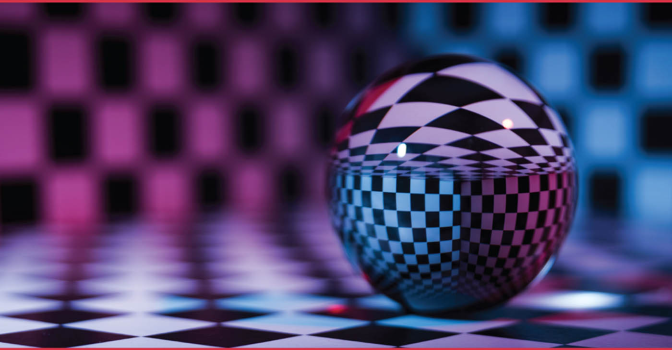
Discovering Modern C++

An Intensive Course for Scientists, Engineers, and Programmers

Peter Gottschling



C++ In-Depth Series   Bjarne Stroustrup

Discovering Modern C++

Discovering Modern C++

An Intensive Course for Scientists, Engineers, and Programmers

Peter Gottschling

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town

Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City

Sao Paulo • Sidney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Library of Congress Control Number: 2015955339

ISBN-13: 978-0-13-438358-3
ISBN-10: 0-13-438358-3
Text printed in the United States on recycled paper at Edwards Brothers Malloy in Ann Arbor, Michigan.
First printing, December 2015

To my parents, Helga and Hans-Werner

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 of Google, Amazon, and Facebook are built to a large extent in C++. In addition, a considerable fraction of the underlying technology is implemented in C++. In telecommunications, almost all landline and cellular phone connections are driven by C++ software. Most importantly, all the major transmission nodes in Germany are handled with C++, which means that peace in the author's family unconditionally relies on C++ software.

Even software written in other programming languages depends on C++ since the most popular compilers are realized in C++: Visual Studio, `clang`, and newer parts of Gnu and the Intel compiler. This is even more true for software running on Windows which is also implemented in C++ (as well as the Office package). The language is omnipresent; even your cell phone and your car certainly contain components driven by C++. Its inventor, Bjarne Stroustrup, set up a web page with applications where most examples here come from.

In science and engineering, many high-quality software packages are implemented in C++. The strength of the language is manifested in particular when projects exceed a certain size and data structures are rather complex. No wonder that many—if not most—simulation software programs in science and engineering are realized today in C++: the leaders Abaqus, deal.II, FEniCS, OpenFOAM, to name only a few; likewise the leading CAD software CATIA. Even embedded systems are increasingly realized in C++ thanks to more powerful processors and improved compilers (in which not all modern language features and libraries can always be used). Finally, we do not know how many projects would be realized in C++ instead of C if they had been started later. For instance, the author's good friend Matt Knepley, who is coauthor of the very successful scientific library PETSc, admitted that he would program the library today in C++ if rewriting was affordable.

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 to understand the behavior of programs in other languages. In C++ you can write extremely efficient programs that can only be slightly out-performed 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 of 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 applicable in a wider range 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.

```
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);
}
```

```
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 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: to 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 left version makes this optimization difficult.

A longer example comparing old and new style 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—as everywhere:

- 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—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-size tasks efficiently. When projects grow sufficiently large, it becomes increasingly important that the compiler is stricter (e.g., assignments are rejected when the arguments 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 accomplish 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).

- 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 `light blue in typewriter font`.

Programs requiring C++3, C++11, or C++14 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.

⇒ `directory/source_code.cpp`

Except for very short code illustrations, all programming examples in this book were tested on at least one compiler. Indicated by an arrow, the paths of the complete programs are given at the beginning of the paragraph or section in which the contained code snippets are discussed.

All programs are available on GitHub in the public repository `https://github.com/petergottschling/discovering_modern_cpp` and can thus be cloned by:

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

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

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 Mulanasky a great debt for contributing to 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 and gave many suggestions toward standard compliance and comprehensibility.

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 (not to forget) for creating C++. All these proofreaders pushed me hard to update the old lecture material with C++11 and C++14 features as far as possible.

In addition, I thank Karsten Ahnert for his recommendations and Markus Abel for helping me to get rid of the Preface's verbosity.

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 over 3 years, and I appreciate the constructive feedback from the students from this course. Likewise, I am grateful to the attendees of my C++ training.

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

Elizabeth Ryan deserves credit for managing the entire production process while having patience for all my special requests.

Last but not least, I wholeheartedly thank my family—my wife, Yasmine, and my children Yanis, Anissa, Vincent, and Daniely—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 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, vice-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 up to a bachelor-like level and finished the former with a PhD. After an odyssey through academic institutions, he founded his own company, SimuNova, and returned to his beloved hometown of Leipzig, just in time for its 1000th anniversary. He is married and has four children.

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 it 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 manuals `http://www.cplusplus.com/` and `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 prints 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.
- `std::` denotes that the type or function is used from the standard *Namespace*. Namespaces help us to organize our names and to deal with naming conflicts; see §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 a little bit with it, 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` (overview in §7.2.2.1) that provides (in recent versions) 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 have a few milliseconds to read the output before the console closes. To extend the reading phase to a 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 "Express" 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 "Express" 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, to show the in-code documentation, to 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` 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 that comes with some unique features such as a miniaturized colored page view as scroll bar.

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.

1.2 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;
```

