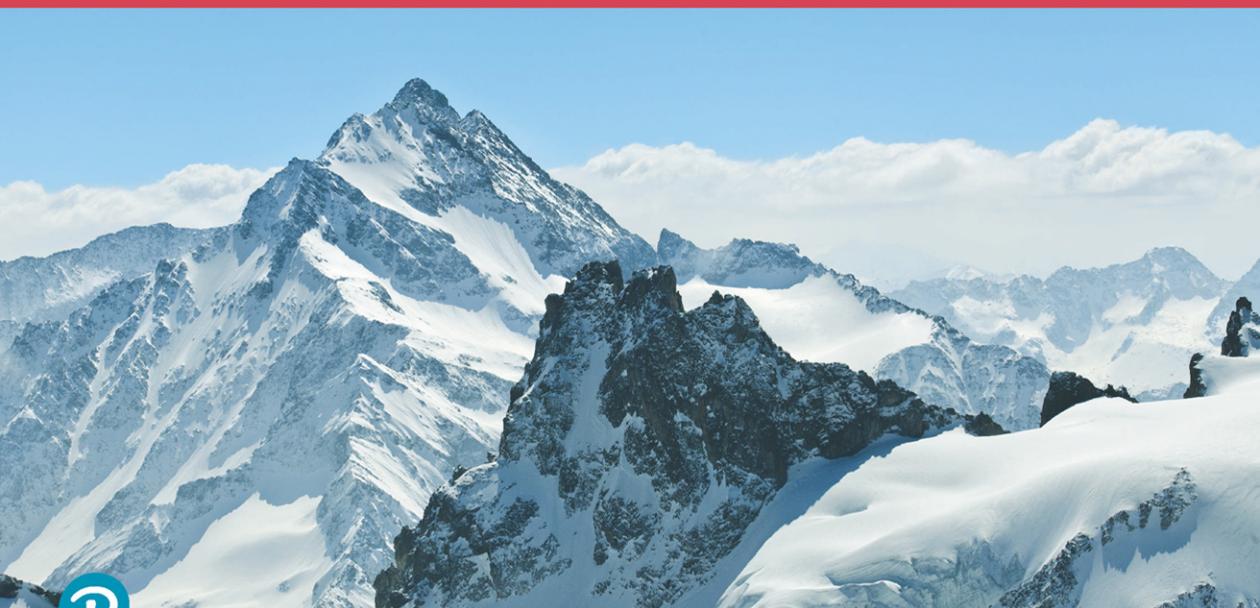




A Tour of C++

Second Edition

Bjarne Stroustrup



C++ In-Depth Series Bjarne Stroustrup

FREE SAMPLE CHAPTER

SHARE WITH OTHERS



A Tour of C++

Second Edition

A Tour of C++

Second Edition

Bjarne Stroustrup

◆ Addison-Wesley

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

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2018941627

Copyright © 2018 by Pearson Education, Inc.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearsoned.com/permissions/.

This book was typeset in Times and Helvetica by the author.

ISBN-13: 978-0-13-499783-4

ISBN-10: 0-13-499783-2

First printing, June 2018

1 18

Contents

Preface	xi
1 The Basics	1
1.1 Introduction	1
1.2 Programs	2
1.3 Functions	4
1.4 Types, Variables, and Arithmetic	5
1.5 Scope and Lifetime	9
1.6 Constants	9
1.7 Pointers, Arrays, and References	11
1.8 Tests	14
1.9 Mapping to Hardware	16
1.10 Advice	18
2 User-Defined Types	21
2.1 Introduction	21
2.2 Structures	22
2.3 Classes	23
2.4 Unions	25
2.5 Enumerations	26
2.6 Advice	27

3 Modularity	29
3.1 Introduction	29
3.2 Separate Compilation	30
3.3 Modules (C++20)	32
3.4 Namespaces	34
3.5 Error Handling	35
3.6 Function Arguments and Return Values	36
3.7 Advice	46
4 Classes	47
4.1 Introduction	47
4.2 Concrete Types	48
4.3 Abstract Types	54
4.4 Virtual Functions	56
4.5 Class Hierarchies	57
4.7 Advice	63
5 Essential Operations	65
5.1 Introduction	65
5.2 Copy and Move	52
5.3 Resource Management	72
5.4 Conventional Operations	74
5.5 Advice	77
6 Templates	79
6.1 Introduction	79
6.2 Parameterized Types	79
6.3 Parameterized Operations	84
6.4 Template Mechanisms	89
6.5 Advice	92
7 Concepts and Generic Programming	93
7.1 Introduction	93
7.2 Concepts	94
7.3 Generic Programming	98
7.4 Variadic Templates	100
7.5 Template Compilation Model	104
7.6 Advice	104

8 Library Overview	107
8.1 Introduction	107
8.2 Standard-Library Components	108
8.3 Standard-Library Headers and Namespace	109
8.4 Advice	110
9 Strings and Regular Expressions	111
9.1 Introduction	111
9.2 Strings	111
9.3 String Views	114
9.4 Regular Expressions	116
9.5 Advice	122
10 Input and Output	123
10.1 Introduction	123
10.2 Output	123
10.3 Input	125
10.4 I/O State	127
10.5 I/O of User-Defined Types	128
10.6 Formatting	129
10.7 File Streams	130
10.8 String Streams	130
10.9 C-style I/O	131
10.10 File System	132
10.11 Advice	136
11 Containers	137
11.1 Introduction	137
11.2 vector	138
11.3 list	142
11.4 map	144
11.5 unordered_map	144
11.6 Container Overview	146
11.7 Advice	148
12 Algorithms	149
12.1 Introduction	149
12.2 Use of Iterators	150
12.3 Iterator Types	153

12.4 Stream Iterators 154
12.5 Predicates 155
12.6 Algorithm Overview 156
12.7 Concepts (C++20) 157
12.8 Container Algorithms 160
12.9 Parallel Algorithms 161
12.10 Advice 161

13 Utilities

163

13.1 Introduction 163
13.2 Resource Management 164
13.3 Range Checking: **span** 168
13.4 Specialized Containers 170
13.5 Alternatives 174
13.6 Allocators 178
13.7 Time 179
13.8 Function Adaption 180
13.9 Type Functions 181
13.10 Advice 185

14 Numerics

187

14.1 Introduction 187
14.2 Mathematical Functions 188
14.3 Numerical Algorithms 189
14.4 Complex Numbers 190
14.5 Random Numbers 191
14.6 Vector Arithmetic 192
14.7 Numeric Limits 193
14.8 Advice 193

15 Concurrency

195

15.1 Introduction 195
15.2 Tasks and **threads** 196
15.3 Passing Arguments 197
15.4 Returning Results 198
15.5 Sharing Data 199
15.6 Waiting for Events 200
15.7 Communicating Tasks 202
15.8 Advice 205

16 History and Compatibility **207**

- 16.1 History 207
- 16.2 C++ Feature Evolution 214
- 16.3 C/C++ Compatibility 218
- 16.4 Bibliography 222
- 16.5 Advice 225

Index **227**

This page intentionally left blank

Preface

*When you wish to instruct,
be brief.
– Cicero*

C++ feels like a new language. That is, I can express my ideas more clearly, more simply, and more directly today than I could in C++98. Furthermore, the resulting programs are better checked by the compiler and run faster.

This book gives an overview of C++ as defined by C++17, the current ISO C++ standard, and implemented by the major C++ suppliers. In addition, it mentions concepts and modules, as defined in ISO Technical Specifications and in current use, but not scheduled for inclusion into the standard until C++20.

Like other modern languages, C++ is large and there are a large number of libraries needed for effective use. This thin book aims to give an experienced programmer an idea of what constitutes modern C++. It covers most major language features and the major standard-library components. This book can be read in just a few hours but, obviously, there is much more to writing good C++ than can be learned in a day. However, the aim here is not mastery, but to give an overview, to give key examples, and to help a programmer get started.

The assumption is that you have programmed before. If not, please consider reading a textbook, such as *Programming: Principles and Practice Using C++ (Second edition)* [Stroustrup,2014], before continuing here. Even if you have programmed before, the language you used or the applications you wrote may be very different from the style of C++ presented here.

Think of a sightseeing tour of a city, such as Copenhagen or New York. In just a few hours, you are given a quick peek at the major attractions, told a few background stories, and given some suggestions about what to do next. You do *not* know the city after such a tour. You do *not* understand all you have seen and heard. You do *not* know how to navigate the formal and informal rules that govern life in the city. To really know a city, you have to live in it, often for years. However, with a bit of luck, you will have gained a bit of an overview, a notion of what is special about the city, and ideas of what might be of interest to you. After the tour, the real exploration can begin.

This tour presents the major C++ language features as they support programming styles, such as object-oriented and generic programming. It does not attempt to provide a detailed, reference-manual, feature-by-feature view of the language. In the best textbook tradition, I try to explain a feature before I use it, but that is not always possible and not everybody reads the text strictly sequentially. So, the reader is encouraged to use the cross references and the index.

Similarly, this tour presents the standard libraries in terms of examples, rather than exhaustively. It does not describe libraries beyond those defined by the ISO standard. The reader can search out supporting material as needed. [Stroustrup,2013] and [Stroustrup,2014] are examples of such material, but there is an enormous amount of material (of varying quality) available on the Web, e.g., [Cplusplusreference]. For example, when I mention a standard-library function or class, its definition can easily be looked up, and by examining its documentation, many related facilities can be found.

