened by well-designed encapsulation so that the different implementations are used over common interfaces.

1.9.2.4 Nestable Comments

The directive `#if` can be used to comment out code blocks:

```
#if 0
    ... Here we wrote pretty evil code! One day we will fix it. Seriously.
#endif
```

The advantage over `/* ... */` is that it can be nested:

```
#if 0
    ... Here the nonsense begins.
#if 0
    ... Here we have nonsense within nonsense.
#endif
    ... The finale of our nonsense. (Fortunately ignored.)
#endif
```

Nonetheless, this technique should be used with moderation: if three-quarters of the program are comments, we should consider a serious revision.

More Details: In Appendix A.3, we show a real-world example that recapitulates many features of this first chapter. We haven't included it in the main reading track to keep the high pace for the impatient audience. For those not in such a rush we recommend taking the time to read it and to see how nontrivial software evolves.

1.10 Exercises

1.10.1 Narrowing

Assign large values to different integer types with uniform initialization, i.e., using braces. For instance,

```
const unsigned c1{4000000000};
```

Does this compile on your machine? Try different values (including negative values) and different types and see what compiles on your platform. If possible, try other platforms by changing the machine or the compiler flags regarding the target platform.

1.10.2 Literals

Refactor your examples from Exercise 1.10.1 by using literal suffixes `u` and `l` as well as legal combinations thereof. If you like, you can change the variable/constant types to `auto`.

1.10.3 Operators

Program expressions that calculate the volumes and surfaces of different solid figures. Try to use as few parentheses as possible (in real life you may again apply as many as you want). Attempt to reuse common partial expressions by storing their results in intermediate variables.

1.10.4 Branching

Write a program that reads three numbers as `double` and prints the middle one. Optional: try expressing the same with `?:`.

1.10.5 Loops

Find the zero of $f = \sin(5x) + \cos(x)$ between 0 and 1 by interval nesting. For a given interval, split it in the middle and see on which side the sign of f is changing, then continue with this interval. Stop the calculation when the interval is smaller than 10^{-12} (so, we need `double`) and print the center of that interval with 11 digits as an approximated result. Hint: The result should be approximately 0.785. C++11 introduces the function `signbit` that is helpful here. Try different kinds of loops and think about which one feels most appropriate in this context.

1.10.6 I/O

Refactor Exercise 1.10.3 by writing all input parameters used in the calculation and all results to a file. Use a new line for a new set of values. Pay attention to leave a space between two numbers. Write a second program that reads the values from the file and compares it with the original ones.

1.10.7 Arrays and Pointers

Make a small program that creates arrays on the stack (fixed-size arrays) and arrays on the heap (using allocation). Use `valgrind` (or some Visual Studio tool on Windows) to check what happens when you do not `delete` them correctly or you use the wrong `delete` (array vs. single entry).

1.10.8 Functions

Write functions that convert SI units like `meter2km` and/or convert between SI and old-fashioned Anglo-American units like `usgallon2liter`. Test your implementations with `assert`. Use an ε-environment to deal with floating-point rounding issues, i.e. that the magnitude of the difference between computed and expected values is smaller than some predefined ε.

Subject Index