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.

```
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.

Back to the variables! Their basic types—also called *Intrinsic Types*—are given in Table 1–1.

The first five types are integer numbers of non-decreasing 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). `signed` and `unsigned` can be considered adjectives for the nouns `short`, `int`, et cetera with `int` as the default noun when the adjective only is declared.

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`) in 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. We can also represent any letter whose code fits into 8 bits. It can be even mixed; 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.

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

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 80 bits for `long double`.

In the following section, we show operations that are often applied to integer and float types. In contrast to other languages like Python, where `'` and `"` are used for both characters and strings, C++ distinguishes between the two of them. The C++ compiler considers `'a'` as the character "a" (it has type `char`) and `"a"` is the string containing "a" and a binary 0 as termination (i.e., its type is `char[2]`). If you are used to Python, please pay attention to this.

Advice

Declare variables as late as possible, usually right before using them the first time and whenever possible 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 can deduce the type of a variable for us, e.g.: C++11

```cpp
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.

1.2.1 Constants

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

```cpp
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.

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).

1.2.2 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 number of digits. 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, three 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).[2] 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
```

If the previous three paragraphs were too brief for your taste, there is a more detailed version in Section A.2.1.

2. Mixed arithmetic is implementable, however, as demonstrated at [18].

Non-decimal 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 introduces binary literals which are prefixed by `0b` or `0B`:

C++14

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

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

C++14

```
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;
```

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.";
```

For more details on literals, see for instance [43, §6.2].

1.2.3 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 in Section 2.3.4. Values in braces cannot be narrowed:

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

Now, the compiler will check if the variable `l` can hold the value on the target architecture.

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 or constant (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.

1.2.4 Scopes

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

1.2.4.1 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. Believe us. Global constants like

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

are fine because they cannot cause side effects.

1.2.4.2 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= 3.14159265358979323846264338327950288419716939;
    std::cout ≪ "pi is " ≪ pi ≪ ".\n";
}
```

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

```
int main ()
{
    {
        const double pi= 3.14159265358979323846264338327950288419716939;
    }
    std::cout ≪ "pi is " ≪ pi ≪ ".\n"; // ERROR: pi is out of scope
}
```

In this example, the definition of π 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*.

1.2.4.3 Hiding

When a variable with the same name exists in nested scopes, then only one variable is visible. The variable in the inner scope hides the homonymous variables in the outer scopes. 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.[3] 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 till 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.2.2.

1.3 Operators

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

- Computational:
  - Arithmetic: `++`, `+`, `*`, `%`, ...

3. 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.

  - 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 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 one or multiple arguments in an expression are our types.

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 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.

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* (see Definition C–1

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`

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, then 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, prefix notations are prioritized over postfix 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`. 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 Section 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.

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.

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.