This tour presents C++ as an integrated whole, rather than as a layer cake. Consequently, it does not identify language features as present in C, part of C++98, or new in C++11, C++14, or C++17. Such information can be found in Chapter 16 (History and Compatibility). I focus on fundamentals and try to be brief, but I have not completely resisted the temptation to overrepresent novel features. This also seems to satisfy the curiosity of many readers who already know some older version of C++.

A programming language reference manual or standard simply states what can be done, but programmers are often more interested in learning how to use the language well. This aspect is partly addressed in the selection of topics covered, partly in the text, and specifically in the advice sections. More advice about what constitutes *good modern* C++ can be found in the C++ Core Guidelines [Stroustrup,2015]. The core guidelines can be a good source for further exploration of the ideas presented in this book. You may note a remarkable similarity of the advice formulation and even the numbering of advice between the Core Guidelines and this book. One reason is that the first edition of *A Tour of C++* was a major source of the initial Core Guidelines.

Acknowledgments

Some of the material presented here is borrowed from TC++PL4 [Stroustrup,2013], so thanks to all who helped completing that book.

Thanks to all who help complete and correct the first edition of “A Tour of C++.”

Thanks to Morgan Stanley for giving me time to write this second edition. Thanks to the Columbia University Spring 2018 “Design Using C++” class for finding many a typo and bug in an early draft of this book and for making many constructive suggestions.

Thanks to Paul Anderson, Chuck Allison, Peter Gottschling, William Mons, Charles Wilson, and Sergey Zubkov for reviewing the book and suggesting many improvements.

Manhattan, New York

Bjarne Stroustrup

Modularity

Don't interrupt me while I'm interrupting.
– Winston S. Churchill

- Introduction
- Separate Compilation
- Modules
- Namespaces
- Error Handling
 - Exceptions; Invariants; Error-Handling Alternatives; Contracts; Static Assertions
- Function Arguments and Return Values
 - Argument Passing; Value Return; Structured Binding
- Advice

3.1 Introduction

A C++ program consists of many separately developed parts, such as functions (§1.2.1), user-defined types (Chapter 2), class hierarchies (§4.5), and templates (Chapter 6). The key to managing this is to clearly define the interactions among those parts. The first and most important step is to distinguish between the interface to a part and its implementation. At the language level, C++ represents interfaces by declarations. A *declaration* specifies all that's needed to use a function or a type. For example:

```
double sqrt(double);    // the square root function takes a double and returns a double
```

```
class Vector {  
public:  
    Vector(int s);  
    double& operator[](int i);  
    int size();
```

```
private:
    double* elem; // elem points to an array of sz doubles
    int sz;
};
```

The key point here is that the function bodies, the function *definitions*, are “elsewhere.” For this example, we might like for the representation of **Vector** to be “elsewhere” also, but we will deal with that later (abstract types; §4.3). The definition of **sqrt()** will look like this:

```
double sqrt(double d)           // definition of sqrt()
{
    // ... algorithm as found in math textbook ...
}
```

For **Vector**, we need to define all three member functions:

```
Vector::Vector(int s)           // definition of the constructor
    :elem(new double[s]), sz{s} // initialize members
{
}

double& Vector::operator[](int i) // definition of subscripting
{
    return elem[i];
}

int Vector::size()              // definition of size()
{
    return sz;
}
```

We must define **Vector**’s functions, but not **sqrt()** because it is part of the standard library. However, that makes no real difference: a library is simply “some other code we happen to use” written with the same language facilities we use.

There can be many declarations for an entity, such as a function, but only one definition.

3.2 Separate Compilation

C++ supports a notion of separate compilation where user code sees only declarations of the types and functions used. The definitions of those types and functions are in separate source files and are compiled separately. This can be used to organize a program into a set of semi-independent code fragments. Such separation can be used to minimize compilation times and to strictly enforce separation of logically distinct parts of a program (thus minimizing the chance of errors). A library is often a collection of separately compiled code fragments (e.g., functions).

Typically, we place the declarations that specify the interface to a module in a file with a name indicating its intended use. For example:

```

// Vector.h:

class Vector {
public:
    Vector(int s);
    double& operator[](int i);
    int size();
private:
    double* elem;    // elem points to an array of sz doubles
    int sz;
};

```

This declaration would be placed in a file **Vector.h**. Users then *include* that file, called a *header file*, to access that interface. For example:

```

// user.cpp:

#include "Vector.h"    // get Vector's interface
#include <cmath>       // get the standard-library math function interface including sqrt()

double sqrt_sum(Vector& v)
{
    double sum = 0;
    for (int i=0; i<v.size(); ++i)
        sum+=std::sqrt(v[i]);    // sum of square roots
    return sum;
}

```

To help the compiler ensure consistency, the **.cpp** file providing the implementation of **Vector** will also include the **.h** file providing its interface:

```

// Vector.cpp:

#include "Vector.h" // get Vector's interface

Vector::Vector(int s)
    :elem(new double[s]), sz{s}    // initialize members
{
}

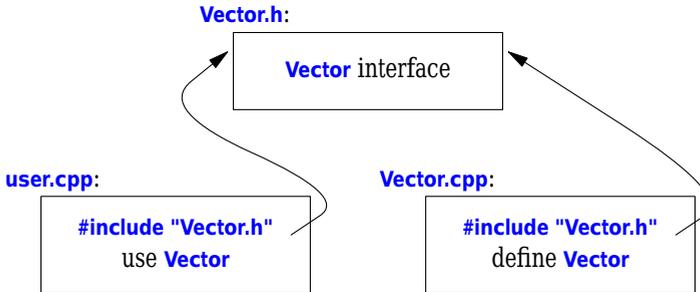
double& Vector::operator[](int i)
{
    return elem[i];
}

int Vector::size()
{
    return sz;
}

```

The code in **user.cpp** and **Vector.cpp** shares the **Vector** interface information presented in **Vector.h**,

but the two files are otherwise independent and can be separately compiled. Graphically, the program fragments can be represented like this:



Strictly speaking, using separate compilation isn't a language issue; it is an issue of how best to take advantage of a particular language implementation. However, it is of great practical importance. The best approach to program organization is to think of the program as a set of modules with well-defined dependencies, represent that modularity logically through language features, and then exploit the modularity physically through files for effective separate compilation.

A `.cpp` file that is compiled by itself (including the `h` files it `#includes`) is called a *translation unit*. A program can consist of many thousand translation units.

3.3 Modules (C++20)

The use of `#includes` is a very old, error-prone, and rather expensive way of composing programs out of parts. If you `#include header.h` in 101 translation units, the text of `header.h` will be processed by the compiler 101 times. If you `#include header1.h` before `header2.h` the declarations and macros in `header1.h` might affect the meaning of the code in `header2.h`. If instead you `#include header2.h` before `header1.h`, it is `header2.h` that might affect the code in `header1.h`. Obviously, this is not ideal, and in fact it has been a major source of cost and bugs since 1972 when this mechanism was first introduced into C.

We are finally about to get a better way of expressing physical modules in C++. The language feature, called `modules` is not yet ISO C++, but it is an ISO Technical Specification [ModulesTS]. Implementations are in use, so I risk recommending it here even though details are likely to change and it may be years before everybody can use it in production code. Old code, in this case code using `#include`, can “live” for a very long time because it can be costly and time consuming to update.

Consider how to express the `Vector` and `use()` example from §3.2 using `modules`:

```
// file Vector.cpp:
```

```
module; // this compilation will define a module
```

```
// ... here we put stuff that Vector might need for its implementation ...
```

```

export module Vector; // defining the module called "Vector"

export class Vector {
public:
    Vector(int s);
    double& operator[](int i);
    int size();
private:
    double* elem; // elem points to an array of sz doubles
    int sz;
};

Vector::Vector(int s)
    :elem{new double[s]}, sz{s} // initialize members
{
}

double& Vector::operator[](int i)
{
    return elem[i];
}

int Vector::size()
{
    return sz;
}

export int size(const Vector& v) { return v.size(); }

```

This defines a **module** called **Vector**, which exports the class **Vector**, all its member functions, and the non-member function **size**().

The way we use this **module** is to **import** it where we need it. For example:

// file user.cpp:

```

import Vector; // get Vector's interface
#include <cmath> // get the standard-library math function interface including sqrt()

double sqrt_sum(Vector& v)
{
    double sum = 0;
    for (int i=0; i!=v.size(); ++i)
        sum+=std::sqrt(v[i]); // sum of square roots
    return sum;
}

```

I could have **imported** the standard library mathematical functions also, but I used the old-fashioned **#include** just to show that you can mix old and new. Such mixing is essential for gradually upgrading older code from using **#include** to **import**.

The differences between headers and modules are not just syntactic.

- A module is compiled once only (rather than in each translation unit in which it is used).
- Two modules can be **imported** in either order without changing their meaning.
- If you import something into a module, users of your module do not implicitly gain access to (and are not bothered by) what you imported: **import** is not transitive.

The effects on maintainability and compile-time performance can be spectacular.

3.4 Namespaces

In addition to functions (§1.3), classes (§2.3), and enumerations (§2.5), C++ offers *namespaces* as a mechanism for expressing that some declarations belong together and that their names shouldn't clash with other names. For example, I might want to experiment with my own complex number type (§4.2.1, §14.4):

```
namespace My_code {
    class complex {
        // ...
    };

    complex sqrt(complex);
    // ...

    int main();
}

int My_code::main()
{
    complex z {1,2};
    auto z2 = sqrt(z);
    std::cout << '{' << z2.real() << ',' << z2.imag() << "}\n";
    // ...
}

int main()
{
    return My_code::main();
}
```

By putting my code into the namespace **My_code**, I make sure that my names do not conflict with the standard-library names in namespace **std** (§3.4). That precaution is wise, because the standard library does provide support for **complex** arithmetic (§4.2.1, §14.4).

The simplest way to access a name in another namespace is to qualify it with the namespace name (e.g., **std::cout** and **My_code::main**). The “real **main()**” is defined in the global namespace, that is, not local to a defined namespace, class, or function.

If repeatedly qualifying a name becomes tedious or distracting, we can bring the name into a scope with a **using**-declaration:

```

void my_code(vector<int>& x, vector<int>& y)
{
    using std::swap;           // use the standard-library swap
    // ...
    swap(x,y);                // std::swap()
    other::swap(x,y);         // some other swap()
    // ...
}

```

A **using**-declaration makes a name from a namespace usable as if it was declared in the scope in which it appears. After **using std::swap**, it is exactly as if **swap** had been declared in **my_code()**.

To gain access to all names in the standard-library namespace, we can use a **using**-directive:

```
using namespace std;
```

A **using**-directive makes unqualified names from the named namespace accessible from the scope in which we placed the directive. So after the **using**-directive for **std**, we can simply write **cout** rather than **std::cout**. By using a **using**-directive, we lose the ability to selectively use names from that namespace, so this facility should be used carefully, usually for a library that's pervasive in an application (e.g., **std**) or during a transition for an application that didn't use **namespaces**.

Namespaces are primarily used to organize larger program components, such as libraries. They simplify the composition of a program out of separately developed parts.

3.5 Error Handling

Error handling is a large and complex topic with concerns and ramifications that go far beyond language facilities into programming techniques and tools. However, C++ provides a few features to help. The major tool is the type system itself. Instead of painstakingly building up our applications from the built-in types (e.g., **char**, **int**, and **double**) and statements (e.g., **if**, **while**, and **for**), we build types (e.g., **string**, **map**, and **regex**) and algorithms (e.g., **sort()**, **find_if()**, and **draw_all()**) that are appropriate for our applications. Such higher-level constructs simplify our programming, limit our opportunities for mistakes (e.g., you are unlikely to try to apply a tree traversal to a dialog box), and increase the compiler's chances of catching errors. The majority of C++ language constructs are dedicated to the design and implementation of elegant and efficient abstractions (e.g., user-defined types and algorithms using them). One effect of such abstraction is that the point where a run-time error can be detected is separated from the point where it can be handled. As programs grow, and especially when libraries are used extensively, standards for handling errors become important. It is a good idea to articulate a strategy for error handling early on in the development of a program.

3.5.1 Exceptions

Consider again the **Vector** example. What *ought* to be done when we try to access an element that is out of range for the vector from §2.3?

- The writer of **Vector** doesn't know what the user would like to have done in this case (the writer of **Vector** typically doesn't even know in which program the vector will be running).

- The user of `Vector` cannot consistently detect the problem (if the user could, the out-of-range access wouldn't happen in the first place).

Assuming that out-of-range access is a kind of error that we want to recover from, the solution is for the `Vector` implementer to detect the attempted out-of-range access and tell the user about it. The user can then take appropriate action. For example, `Vector::operator[]()` can detect an attempted out-of-range access and throw an `out_of_range` exception:

```
double& Vector::operator[](int i)
{
    if (i<0 || size()<=i)
        throw out_of_range{"Vector::operator[]"};
    return elem[i];
}
```

The `throw` transfers control to a handler for exceptions of type `out_of_range` in some function that directly or indirectly called `Vector::operator[]()`. To do that, the implementation will *unwind* the function call stack as needed to get back to the context of that caller. That is, the exception handling mechanism will exit scopes and functions as needed to get back to a caller that has expressed interest in handling that kind of exception, invoking destructors (§4.2.2) along the way as needed. For example:

```
void f(Vector& v)
{
    // ...
    try { // exceptions here are handled by the handler defined below

        v[v.size()] = 7; // try to access beyond the end of v
    }
    catch (out_of_range& err) { // oops: out_of_range error
        // ... handle range error ...
        cerr << err.what() << '\n';
    }
    // ...
}
```

We put code for which we are interested in handling exceptions into a `try`-block. The attempted assignment to `v[v.size()]` will fail. Therefore, the `catch`-clause providing a handler for exceptions of type `out_of_range` will be entered. The `out_of_range` type is defined in the standard library (in `<stdexcept>`) and is in fact used by some standard-library container access functions.

I caught the exception by reference to avoid copying and used the `what()` function to print the error message put into it at the `throw`-point.

Use of the exception-handling mechanisms can make error handling simpler, more systematic, and more readable. To achieve that, don't overuse `try`-statements. The main technique for making error handling simple and systematic (called *Resource Acquisition Is Initialization; RAII*) is explained in §4.2.2. The basic idea behind RAII is for a constructor to acquire all resources necessary for a class to operate and have the destructor release all resources, thus making resource release guaranteed and implicit.

A function that should never throw an exception can be declared `noexcept`. For example:

```
void user(int sz) noexcept
{
    Vector v(sz);
    iota(&v[0], &v[sz], 1); // fill v with 1,2,3,4... (see §14.3)
    // ...
}
```

If all good intent and planning fails, so that `user()` still throws, `std::terminate()` is called to immediately terminate the program.

3.5.2 Invariants

The use of exceptions to signal out-of-range access is an example of a function checking its argument and refusing to act because a basic assumption, a *precondition*, didn't hold. Had we formally specified `Vector`'s subscript operator, we would have said something like "the index must be in the `[0: size())` range," and that was in fact what we tested in our `operator[]()`. The `[a:b)` notation specifies a half-open range, meaning that `a` is part of the range, but `b` is not. Whenever we define a function, we should consider what its preconditions are and consider whether to test them (§3.5.3). For most applications it is a good idea to test simple invariants; see also §3.5.4.

However, `operator[]()` operates on objects of type `Vector` and nothing it does makes any sense unless the members of `Vector` have "reasonable" values. In particular, we did say "`elem` points to an array of `sz` doubles" but we only said that in a comment. Such a statement of what is assumed to be true for a class is called a *class invariant*, or simply an *invariant*. It is the job of a constructor to establish the invariant for its class (so that the member functions can rely on it) and for the member functions to make sure that the invariant holds when they exit. Unfortunately, our `Vector` constructor only partially did its job. It properly initialized the `Vector` members, but it failed to check that the arguments passed to it made sense. Consider:

```
Vector v(-27);
```

This is likely to cause chaos.

Here is a more appropriate definition:

```
Vector::Vector(int s)
{
    if (s < 0)
        throw length_error("Vector constructor: negative size");
    elem = new double[s];
    sz = s;
}
```

I use the standard-library exception `length_error` to report a non-positive number of elements because some standard-library operations use that exception to report problems of this kind. If operator `new` can't find memory to allocate, it throws a `std::bad_alloc`. We can now write:

```

void test()
{
    try {
        Vector v(-27);
    }
    catch (std::length_error& err) {
        // handle negative size
    }
    catch (std::bad_alloc& err) {
        // handle memory exhaustion
    }
}

```

You can define your own classes to be used as exceptions and have them carry arbitrary information from a point where an error is detected to a point where it can be handled (§3.5.1).

Often, a function has no way of completing its assigned task after an exception is thrown. Then, “handling” an exception means doing some minimal local cleanup and rethrowing the exception. For example:

```

void test()
{
    try {
        Vector v(-27);
    }
    catch (std::length_error&) { // do something and rethrow
        cerr << "test failed: length error\n";
        throw; // rethrow
    }
    catch (std::bad_alloc&) { // Ouch! this program is not designed to handle memory exhaustion
        std::terminate(); // terminate the program
    }
}

```

In well-designed code **try**-blocks are rare. Avoid overuse by systematically using the RAII technique (§4.2.2, §5.3).

The notion of invariants is central to the design of classes, and preconditions serve a similar role in the design of functions. Invariants

- help us to understand precisely what we want
- force us to be specific; that gives us a better chance of getting our code correct (after debugging and testing).

The notion of invariants underlies C++’s notions of resource management supported by constructors (Chapter 4) and destructors (§4.2.2, §13.2).

3.5.3 Error-Handling Alternatives

Error handling is a major issue in all real-world software, so naturally there are a variety of approaches. If an error is detected and it cannot be handled locally in a function, the function must somehow communicate the problem to some caller. Throwing an exception is C++’s most general mechanism for that.

There are languages where exceptions are designed simply to provide an alternate mechanism for returning values. C++ is not such a language: exceptions are designed to be used to report failure to complete a given task. Exceptions are integrated with constructors and destructors to provide a coherent framework for error handling and resource management (§4.2.2, §5.3). Compilers are optimized to make returning a value much cheaper than throwing the same value as an exception.

Throwing an exception is not the only way of reporting an error that cannot be handled locally. A function can indicate that it cannot perform its allotted task by:

- throwing an exception
- somehow return a value indicating failure
- terminating the program (by invoking a function like `terminate()`, `exit()`, or `abort()`).

We return an error indicator (an “error code”) when:

- A failure is normal and expected. For example, it is quite normal for a request to open a file to fail (maybe there is no file of that name or maybe the file cannot be opened with the permissions requested).
- An immediate caller can reasonably be expected to handle the failure.