Table 1–3: Boolean Operators

Operation	Expression
Not	`!b`
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`

Please note that comparisons cannot be chained like this:

```
bool in_bound= min <= x <= y <= max;      // 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 quite 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 all 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. In most cases, 0s are moved in except for negative `signed` values in a right shift where it is implementation-defined. The bitwise AND can be used to test a specific bit of a value. Bitwise inclusive OR can set a bit and exclusive OR flip it. These operations are more important in system programming than scientific applications. As algorithmic entertainment, we will use them in §3.6.1.

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, assignments have a modifiable variable on the left-hand side. While other languages use asymmetric symbols (e.g., `:=` in Pascal), the author uses an asymmetric spacing in C++.

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 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.

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 sub-expression 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 sub-expression:

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

The result of the expression is 65.1 and the computation of the first sub-expression is entirely irrelevant. The sub-expressions can contain the comma operator as well so that arbitrarily long sequences can be defined. With the help of the comma operator, one can

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`

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; see Section 1.8.2.

1.3.7 Access Operators

C++ provides several operators for accessing sub-structures, for referring—i.e., taking the address of a variable—and dereferencing—i.e., accessing the memory referred to by an address. Discussing these operators before talking about pointers and classes makes no sense. We thus postpone their description to the sections given in Table 1–6.

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. The available operators are listed in Table 1–7.

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
Member dereference	`x.*q`	§2.2.3
Dereferred member dereference	`p->*q`	§2.2.3

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	`alignof(x)`
Alignment of type	`alignof(t)`

Note that the `sizeof` operator when used on an expression is the only one that is applicable without parentheses. `alignof` is introduced in C++11; all others exist since 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 (may be added in C++17);
`.*`	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 (wait till 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.

1.3.12 Avoid Side Effects!

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

—Unknown[4]

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

4. 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.

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*	
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*			
lvalue = *expr*	*lvalue* @= *expr*		
throw *expr*			
expr , *expr*			

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.

In the C standard library, there is a function to copy a string (`strcpy`). 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 is somehow. However, 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 sub-expressions, 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++;
```

Looks like no problem. But it is: the behavior of this expression is 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]`.

The last example 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 for the digital natives 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:

- Testing: `*src`
- Modification: `*tgt= *src;`
- 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 non-zero 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 to 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.

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 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 being useless (most likely). 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 section.

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)
    int absx= -x;
else
    int absx= x;
cout ≪ "|x| is " ≪ absx ≪ "\n"; // absx already 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 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.2.3. Finally, we give you the following:

Advice
Although spaces do not affect the 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 is not indented as expected, something is most likely not nested as intended.

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 sub-expression (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 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 [38].

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. An advanced use of `switch` is found in Appendix A.2.4.

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 a certain 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      $x_i = \begin{cases} 3\,x_{i-1} + 1 & \text{if } x_{i-1} \text{ is odd} \\ x_{i-1}/2 & \text{if } x_{i-1} \text{ is even} \end{cases}$
```

As long as 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 when there is only one statement.

C++ also offers a `do`-`while`-loop. In this case, 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 one time—even with an extremely small value for `eps` in our example.

1.4.4.2 `for`-Loop

The most common loop in C++ is the `for`-loop. As a simple example, we add two vectors[5] 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:

- The initialization;
- A *Continuation* criterion; and
- A step operation.

The example above is a typical `for`-loop. In the initialization, we typically declare a new variable and initialize it with 0—this is the start index of most indexed data structures. The condition usually tests whether the loop index is smaller than a certain size and the last operation typically 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

5. Later we will introduce true 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:

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

up to the tenth term:

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

Here it was simpler to compute term 0 separately and start the loop with term 1. We also used less-equal to assure 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= xyz.begin(), end= xyz.end(); i < end; ++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 the next section. We already mentioned that a loop index is typically incremented in the third sub-expression 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 wanted 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 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 new 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 out the primes from the array separated by spaces.

1.4.4.4 Loop Control

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

- `break` and
- `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 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 as it 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.

The general form of a C++ function is

```
[inline] return_type function_name (argument_list)
{
    body of the function
}
```

In this section, we discuss these components in more detail.

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++` within the `increment` function only increments a local copy of `i` 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
```

since we could not compute `(i + 9)++` anyway. In order to call such a function with some temporary value, we need to store it first in a variable and pass this variable to the function.

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 that 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.[6]
- 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 of an assignment is also a non-`const` reference).