We throw an exception when:

- An error is so rare that a programmer is likely to forget to check for it. For example, when did you last check the return value of `printf()`?
- An error cannot be handled by an immediate caller. Instead, the error has to percolate back to an ultimate caller. For example, it is infeasible to have every function in an application reliably handle every allocation failure or network outage.
- New kinds of errors can be added in lower-modules of an application so that higher-level modules are not written to cope with such errors. For example, when a previously single-threaded application is modified to use multiple threads or resources are placed remotely to be accessed over a network.
- No suitable return path for error codes are available. For example, a constructor does not have a return value for a “caller” to check. In particular, constructors may be invoked for several local variables or in a partially constructed complex object so that clean-up based on error codes would be quite complicated.
- The return path of a function is made more complicated or expensive by a need to pass both a value and an error indicator back (e.g., a `pair`; §13.4.3), possibly leading to the use of out-parameters, non-local error-status indicators, or other workarounds.
- The error has to be transmitted up a call chain to an “ultimate caller.” Repeatedly checking an error-code would be tedious, expensive, and error-prone.
- The recovery from errors depends on the results of several function calls, leading to the need to maintain local state between calls and complicated control structures.
- The function that found the error was a callback (a function argument), so the immediate caller may not even know what function was called.
- An error implies that some “undo action” is needed.

We terminate when

- An error is of a kind from which we cannot recover. For example, for many – but not all – systems there is no reasonable way to recover from memory exhaustion.
- The system is one where error-handling is based on restarting a thread, process, or computer whenever a non-trivial error is detected.

One way to ensure termination is to add **noexcept** to a function so that a **throw** from anywhere in the function’s implementation will turn into a **terminate()**. Note that there are applications that can’t accept unconditional terminations, so alternatives must be used.

Unfortunately, these conditions are not always logically disjoint and easy to apply. The size and complexity of a program matters. Sometimes the tradeoffs change as an application evolves. Experience is required. When in doubt, prefer exceptions because their use scales better, and don’t require external tools to check that all errors are handled.

Don’t believe that all error codes or all exceptions are bad; there are clear uses for both. Furthermore, do not believe the myth that exception handling is slow; it is often faster than correct handling of complex or rare error conditions, and of repeated tests of error codes.

RAII (§4.2.2, §5.3) is essential for simple and efficient error-handling using exceptions. Code littered with **try**-blocks often simply reflects the worst aspects of error-handling strategies conceived for error codes.

3.5.4 Contracts

There is currently no general and standard way of writing optional run-time tests of invariants, pre-conditions, etc. A contract mechanism is proposed for C++20 [Garcia,2016] [Garcia,2018]. The aim is to support users who want to rely on testing to get programs right – running with extensive run-time checks – but then deploy code with minimal checks. This is popular in high-performance applications in organizations that rely on systematic and extensive checking.

For now, we have to rely on ad hoc mechanisms. For example, we could use a command-line macro to control a run-time check:

```
double& Vector::operator[](int i)
{
    if (RANGE_CHECK && (i<0 || size()<=i))
        throw out_of_range{"Vector::operator[]"};
    return elem[i];
}
```

The standard library offers the debug macro, **assert()**, to assert that a condition must hold at run time. For example:

```
void f(const char* p)
{
    assert(p!=nullptr); // p must not be the nullptr
    // ...
}
```

If the condition of an **assert()** fails in “debug mode,” the program terminates. If we are not in debug mode, the **assert()** is not checked. That’s pretty crude and inflexible, but often sufficient.

3.5.5 Static Assertions

Exceptions report errors found at run time. If an error can be found at compile time, it is usually preferable to do so. That’s what much of the type system and the facilities for specifying the interfaces to user-defined types are for. However, we can also perform simple checks on most

properties that are known at compile time and report failures to meet our expectations as compiler error messages. For example:

```
static_assert(4<=sizeof(int), "integers are too small"); // check integer size
```

This will write **integers are too small** if **4<=sizeof(int)** does not hold; that is, if an **int** on this system does not have at least 4 bytes. We call such statements of expectations *assertions*.

The **static_assert** mechanism can be used for anything that can be expressed in terms of constant expressions (§1.6). For example:

```
constexpr double C = 299792.458; // km/s

void f(double speed)
{
    constexpr double local_max = 160.0/(60*60); // 160 km/h == 160.0/(60*60) km/s

    static_assert(speed<C,"can't go that fast"); // error: speed must be a constant
    static_assert(local_max<C,"can't go that fast"); // OK

    // ...
}
```

In general, **static_assert(A,S)** prints **S** as a compiler error message if **A** is not **true**. If you don't want a specific message printed, leave out the **S** and the compiler will supply a default message:

```
static_assert(4<=sizeof(int)); // use default message
```

The default message is typically the source location of the **static_assert** plus a character representation of the asserted predicate.

The most important uses of **static_assert** come when we make assertions about types used as parameters in generic programming (§7.2, §13.9).

3.6 Function Arguments and Return Values

The primary and recommended way of passing information from one part of a program to another is through a function call. Information needed to perform a task is passed as arguments to a function and the results produced are passed back as return values. For example:

```
int sum(const vector<int>& v)
{
    int s = 0;
    for (const int i : v)
        s += i;
    return s;
}

vector fib = {1,2,3,5,8,13,21};

int x = sum(fib); // x becomes 53
```

There are other paths through which information can be passed between functions, such as global variables (§1.5), pointer and reference parameters (§3.6.1), and shared state in a class object (Chapter 4). Global variables are strongly discouraged as a known source of errors, and state should typically be shared only between functions jointly implementing a well-defined abstraction (e.g., member functions of a class; §2.3).

Given the importance of passing information to and from functions, it is not surprising that there are a variety of ways of doing it. Key concerns are:

- Is an object copied or shared?
- If an object is shared, is it mutable?
- Is an object moved, leaving an “empty object” behind (§5.2.2)?

The default behavior for both argument passing and value return is “copy” (§1.9), but some copies can implicitly be optimized to moves.

In the `sum()` example, the resulting `int` is copied out of `sum()` but it would be inefficient and pointless to copy the potentially very large `vector` into `sum()`, so the argument is passed by reference (indicated by the `&`; §1.7).

The `sum()` has no reason to modify its argument. This immutability is indicated by declaring the `vector` argument `const` (§1.6), so the `vector` is passed by `const`-reference.

3.6.1 Argument Passing

First consider how to get values into a function. By default we copy (“pass-by-value”) and if we want to refer to an object in the caller’s environment, we use a reference (“pass-by-reference”). For example:

```
void test(vector<int> v, vector<int>& rv)    // v is passed by value; rv is passed by reference
{
    v[1] = 99;    // modify v (a local variable)
    rv[2] = 66;  // modify whatever rv refers to
}

int main()
{
    vector fib = {1,2,3,5,8,13,21};
    test(fib, fib);
    cout << fib[1] << ' ' << fib[2] << '\n';    // prints 2 66
}
```

When we care about performance, we usually pass small values by-value and larger ones by-reference. Here “small” means “something that’s really cheap to copy.” Exactly what “small” means depends on machine architecture, but “the size of two or three pointers or less” is a good rule of thumb.

If we want to pass by reference for performance reasons but don’t need to modify the argument, we pass-by-`const`-reference as in the `sum()` example. This is by far the most common case in ordinary good code: it is fast and not error-prone.

It is not uncommon for a function argument to have a default value; that is, a value that is considered preferred or just the most common. We can specify such a default by a *default function argument*. For example:

```

void print(int value, int base =10); // print value in base "base"

print(x,16); // hexadecimal
print(x,60); // sexagesimal (Sumerian)
print(x); // use the default: decimal

```

This is a notationally simpler alternative to overloading:

```

void print(int value, int base); // print value in base "base"

void print(int value) // print value in base 10
{
    print(value,10);
}

```

3.6.2 Value Return

Once we have computed a result, we need to get it out of the function and back to the caller. Again, the default for value return is to copy and for small objects that's ideal. We return "by reference" only when we want to grant a caller access to something that is not local to the function. For example:

```

class Vector {
public:
    // ...
    double& operator[](int i) { return elem[i]; } // return reference to ith element
private:
    double* elem; // elem points to an array of sz
    // ...
};

```

The *i*th element of a **Vector** exists independently of the call of the subscript operator, so we can return a reference to it.

On the other hand, a local variable disappears when the function returns, so we should not return a pointer or reference to it:

```

int& bad()
{
    int x;
    // ...
    return x; // bad: return a reference to the local variable x
}

```

Fortunately, all major C++ compilers will catch the obvious error in **bad()**.

Returning a reference or a value of a "small" type is efficient, but how do we pass large amounts of information out of a function? Consider:

```

Matrix operator+(const Matrix& x, const Matrix& y)
{
    Matrix res;
    // ... for all res[i,j], res[i,j] = x[i,j]+y[i,j] ...
    return res;
}

Matrix m1, m2;
// ...
Matrix m3 = m1+m2;    // no copy

```

A **Matrix** may be very large and expensive to copy even on modern hardware. So we don't copy, we give **Matrix** a move constructor (§5.2.2) and very cheaply move the **Matrix** out of **operator+**(**)**. We do not need to regress to using manual memory management:

```

Matrix* add(const Matrix& x, const Matrix& y)    // complicated and error-prone 20th century style
{
    Matrix* p = new Matrix;
    // ... for all *p[i,j], *p[i,j] = x[i,j]+y[i,j] ...
    return p;
}

Matrix m1, m2;
// ...
Matrix* m3 = add(m1,m2);    // just copy a pointer
// ...
delete m3;                // easily forgotten

```

Unfortunately, returning large objects by returning a pointer to it is common in older code and a major source of hard-to-find errors. Don't write such code. Note that **operator+**(**)** is as efficient as **add()**, but far easier to define, easier to use, and less error-prone.

If a function cannot perform its required task, it can throw an exception (§3.5.1). This can help avoid code from being littered with error-code tests for "exceptional problems."

The return type of a function can be deduced from its return value. For example:

```

auto mul(int i, double d) { return i*d; }    // here, "auto" means "deduce the return type"

```

This can be convenient, especially for generic functions (function templates; §6.3.1) and lambdas (§6.3.3), but should be used carefully because a deduced type does not offer a stable interface: a change to the implementation of the function (or lambda) can change the type.

3.6.3 Structured Binding

A function can return only a single value, but that value can be a class object with many members. This allows us to efficiently return many values. For example:

```

struct Entry {
    string name;
    int value;
};

```

```

Entry read_entry(istream& is)    // naive read function (for a better version, see §10.5)
{
    string s;
    int i;
    is >> s >> i;
    return {s,i};
}

auto e = read_entry(cin);

cout << "{ " << e.name << " , " << e.value << " }\n";

```

Here, `{s,i}` is used to construct the `Entry` return value. Similarly, we can “unpack” an `Entry`’s members into local variables:

```

auto [n,v] = read_entry(is);
cout << "{ " << n << " , " << v << " }\n";

```

The `auto [n,v]` declares two local variables `n` and `v` with their types deduced from `read_entry()`’s return type. This mechanism for giving local names to members of a class object is called *structured binding*.

Consider another example:

```

map<string,int> m;
// ... fill m ...
for (const auto [key,value] : m)
    cout << "{ " << key << " , " << value << " }\n";

```

As usual, we can decorate `auto` with `const` and `&`. For example:

```

void incr(map<string,int>& m)    // increment the value of each element of m
{
    for (auto& [key,value] : m)
        ++value;
}

```

When structured binding is used for a class with no private data, it is easy to see how the binding is done: there must be the same number of names defined for the binding as there are nonstatic data members of the class, and each name introduced in the binding names the corresponding member. There will not be any difference in the object code quality compared to explicitly using a composite object; the use of structured binding is all about how best to express an idea.

It is also possible to handle classes where access is through member functions. For example:

```

complex<double> z = {1,2};
auto [re,im] = z+2;    // re=3; im=2

```

A `complex` has two data members, but its interface consists of access functions, such as `real()` and `imag()`. Mapping a `complex<double>` to two local variables, such as `re` and `im` is feasible and efficient, but the technique for doing so is beyond the scope of this book.

3.7 Advice

- [1] Distinguish between declarations (used as interfaces) and definitions (used as implementations); §3.1.
- [2] Use header files to represent interfaces and to emphasize logical structure; §3.2; [CG: SF.3].
- [3] **#include** a header in the source file that implements its functions; §3.2; [CG: SF.5].
- [4] Avoid non-inline function definitions in headers; §3.2; [CG: SF.2].
- [5] Prefer **modules** over headers (where **modules** are supported); §3.3.
- [6] Use namespaces to express logical structure; §3.4; [CG: SF.20].
- [7] Use **using**-directives for transition, for foundational libraries (such as **std**), or within a local scope; §3.4; [CG: SF.6] [CG: SF.7].
- [8] Don't put a **using**-directive in a header file; §3.4; [CG: SF.7].
- [9] Throw an exception to indicate that you cannot perform an assigned task; §3.5; [CG: E.2].
- [10] Use exceptions for error handling only; §3.5.3; [CG: E.3].
- [11] Use error codes when an immediate caller is expected to handle the error; §3.5.3.
- [12] Throw an exception if the error is expected to percolate up through many function calls; §3.5.3.
- [13] If in doubt whether to use an exception or an error code, prefer exceptions; §3.5.3.
- [14] Develop an error-handling strategy early in a design; §3.5; [CG: E.12].
- [15] Use purpose-designed user-defined types as exceptions (not built-in types); §3.5.1.
- [16] Don't try to catch every exception in every function; §3.5; [CG: E.7].
- [17] Prefer RAII to explicit **try**-blocks; §3.5.1, §3.5.2; [CG: E.6].
- [18] If your function may not throw, declare it **noexcept**; §3.5; [CG: E.12].
- [19] Let a constructor establish an invariant, and throw if it cannot; §3.5.2; [CG: E.5].
- [20] Design your error-handling strategy around invariants; §3.5.2; [CG: E.4].
- [21] What can be checked at compile time is usually best checked at compile time; §3.5.5 [CG: P.4] [CG: P.5].
- [22] Pass “small” values by value and “large” values by references; §3.6.1; [CG: F.16].
- [23] Prefer pass-by-**const**-reference over plain pass-by-reference; `_module.arguments_`; [CG: F.17].
- [24] Return values as function-return values (rather than by out-parameters); §3.6.2; [CG: F.20] [CG: F.21].
- [25] Don't overuse return-type deduction; §3.6.2.
- [26] Don't overuse structured binding; using a named return type is often clearer documentation; §3.6.3.

I

Index

*Knowledge is of two kinds.
We know a subject ourselves,
or we know where
we can find information on it.
– Samuel Johnson*

Token

`!=`
 container 147
 not-equal operator 6
`"`, string literal 3
`$`, [regex](#) 117
`%`
 modulus operator 6
 remainder operator 6
`%=`, operator 7
`&`
 address-of operator 11
 reference to 12
`&&`, rvalue reference 71
`(`, [regex](#) 117
`()`, call operator 85
`(?:` pattern 120
`)`, [regex](#) 117
`*`
 contents-of operator 11
 multiply operator 6
 pointer to 11
 [regex](#) 117
`*=`, scaling operator 7

`*?` lazy 118
`+`
 plus operator 6
 [regex](#) 117
 string concatenation 111
`++`, increment operator 7
`+=`
 operator 7
 string append 112
`+`? lazy 118
`-`, minus operator 6
`--`, decrement operator 7
`.`, [regex](#) 117
`/`, divide operator 6
`//` comment 2
`/=`, scaling operator 7
`:public` 55
`<<` 75
 output operator 3
`<=`
 container 147
 less-than-or-equal operator 6
`<`
 container 147
 less-than operator 6
`=`
 0 54
 and `==` 7

- assignment 16
- [auto](#) 8
- container 147
- initializer 7
- [string](#) assignment 112
- ==**
 - = and 7
 - container 147
 - equal operator 6
 - [string](#) 112
- >**
 - container 147
 - greater-than operator 6
- >=**
 - container 147
 - greater-than-or-equal operator 6
- >>**
 - 75
 - template arguments 215
- [?, regex](#) 117
- [??](#) lazy 118
- [\[, regex](#) 117
- [\[\]](#)
 - [array](#) 171
 - array of 11
 - [string](#) 112
- [\](#), backslash 3
- [\], regex](#) 117
- [^](#), regex 117
- [{, regex](#) 117
- [{ }](#)
 - grouping 2
 - initializer 8
- [{ }?](#) lazy 118
- [|, regex](#) 117
- [}, regex](#) 117
- [,](#) destructor 51
- 0**
 - = 54
 - [nullptr](#) **NULL** 13

A

- [abs\(\)](#) 188
- abstract
 - [class](#) 54
 - type 54
- [accumulate\(\)](#) 189
- acquisition RAI, resource 164
- adaptor, lambda as 180
- address, memory 16
- address-of operator [&](#) 11
- [adjacent_difference\(\)](#) 189
- aims, C++11 213
- algorithm 149
 - container 150, 160
 - lifting 100
 - numerical 189
 - parallel 161
 - standard library 156
- [<algorithm>](#) 109, 156
- alias, [using](#) 90
- [alignas](#) 215
- [alignof](#) 215
- allocation 51
- allocator [new](#), container 178
- almost container 170
- [alnum, regex](#) 119
- [alpha, regex](#) 119
- [\[:alpha:\]](#) letter 119
- ANSI C++ 212
- [any](#) 177
- append [+=](#), [string](#) 112
- argument
 - constrained 81
 - constrained template 82
 - default function 42
 - default template 98
 - function 41
 - passing, function 66
 - type 82
 - value 82
- arithmetic
 - conversions, usual 7
 - operator 6
 - vector 192
- ARM 212
- array
 - [array](#) vs. 172
 - of [\[\]](#) 11
- [array](#) 171
 - [\[\]](#) 171
 - [data\(\)](#) 171
 - initialize 171
 - [size\(\)](#) 171
 - vs. [array](#) 172
 - vs. [vector](#) 171
- [<array>](#) 109
- [asin\(\)](#) 188
- assembler 210
- [assert\(\)](#) 40
- assertion [static_assert](#) 40
- [Assignable](#) 158
- assignment
 - = 16
 - =, [string](#) 112
 - copy 66, 69
 - initialization and 18
 - move 66, 72
- associative array – see [map](#)
- [async\(\)](#) launch 204
- [at\(\)](#) 141
- [atan\(\)](#) 188

atan2() 188
 AT&T Bell Laboratories 212
 auto = 8
 auto_ptr, deprecated 218

B

back_inserter() 150
 backslash \ 3
 bad_variant_access 176
 base and derived class 55
 basic_string 114
 BCPL 219
 begin() 75, 143, 147, 150
 beginner, book for 1
 Bell Laboratories, AT&T 212
 beta() 188
 bibliography 222
 BidirectionalIterator 159
 BidirectionalRange 160
 binary search 156
 binding, structured 45
 bit-field, bitset and 172
 bitset 172
 and bit-field 172
 and enum 172
 blank, regex 119
 block
 as function body, try 141
 try 36
 body, function 2
 book for beginner 1
 bool 5
 Boolean 158
 BoundedRange 160
 break 15