In contrast to mutable[7] 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 Defaults

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) { ... }
```

6. Assuming the argument is properly copied. User types with broken copy implementations can undermine the integrity of the passed-in data.

7. 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.

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 argument list. In other words, after an argument with a default value we cannot have one without.

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 examples before, 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 Amounts of Data

Functions that compute new values of large data structures are more difficult. For the details, we will put you off till 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 before-mentioned 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` does not need to return something:

```
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`.

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 `double` 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 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. These conversions can be declared `explicit` and are then only applied when a conversion is explicitly requested but not for matching function arguments.

⇒ `c++11/overload_testing.cpp`

More formally phrased, function overloads must differ in their *Signature*. The signature consists in C++ 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) {}
void f(int& x) {}
void f(const int& x) {}
```

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

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

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

All three function calls are ambiguous because the best matches are in every case the first overload with the value argument and one of the reference-argument overloads respectively. Mixing overloads of reference and value arguments almost always fails. Thus, when one overload has a reference-qualified argument, then 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 (`argc[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 which 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 standard-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.2.5.

1.6 Error Handling

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

—John F. Kennedy

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` then 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

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

The fact is when our result is less than zero, 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` include `-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 possible), 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. In the case that there are so many properties that the actual functionality gets cluttered by the tests, one can outsource the tests into 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 to 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 more work than the actual task. In such cases, it is usually more efficient to try to accomplish the task and check for *Exceptions* along the way.

1.6.2.1 Motivation

Before illustrating the old-style error handling, we introduce our anti-hero Herbert[8] who is an ingenious mathematician and considers programming a necessary evil for demonstrating how magnificently his algorithms work. He learned to program like a real man and 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, ...)
{
    fstream f(fname);
    if (!f.is_open())
        return 1;
        ...
    return 0;
}
```

So, we 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 what happens when the caller simply ignores our return code? Nothing! The program keeps going and might crash later on absurd data or even worse 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 point across to programming dinosaurs like Herbert might not convince them: “Not only are you dumb enough to pass in a non-existing file to my perfectly implemented function, then you do not even check the return code. You do everything wrong, not me.”

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.

1.6.2.2 Throwing

The better approach is to `throw` an exception:

```
matrix read_matrix_file(const char* fname, ...)
{
    fstream f(fname);
```

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

```
        if (!f.is_open())
            throw "Cannot open file.";
        ...
    }
```

In this version, we throw an exception. The calling application is now obliged to react on it—otherwise the program crashes.

The advantage of exception handling over error codes is that we only need to bother with a problem where we can handle it. For instance, in the function that called read_matrix_file it might not be possible to deal with a non-existing file. In this case, the code is implemented as there is no exception thrown. So, we do not need to obfuscate our program with returning error codes. In the case of an exception, it is passed up to the appropriate exception handling. In our scenario, this handling might be contained in the GUI where a new file is requested from the user. Thus, exceptions lead at the same time to more readable sources and more reliable error handling.

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

```
struct cannot_open_file {};

void read_matrix_file(const char* fname, ...)
{
    fstream f(fname);
    if (!f.is_open())
        throw cannot_open_file{};
    ...
}
```

Here, we introduced our own exception type. In Chapter 2, we will explain in detail how classes can be defined. In the example above, we defined an empty class that only requires opening and closing brackets followed by a semicolon. Larger projects usually establish an entire hierarchy of exception types that are often 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:

```
try {
    ...
} catch (e1_type& e1)
{ ...
} catch (e2_type& e2) { ... }
```

Wherever we expect a problem that we can solve (or at least do something about), we open a try-block. After the closing braces, we can catch exceptions and start a rescue depending on the type of the exception and possibly on its value. It is recommended to catch exceptions by reference [45, Topic 73], especially when polymorphic types (Definition 6–1 in §6.1.3) are involved. When an exception is thrown, the first catch-block with a matching type is executed. Further catch-blocks of the same type (or sub-types; §6.1.1) are ignored. A catch-block with an ellipsis, i.e., three dots literally, catches all exceptions:

```
try {    ...
} catch (e1_type& e1) { ... }
  catch (e2_type& e2) { ... }
  catch (...) { // deal with all other exceptions
}
```

Obviously, the catch-all handler should be the last one.

If nothing else, we can catch the exception to provide an informative error message before terminating the program:

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

In our case, we are already in the main function and there is nothing else on the call stack to catch our exception. For rethrowing the current one, there exists a shorter notation:

```
} catch (cannot_open_file&) {
    ...
    throw;
}
```

This shortcut is preferred since it is less error-prone and shows more clearly that we rethrow the original exception. Ignoring an exception is easily implemented by an empty block:

```
} catch (cannot_open_file&) {} // File is rubbish, keep going
```

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

```
bool keep_trying= true;
do {
        char fname[80]; // std::string is better
        cout ≪ "Please enter the file name: ";
        cin ≫ fname;
        try {
            A= read_matrix_file(fname);
            ...
            keep_trying= false;
        } catch (cannot_open_file& e) {
            cout ≪ "Could not open the file. Try another one!\n";
        } catch (...)
            cout ≪ "Something is fishy here. Try another file!\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`.

A great advantage of exceptions is that issues that cannot be handled in the context where they are detected can be postponed for later. 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.

1.6.2.4 Who Throws?

C++11

Already C++03 allowed specifying which types of exceptions can be thrown from a function. Without going into details, these specifications turned out to be not very useful and are deprecated now.

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.2.6 and leave it to you when you read it.

C++11

1.6.3 Static Assertions

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. An example would not make sense at this point and we postpone it till Section 5.2.5.

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

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

```
int age;
std::cin ≫ age;
```

`std::cin` 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 myfile;
    square_file.open("squares.txt");
    for (int i= 0; i < 10; ++i)
        square_file ≪ i ≪ "ˆ2 = " i*i ≪ std::endl;
    square_file.close();
}
```

This code creates a file named `squares.txt` (or overwrites it if it already exists) and writes a sentence 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 file name 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,[9] in this case at the end of the `main` function. The short version of the previous 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 ≪ std::endl;
}
```

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[10] from `istream`, `ostream`, or `iostream` and provides implementations for the functions of those classes. For instance, Boost.Asio offers streams for TCP/IP and Boost.IOStream as 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 ≪ std::endl;
}

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

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

9. Thanks to the powerful technique named RAII, which we will discuss in Section 2.4.2.1.
10. 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`.

```
    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
```

In Section 4.3.1, we will show how the precision can be adjusted to the type's representable number of digits.

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. Some less frequently used format options can only be set this way, e.g., whether the sign is shown for positive values as well. 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.

1.7.6 Dealing with I/O Errors

To make one thing clear from the beginning: I/O in C++ is not fail-safe (let alone idiot-proof). 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 non-existing file and the program goes on. 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.

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 file names 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, give a new file name: ";
            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 example above, 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 exceptions that might be thrown.

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 appropriate variable type, e.g., `int` when setting `i1`. It stops at the first character that cannot be part of the value, first at the `.` 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 `o1` 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 utter attention. Numbers must be separated properly (e.g., by spaces) and read with the same type as they were written. When the output contains branches such 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.2.7 and A.2.8, respectively. You can also read this later when you need it.

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., `gcc`) 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.

The list initialization in C++11 cannot be narrowed any further. This will rarely make a difference in practice. For instance, the following:

C++11

```
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 first function parameter whereas array parameters don't contain size information.[11] 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 efforts 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:

- Indices are not checked before accessing an array, and we can find ourselves outside the array and 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.

- The size of the array must be known at compile time.[12] For instance, we have an array stored to a file and need to read it back into memory:

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

  This does not work because the size needs to be known during compilation.

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

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

12. 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 postponed as not all subtleties were entirely clarified.

1.8.2 Pointers

A pointer is a variable that contains a memory address. This address can be that of another variable that we can get with 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 previous 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 not needed anymore. This is done by

```
delete[] v;
```

Since arrays as function parameters are treated internally as pointers, the `vector_add` function from page 47 works with pointers as well:

```
int main (int argc, char* argv[])
{
    double *x= new double[3], *y= new double[3], *sum= new double[3];
    for (unsigned i= 0; i < 3; ++i)
        x[i]= i+2, y[i]= 4-2*i;
    vector_add(3, 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 previous 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 will handle the deallocation incorrectly and most likely crash at this point. 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.

C++11 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

```
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 it cannot be confused with other types and is self-explanatory, it is preferred over the other notations. The initialization with an empty braced list also sets a `nullptr`.