C

C 209
 and C++ compatibility 218
 Classic 219
 difference from 218
 K&R 219
 void * assignment, difference from 221
 with Classes 208
 with Classes language features 210
 with Classes standard library 211
 C++
 ANSI 212
 compatibility, C and 218
 Core Guidelines 214
 core language 2
 history 207
 ISO 212

 meaning 209
 modern 214
 pronunciation 209
 standard, ISO 2
 standard library 2
 standardization 212
 timeline 208
 C++03 212
 C++0x, C++11 209, 212
 C++11
 aims 213
 C++0x 209, 212
 language features 215
 library components 216
 C++14
 language features 216
 library components 217
 C++17
 language features 216
 library components 217
 C++98 212
 standard library 211
 C11 218
 C89 and C99 218
 C99, C89 and 218
 call operator () 85
 callback 181
 capacity() 139, 147
 capture list 87
 carries_dependency 215
 cast 53
 catch
 clause 36
 every exception 141
 catch(...) 141
 ceil() 188
 char 5
 character sets, multiple 114
 check
 compile-time 40
 run-time 40
 checking, cost of range 142
 chrono, namespace 179
 <chrono> 109, 179, 200
 class 48
 concrete 48
 scope 9
 template 79
 class
 abstract 54
 base and derived 55
 hierarchy 57
 Classic C 219
 C-library header 110
 clock timing 200
 <cmath> 109, 188

- `cntrl, regex` 119
- code complexity, function and 4
- comment, // 2
- `Common` 158
- `CommonReference` 158
- `common_type_t` 158
- communication, task 202
- comparison 74
 - operator 6, 74
- compatibility, C and C++ 218
- compilation
 - model, template 104
 - separate 30
- compiler 2
- compile-time
 - check 40
 - computation 181
 - evaluation 10
- complete encapsulation 66
- `complex` 49, 190
- `<complex>` 109, 188, 190
- complexity, function and code 4
- components
 - C++11 library 216
 - C++14 library 217
 - C++17 library 217
- computation, compile-time 181
- concatenation +, `string` 111
- concept 81, 94
 - range 157
- `concept` support 94
- concrete
 - class 48
 - type 48
- concurrency 195
- condition, declaration in 61
- `condition_variable` 201
 - `notify_one()` 202
 - `wait()` 201
- `<condition_variable>` 201
- `const`
 - immutability 9
 - member function 50
- constant expression 10
- `const_cast` 53
- `constexpr`
 - function 10
 - immutability 9
- `const_iterator` 154
- constrained
 - argument 81
 - template 82
 - template argument 82
- `Constructible` 158
- constructor
 - and destructor 210
 - copy 66, 69
 - default 50
 - delegating 215
 - `explicit` 67
 - inheriting 216
 - initializer-list 52
 - invariant and 37
 - move 66, 71
- container 51, 79, 137
 - > 147
 - = 147
 - >= 147
 - < 147
 - == 147
 - != 147
 - <= 147
 - algorithm 150, 160
 - allocator `new` 178
 - almost 170
 - object in 140
 - overview 146
 - `return` 151
 - `sort()` 181
 - specialized 170
 - standard library 146
- contents-of operator * 11
- contract 40
- conversion 67
 - explicit type 53
 - narrowing 8
- conversions, usual arithmetic 7
- `ConvertibleTo` 158
- `copy` 68
 - assignment 66, 69
 - constructor 66, 69
 - cost of 70
 - elision 72
 - elision 66
 - memberwise 66
- `copy()` 156
- `Copyable` 158
- `CopyConstructible` 158
- `copy_if()` 156
- Core Guidelines, C++ 214
- core language, C++ 2
- coroutine 211
- `cos()` 188
- `cosh()` 188
- cost
 - of copy 70
 - of range checking 142
- `count()` 156
- `count_if()` 155–156
- `cout`, output 3
- `<cstdlib>` 110
- C-style

error handling 188
string 13

D

`\d, regex` 119
`d, regex` 119
`\D, regex` 119
data race 196
`data(), array` 171
D&E 208
deadlock 199
deallocation 51
debugging `template` 100
declaration 5
 function 4
 in condition 61
 interface 29
-declaration, `using` 34
declarator operator 12
`decltype` 215
decrement operator `--` 7
deduction
 guide 83, 176
 `return`-type 44
default
 constructor 50
 function argument 42
 member initializer 68
 operations 66
 template argument 98
`=default` 66
`DefaultConstructible` 158
definition implementation 30
delegating constructor 215
`=delete` 67
`delete`
 naked 52
 operator 51
deprecated
 `auto_ptr` 218
 feature 218
`deque` 146
derived `class`, base and 55
`DerivedFrom` 158
`Destructible` 158
destructor 51, 66
 - 51
 constructor and 210
 `virtual` 59
dictionary – see `map`
difference
 from C 218
 from C `void*` assignment 221
digit, `[[:digit:]]` 119
`digit, regex` 119

`[[:digit:]]` digit 119
-directive, `using` 35
dispatch, tag 181
distribution, `random` 191
divide operator `/` 6
domain error 188
`double` 5
duck typing 104
`duration` 179
`duration_cast` 179
dynamic store 51
`dynamic_cast` 61
 is instance of 62
 is kind of 62

E

`EDOM` 188
element requirements 140
elision, copy 66
`emplace_back()` 147
`empty()` 147
`enable_if` 184
encapsulation, complete 66
`end()` 75, 143, 147, 150
engine, `random` 191
`enum, bitset` and 172
equal operator `==` 6
equality preserving 159
`EqualityComparable` 158
`equal_range()` 156, 173
`ERANGE` 188
`erase()` 143, 147
`errno` 188
error
 domain 188
 handling 35
 handling, C-style 188
 range 188
 recovery 38
 run-time 35
error-code, exception vs 38
essential operations 66
evaluation
 compile-time 10
 order of 7
example
 `find_all()` 151
 Hello, World! 2
 `Rand_int` 191
 `Vec` 141
exception 35
 and `main()` 141
 `catch` every 141
 specification, removed 218
 vs error-code 38

[exclusive_scan\(\)](#) 189
 execution policy 161
 explicit type conversion 53
[explicit](#) constructor 67
[exponential_distribution](#) 191
[export](#) removed 218
[expr\(\)](#) 188
 expression
 constant 10
 lambda 87
[extern template](#) 215

F

[fabs\(\)](#) 188
 facilities, standard library 108
[fail_fast](#) 170
 feature, deprecated 218
 features
 C with Classes language 210
 C++11 language 215
 C++14 language 216
 C++17 language 216
 file, header 31
[final](#) 216
[find\(\)](#) 150, 156
[find_all\(\)](#) example 151
[find_if\(\)](#) 155–156
[first](#), [pair](#) member 173
[floor\(\)](#) 188
[fmod\(\)](#) 188
[for](#)
 statement 11
 statement, range 11
[forward\(\)](#) 167
 forwarding, perfect 168
[ForwardIterator](#) 159
[forward_list](#) 146
 singly-linked list 143
[<forward_list>](#) 109
[ForwardRange](#) 160
 free store 51
[frexp\(\)](#) 188
[<fstream>](#) 109
[__func__](#) 215
[function](#) 2
 and code complexity 4
 argument 41
 argument, default 42
 argument passing 66
 body 2
 body, [try](#) block as 141
 [const](#) member 50
 [constexpr](#) 10
 declaration 4
 implementation of [virtual](#) 56

 mathematical 188
 object 85
 overloading 4
 return value 41
 [template](#) 84
 type 181
 value return 66
[function](#) 180
 and [nullptr](#) 180
 fundamental type 5
[future](#)
 and [promise](#) 202
 member [get\(\)](#) 202
[<future>](#) 109, 202

G

garbage collection 73
 generic programming 93, 210
[get<>\(\)](#)
 by index 174
 by type 174
[get\(\)](#), [future](#) member 202
[graph](#), [regex](#) 119
 greater-than operator [>](#) 6
 greater-than-or-equal operator [>=](#) 6
 greedy match 118, 121
 grouping, [{}](#) 2
[gsl](#)
 [namespace](#) 168
 [span](#) 168
 Guidelines, C++ Core 214

H

half-open sequence 156
[handle](#) 52
 resource 69, 165
 hardware, mapping to 16
 hash table 144
[hash<>](#), [unordered_map](#) 76
[header](#)
 C-library 110
 file 31
 standard library 109
[heap](#) 51
[Hello, World!](#) example 2
 hierarchy
 [class](#) 57
 navigation 61
 history, C++ 207
 HOPL 208