The biggest danger of pointers is *Memory Leaks*. For instance, our array `y` became too small and we want to assign a new array:

```
int* y= new int[15];
```

We can now use more space in `y`. Nice. 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 too. 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 our software. Fortunately, there are tools to help you to find memory leaks, as demonstrated in Section B.3.

The demonstrated issues with pointers are not intended as 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 utter care to avoid all the really severe problems mentioned above.

There are three strategies to minimize pointer-related errors:

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.

Encapsulate: dynamic memory management in classes. Then we have to deal with it only once per class.[13] When all memory allocated by an object is released when the object is destroyed, then 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 Allocation 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 to prevent than to fix.[14] 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.

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

Pointers serve two purposes:

- Referring to objects; and
- 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*.

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 should not be used anymore. 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.

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

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

C++11

1.8.3.1 Unique Pointer

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;
    ...
}
```

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

The destructor of pointer `dd` will try to delete `d`.

Unique pointers cannot be assigned to other pointer types or implicitly converted. For referring to the pointer's data in a raw pointer, we can use the member function `get`:

```
    double* raw_dp= dp.get();
```

It cannot even be assigned to another unique pointer:

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

It can only be moved:

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

We will discuss move semantics in Section 2.3.5. Right now let us just say this much: whereas a copy duplicates the data, a *Move* transfers the data from the source to the target. 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[15] 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 our own *Deleter*; for details see [26, §5.2.5f], [43, §34.3.1], or an online reference (e.g., `cppreference.com`).

1.8.3.2 Shared Pointer

C++11

As its name indicates, a `shared_ptr` manages memory that is used in common by 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= p2;
    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 the example, we allocated memory for two `double` values: in `p1` and in `p2`. The pointer `p2` is copied into `p3` so that both point to the same memory as illustrated in Figure 1–1.

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

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

When the function returns, the pointers are destroyed and the memory referred to by `p1` is released (without ever being used). The second allocated memory block still exists since `p` from the `main` function is still referring to it.

15. Specialization will be discussed in §3.6.1 and §3.6.3.

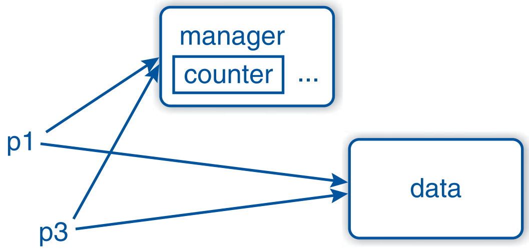


Figure 1–1: Shared pointer in memory

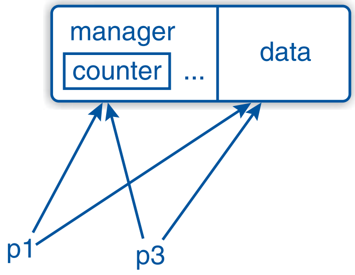


Figure 1–2: Shared pointer in memory after `make_shared`

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

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

Then the management and business data are stored together in memory—as shown in Figure 1–2—and the memory caching is more efficient. Since `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 [26, §5.2], [43, §34.3.2], or an online reference.

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: [26, §5.2.2], [43, §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 << "j = " << j << '\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 sub-object. Whenever we define a reference, we must directly declare what it is referring to (other than pointers). It is not possible to refer to another variable later.

So far, that does not sound extremely useful. References are extremely 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, the new standard 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 §4.4.2. 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.[16] 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.

In short, references are not fail-safe but are much less error-prone than pointers. Pointers should be only 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.

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) // DO NOT!
{
    double s= d * d;
    return s;
}
```

16. 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	✓	

Here, our function result refers the local variable `s` which does not exist anymore. 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. Actually, such hidden errors are even worse than the obvious ones because they can ruin our program only under certain conditions and then they are very hard to find.

Such references are called *Stale References*. Good compilers will warn us when we are referring to a local variable. Sadly enough, we have seen such examples in web tutorials.

The same applies to pointers:

```
double* square_ptr(double d) // DO NOT!
{
    double s= d * d;
    return &s;
}
```

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

Returning references or pointers can be correct in member functions when member data is referred to; see Section 2.6.

Advice
Only return pointers and references to dynamically allocated data, data that existed before the function was called, or static data.

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.

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.

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

```
    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: The array size can be deduced with templates, which we leave as an exercise for later (see §3.11.9).

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 previous one where the target vector is passed in as a reference. We will later discuss the possibilities of optimization: both on the compiler and on the user side. 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 non-scalar 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.

In Section A.2.9, we make some comments on *Garbage Collection* which is essentially saying that we can live well enough without it.

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 to 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 first sub-section. In this light, we will present the macro mechanism and other language features.

First of all, we want to discuss briefly 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 in 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 such 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 expression you want while the implementation under the hood squeezes out the maximal performance.

Another frequent usage of comments is to let code fractions disappear temporarily to experiment with alternative implementations, e.g.:

```
for ( ... ) {
    // int x78zy97= yo6954fq, y89haf= q6843, ...
    int x78zy98= yo6953fq, y89haf= q6842, ...
    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. Almost all programmers run into this trap: 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 possibility to deactivate multiple lines conveniently is by using the appropriate function of IDEs and language-aware editors.

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 use those macros. Visual Studio defines—even today!!!—`min` and `max` as macros, and we strongly advise you to disable this by compiling with `/DNO_MIN_MAX`. 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 few examples in Appendix A.2.10 with some tips for how to deal with them. Feel free to postpone the reading until you run into some issue.

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

1.9.2.2 Inclusion

To keep the language C 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`, `/usr/local/include`, and so on. 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 file name 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.[17] This is equivalent to quoting with angle brackets and adding the current directory to the search path. Some people argue that angle brackets should only be used for system headers and user headers should use double quotes.

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/more_functions.h>
```

The slashes are portable and also work under Windows despite the fact that sub-directories are denoted by backslashes there.

Include guards: Frequently used header files may be included multiple times in one translation unit 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
```

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

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 utter 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 file name. It can also contain project-relative paths or namespaces (§3.2.1). It is 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 before-mentioned information or 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);
...
```

This pragma is not part of the standard but all major compilers support it today. By using the pragma, it becomes the compiler's responsibility to avoid double inclusions.

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 non-portable libraries. Say we have a library only available on Windows, more precisely only with Visual Studio. 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, move semantics (§2.3.5):

```
#ifdef MY_LIBRARY_WITH_MOVE_SEMANTICS
   ... make something efficient with move
#else
   ... make something less efficient but portable
#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. Conditional compilation is quite powerful but it has its 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 a 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 fix it. Seriously.
#enif
```

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

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

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

Recapitulating this chapter, we illustrate the fundamental features of C++ in Appendix A.3. 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 non-trivial software evolves.

1.10 Exercises

1.10.1 Age

Write a program that asks input from the keyboard and prints the result on the screen and writes it to a file. The question is: "What is your age?"

1.10.2 Arrays and Pointers

1. Write the following declarations: pointer to a character, array of 10 integers, pointer to an array of 10 integers, pointer to an array of character strings, pointer to pointer to a character, integer constant, pointer to an integer constant, constant pointer to an integer. Initialize all these objects.
2. Make a small program that creates arrays on the stack (fixed-size arrays) and arrays on the heap (using allocation). Use `valgrind` to check what happens when you do not `delete` them correctly.

1.10.3 Read the Header of a Matrix Market File

The Matrix Market data format is used to store dense and sparse matrices in ASCII format. The header contains some information about the type and the size of the matrix. For a sparse matrix, the data is stored in three columns. The first column is the row number, the second column the column number, and the third column the numerical value. When the value type of the matrix is complex, a fourth column is added for the imaginary part.

An example of a Matrix Market file is

```
%%MatrixMarket matrix coordinate real general
%
% ATHENS course matrix
%
        2025            2025          100015
           1               1    .9273558001498543E-01
           1               2    .3545880644900583E-01
...................
```

The first line that does not start with `%` contains the number of rows, the number of columns, and the number of non-zero elements on the sparse matrix.

Use `fstream` to read the header of a Matrix Market file and print the number of rows and columns, and the number of non-zeroes on the screen.

Index