I

if statement 14
 immutability
 const 9
 constexpr 9
 implementation
 definition 30
 inheritance 60
 iterator 153
 of **virtual** function 56
 string 113
 in-class member initialization 215
#include 31
inclusive_scan() 189
 increment operator **++** 7
 index, **get<>()** by 174
 inheritance 55
 implementation 60
 interface 60
 multiple 211
 inheriting constructor 216
 initialization
 and assignment 18
 in-class member 215
 initialize 52
 array 171
 initializer
 = 7
 {} 8
 default member 68
 initializer-list constructor 52
initializer_list 52
inline 49
 namespace 215
 inlining 49
inner_product() 189
InputIterator 159
InputRange 160
insert() 143, 147
 instantiation 81
 instruction, machine 16
int 5
 output bits of 172
Integral 158
 interface
 declaration 29
 inheritance 60
 invariant 37
 and constructor 37
Invocable 159
InvocableRegular 159
 I/O, iterator and 154
<ios> 109
<iostream> 3, 109
iota() 189
 is

 instance of, **dynamic_cast** 62
 kind of, **dynamic_cast** 62

ISO

C++ 212
 C++ standard 2
 ISO-14882 212
istream_iterator 154
 iterator 75, 150
 and I/O 154
 implementation 153
Iterator 159
iterator 143, 154
<iterator> 182
iterator_category 182
iterator_traits 181-182
iterator_type 182

J

join(), thread 196

K

key and value 144
 K&R C 219

L

\\, regex 119
\\L, regex 119
 lambda
 as adaptor 180
 expression 87
 language
 and library 107
 features, C with Classes 210
 features, C++11 215
 features, C++14 216
 features, C++17 216
 launch, **async()** 204
 lazy
 *? 118
 +? 118
 ?? 118
 {}? 118
 match 118, 121
ldexp() 188
 leak, resource 62, 72, 164
 less-than operator **<** 6
 less-than-or-equal operator **<=** 6
 letter, **[[:alpha:]]** 119
 library
 algorithm, standard 156
 C with Classes standard 211
 C++98 standard 211

- components, C++11 216
- components, C++14 217
- components, C++17 217
- container, standard 146
- facilities, standard 108
- language and 107
- non-standard 107
- standard 107
- lifetime, scope and 9
- lifting algorithm 100
- `<limits>` 181, 193
- linker 2
- list
 - capture 87
 - `forward_list` singly-linked 143
- `list` 142, 146
- literal
 - "", string 3
 - raw string 116
 - suffix, `s` 113
 - suffix, `sv` 115
 - type of string 113
 - user-defined 75, 215
- literals
 - `string_literals` 113
 - `string_view_literals` 115
- local scope 9
- lock, reader-writer 200
- `log()` 188
- `log10()` 188
- long long 215
- lower, `regex` 119

M

- machine instruction 16
- `main()` 2
 - exception and 141
- `make_pair()` 173
- `make_shared()` 166
- `make_tuple()` 174
- `make_unique()` 166
- management, resource 72, 164
- `map` 144, 146
 - and `unordered_map` 146
- `<map>` 109
- mapped type, value 144
- mapping to hardware 16
- match
 - greedy 118, 121
 - lazy 118, 121
- mathematical
 - function 188
 - functions, special 188
 - functions, standard 188
- `<math.h>` 188

- Max Munch rule 118
- meaning, C++ 209
- member
 - function, `const` 50
 - initialization, in-class 215
 - initializer, default 68
- memberwise copy 66
- `mem_fn()` 180
- memory 73
 - address 16
- `<memory>` 109, 164, 166
- `merge()` 156
- `Mergeable` 159
- minus operator - 6
- model, template compilation 104
- modern C++ 214
- `modf()` 188
- modularity 29
- `module` 32
 - support 32
- modulus operator `%` 6
- `Movable` 158
- move 71
 - assignment 66, 72
 - constructor 66, 71
- `move()` 72, 156, 167
- `MoveConstructible` 158
- moved-from
 - object 72
 - state 168
- move-only type 167
- multi-line pattern 117
- `multimap` 146
- multiple
 - character sets 114
 - inheritance 211
 - return-values 44
- multiply operator `*` 6
- `multiset` 146
- `mutex` 199
- `<mutex>` 199

N

- `\n`, newline 3
- naked
 - `delete` 52
 - `new` 52
- namespace scope 9
- `namespace` 34
 - `chrono` 179
 - `gsl` 168
 - `inline` 215
 - `pmr` 178
 - `std` 3, 35, 109
- narrowing conversion 8

navigation, hierarchy 61
[new](#)
 container allocator 178
 naked 52
 operator 51
 newline [\n](#) 3
[noexcept](#) 37
[noexcept\(\)](#) 215
 non-memory resource 73
 non-standard library 107
[noreturn](#) 215
[normal_distribution](#) 191
 notation, regular expression 117
 not-equal operator [!=](#) 6
[notify_one\(\), condition_variable](#) 202
[NULL 0, nullptr](#) 13
[nullptr](#) 13
 function and 180
 [NULL 0](#) 13
 number, random 191
[<numeric>](#) 189
 numerical algorithm 189
[numeric_limits](#) 193

O

object 5
 function 85
 in container 140
 moved-from 72
 object-oriented programming 57, 210
 operations
 default 66
 essential 66
 operator
 [% =](#) 7
 [+ =](#) 7
 [&](#), address-of 11
 [\(\)](#), call 85
 [*](#), contents-of 11
 [--](#), decrement 7
 [/](#), divide 6
 [==](#), equal 6
 [>](#), greater-than 6
 [> =](#), greater-than-or-equal 6
 [++](#), increment 7
 [<](#), less-than 6
 [< =](#), less-than-or-equal 6
 [-](#), minus 6
 [%](#), modulus 6
 [*](#), multiply 6
 [!=](#), not-equal 6
 [<<](#), output 3
 [+](#), plus 6
 [%](#), remainder 6
 [*=](#), scaling 7

[/=](#), scaling 7
 arithmetic 6
 comparison 6, 74
 declarator 12
 [delete](#) 51
 [new](#) 51
 overloaded 51
 user-defined 51
 optimization, short-string 113
[optional](#) 176
 order of evaluation 7
[ostream_iterator](#) 154
[out_of_range](#) 141
 output
 bits of [int](#) 172
 [cout](#) 3
 operator [<<](#) 3
[OutputIterator](#) 159
[OutputRange](#) 160
 overloaded operator 51
 overloading, function 4
[override](#) 55
 overview, container 146
 ownership 164

P

[packaged_task thread](#) 203
[pair](#) 173
 and structured binding 174
 member [first](#) 173
 member [second](#) 173
[par](#) 161
 parallel algorithm 161
 parameterized type 79
[partial_sum\(\)](#) 189
[par_unseq](#) 161
 passing data to task 197
 pattern 116
 [\(?\)](#): 120
 multi-line 117
 perfect forwarding 168
[Permutable](#) 159
[phone_book](#) example 138
 plus operator [+](#) 6
[pmr, namespace](#) 178
 pointer 17
 smart 164
 to [*](#) 11
 policy, execution 161
 polymorphic type 54
[pow\(\)](#) 188
 precondition 37
 predicate 86, 155
 type 183
[Predicate](#) 159

[print, regex](#) 119
 procedural programming 2
 program 2
 programming
 generic 93, 210
 object-oriented 57, 210
 procedural 2
[promise](#)
 future and 202
 member [set_exception\(\)](#) 202
 member [set_value\(\)](#) 202
 pronunciation, C++ 209
[punct, regex](#) 119
 pure virtual 54
 purpose, [template](#) 93
[push_back\(\)](#) 52, 139, 143, 147
[push_front\(\)](#) 143

R

[R"](#) 116
 race, data 196
[RAII](#)
 and resource management 36
 and [try-block](#) 40
 and [try-statement](#) 36
 resource acquisition 164
 [scoped_lock](#) and 199–200
[RAII](#) 52
[Rand_int](#) example 191
 random number 191
[random](#)
 distribution 191
 engine 191
[<random>](#) 109, 191
[RandomAccessIterator](#) 159
[RandomAccessRange](#) 160
 range
 checking, cost of 142
 checking [Vec](#) 140
 concept 157
 error 188
 for statement 11
[Range](#) 157, 160
 raw string literal 116
 reader-writer lock 200
 recovery, error 38
[reduce\(\)](#) 189
 reference 17
 [&&](#), rvalue 71
 rvalue 72
 to [&](#) 12
[regex](#)
] 117
 [117
 . 117

? 117
 . 117
 + 117
 * 117
) 117
 (117
 \$ 117
 { 117
 } 117
 | 117
[alnum](#) 119
[alpha](#) 119
[blank](#) 119
[cntrl](#) 119
[\D](#) 119
[\d](#) 119
[d](#) 119
[digit](#) 119
[graph](#) 119
[\l](#) 119
[\L](#) 119
[lower](#) 119
[print](#) 119
[punct](#) 119
 regular expression 116
 repetition 118
[\s](#) 119
[\S](#) 119
[s](#) 119
[space](#) 119
[\U](#) 119
[\u](#) 119
[upper](#) 119
[w](#) 119
[\w](#) 119
[\W](#) 119
[xdigit](#) 119
[<regex>](#) 109, 116
 regular expression 116
[regex_iterator](#) 121
[regex_search](#) 116
 regular
 expression notation 117
 expression [<regex>](#) 116
 expression [regex](#) 116
[Regular](#) 158
[reinterpret_cast](#) 53
[Relation](#) 159
 remainder operator [%](#) 6
 removed
 exception specification 218
 [export](#) 218
 repetition, [regex](#) 118
[replace\(\)](#) 156
 [string](#) 112
[replace_if\(\)](#) 156

requirement, [template](#) 94
 requirements, element 140
[reserve\(\)](#) 139, 147
[resize\(\)](#) 147
 resource
 acquisition RAI 164
 handle 69, 165
 leak 62, 72, 164
 management 72, 164
 management, RAI and 36
 non-memory 73
 retention 73
 safety 72
 rethrow 38
 return
 function value 66
 type, suffix 215
 value, function 41
[return](#)
 container 151
 type, [void](#) 3
 returning results from task 198
[return-type deduction](#) 44
 return-values, multiple 44
[riemannzeta\(\)](#) 188
 rule
 Max Munch 118
 of zero 67
 run-time
 check 40
 error 35
 rvalue
 reference 72
 reference [&&](#) 71

S

[s](#) literal suffix 113
[\s, regex](#) 119
[s, regex](#) 119
[\S, regex](#) 119
 safety, resource 72
[Same](#) 158
 scaling
 operator [/=](#) 7
 operator [*=](#) 7
 scope
 and lifetime 9
 class 9
 local 9
 namespace 9
[scoped_lock](#) 164
 and RAI 199–200
 [unique_lock](#) and 201
[scoped_lock\(\)](#) 199
 search, binary 156

[second, pair](#) member 173
[Semiregular](#) 158
[Sentinel](#) 159
 separate compilation 30
 sequence 150
 half-open 156
[set](#) 146
[<set>](#) 109
[set_exception\(\), promise](#) member 202
[set_value\(\), promise](#) member 202
[shared_lock](#) 200
[shared_mutex](#) 200
[shared_ptr](#) 164
 sharing data task 199
 short-string optimization 113
[SignedIntegral](#) 158
 SIMD 161
 Simula 207
[sin\(\)](#) 188
 singly-linked list, [forward_list](#) 143
[sinh\(\)](#) 188
 size of type 6
[size\(\)](#) 75, 147
 array 171
[SizeRange](#) 160
[SizedSentinel](#) 159
[sizeof](#) 6
[sizeof\(\)](#) 181
[size_t](#) 90
 smart pointer 164
[smatch](#) 116
[sort\(\)](#) 149, 156
 container 181
[Sortable](#) 159
[space, regex](#) 119
[span](#)
 [gsl](#) 168
 [string_view](#) and 168
 special mathematical functions 188
 specialized container 170
[spherical_bessel\(\)](#) 188
[sqrt\(\)](#) 188
[<sstream>](#) 109
 standard
 ISO C++ 2
 library 107
 library algorithm 156
 library, C++ 2
 library, C with Classes 211
 library, C++98 211
 library container 146
 library facilities 108
 library header 109
 library [std](#) 109
 mathematical functions 188
 standardization, C++ 212

- state, moved-from 168
- statement
 - for 11
 - if 14
 - range for 11
 - switch 14
 - while 14
- static_assert 193
 - assertion 40
- static_cast 53
- std
 - namespace 3, 35, 109
 - standard library 109
- <stdexcept> 109
- STL 211
- store
 - dynamic 51
 - free 51
- StrictTotallyOrdered 158
- StrictWeakOrder 159
- string
 - C-style 13
 - literal " 3
 - literal, raw 116
 - literal, type of 113
 - Unicode 114
- string 111
 - [] 112
 - == 112
 - append += 112
 - assignment = 112
 - concatenation + 111
 - implementation 113
 - replace() 112
 - substr() 112
- <string> 109, 111
- string_literals, literals 113
- string_span 170
- string_view 114
 - and span 168
- string_view_literals, literals 115
- structured
 - binding 45
 - binding, pair and 174
 - binding, tuple and 174
- subclass, superclass and 55
- [] subscripting 147
- substr(), string 112
- suffix 75
 - return type 215
 - s literal 113
 - sv literal 115
- superclass and subclass 55
- support, module 32
- support, concept 94
- sv literal suffix 115

- swap() 76
- Swappable 158
- SwappableWith 158
- switch statement 14
- synchronized_pool_resource 178

T

- table, hash 144
- tag dispatch 181
- tanh() 188
- task
 - and thread 196
 - communication 202
 - passing data to 197
 - returning results from 198
 - sharing data 199
- TC++PL 208
- template
 - argument, constrained 82
 - argument, default 98
 - arguments, >> 215
 - compilation model 104
 - constrained 82
 - variadic 100
- template 79
 - class 79
 - debugging 100
 - extern 215
 - function 84
 - purpose 93
 - requirement 94
- this 70
- thread
 - join() 196
 - packaged_task 203
 - task and 196
- <thread> 109, 196
- thread_local 216
- time 179
- timeline, C++ 208
- time_point 179
- timing, clock 200
- to hardware, mapping 16
- transform_reduce() 189
- translation unit 32
- try
 - block 36
 - block as function body 141
- try-block, RAI and 40
- try-statement, RAI and 36
- tuple 174
 - and structured binding 174
- type 5
 - abstract 54
 - argument 82

- concrete 48
- conversion, explicit 53
- function 181
- fundamental 5
- get-<>() by 174
- move-only 167
- of string literal 113
- parameterized 79
- polymorphic 54
- predicate 183
- size of 6
- typename 79, 152
- <type_traits> 183
- typing, duck 104

U

- \U, regex 119
- \u, regex 119
- udl 75
- Unicode string 114
- uniform_int_distribution 191
- uninitialized 8
- unique_copy() 149, 156
- unique_lock 200–201
 - and scoped_lock 201
- unique_ptr 62, 164
- unordered_map 144, 146
 - hash<> 76
 - map and 146
- <unordered_map> 109
- unordered_multimap 146
- unordered_multiset 146
- unordered_set 146
- unsigned 5
- UnsignedIntegral 158
- upper, regex 119
- user-defined
 - literal 75, 215
 - operator 51
- using
 - alias 90
 - declaration 34
 - directive 35
- usual arithmetic conversions 7
- <utility> 109, 173–174

V

- valarray 192
- <valarray> 192
- value 5
 - argument 82
 - key and 144
 - mapped type 144

- return, function 66
- value_type 90
- valuetype 147
- variable 5
- variadic template 100
- variant 175
- Vec
 - example 141
 - range checking 140
- vector arithmetic 192
- vector 138, 146
 - array vs. 171
- <vector> 109
- vector<bool> 170
- vectorized 161
- View 160
- virtual 54
 - destructor 59
 - function, implementation of 56
 - function table vtbl 56
 - pure 54
- void
 - * 221
 - * assignment, difference from C 221
 - return type 3
- vtbl, virtual function table 56

W

- w, regex 119
- \w, regex 119
- \W, regex 119
- wait(), condition_variable 201
- WeaklyEqualityComparable 158
- WG21 208
- while statement 14

X

- X3J16 212
- xdigit, regex 119

Z

- zero, rule of 67

Credits

- Page ii: "I have made this letter longer than usual, because I lack the time to make it short." Pascal, B. (1904). *The provincial letters of Blaise Pascal*, J.M. Dent.
- Page x: "When you wish to instruct, be brief", Marcus Tullius Cicero - *Horace for English Readers, Being a Translation of the Poems of Quintus Horatius Flaccus into English Prose*, trans. E.C. Wickham (Oxford: Clarendon Press, 1903).
- Page 1: "The first thing we do, let's kill all the language lawyers", paraphrasing William Shakespeare - *Henry The Sixth, Part 2 Act 4, scene 2, 71-78*.
- Page 21: "Don't Panic!", Neil Gaiman, *Don't Panic: The Official Hitchhiker's Guide to the Galaxy Companion Pocket Books*, 1988.
- Page 29: "Don't interrupt me while I'm interrupting", Winston Churchill (1966). "The Irrepressible Churchill: Stories, Sayings and Impressions of Sir Winston Churchill", World Publishing Company.
- Page 49: "Those types are not 'abstract'; they are as real as int and float", Doug McIlroy.
- Page 67: "When someone says I want a programming language in which I need only say what I wish done, give him a lollipop", Alan Perlis ACM-SIGPLAN '82, *Epigrams in Programming*.
- Page 95: "Programming: you have to start with interesting algorithms", Alex Stepanov.
- Page 109: "Why waste time learning when ignorance is instantaneous?", Watterson, B. (1992). *Attack of the deranged mutant killer monster snow goons: A Calvin and Hobbes collection*. Kansas City: Andrews and McMeel.
- Page 113: "Prefer the standard to the offbeat", Strunk, W., & White, E. B. (1959). *The elements of style*. N.Y: Macmillan.
- Page 125: "What you see is all you get", Kernighan, B. W., & Ritchie, D. M. (1988). *The C programming language*. Upper Saddle River NJ: Prentice Hall.
- Page 141: "It was new. It was singular. It was simple. It must succeed!", Horatio Nelson.
- Page 153: "Do not multiply entities beyond necessity." William Occam.
- Page 160: "a finite set of rules...Input ... Output ... Effectiveness", Donald E. Knuth: *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts. 1968.
- Page 167: "The time you enjoy wasting is not wasted time." Bertrand Russell.
- Page 193: "The purpose of computing is insight, not numbers", Hamming, R. W. (1962). *Numerical methods for scientists and engineers*.
- Page 193: "... but for the student, numbers are often the best road to insight." *A first course in numerical analysis*, Anthony Ralston, McGraw-Hill, 1965.
- Page 201: "Keep it simple: as simple as possible, but no simpler", Albert Einstein.
- Page 215: "Hurry Slowly (festina lente)", Octavius, Caesar Augustus quoted in Alison Jones, Stephanie Pickering, Megan Thomson (1997). *Chambers dictionary of quotations*, Chambers.