# 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
16 History and Compatibility

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
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., [Cppreference]. 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

3

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:

```cpp
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:

```cpp
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:
```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:
```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:

![Diagram showing program organization and separate compilation]

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 `#include`s 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*:

```cpp
// file Vector.cpp:

module; // if 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):

```cpp
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() << ')' << std::endl;
  // ...
}

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); // if std::swap()
    other::swap(x,y); // if 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:

```cpp
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 `namespace`.

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:

```cpp
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:

```cpp
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() << '
';
    }
    // ...
}
```

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:

```cpp
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:

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

This is likely to cause chaos.

Here is a more appropriate definition:

```cpp
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:
```cpp
void test()
{
    try {
        Vector v(-27);
    }
    catch (std::length_error& err) { // handle negative size
        std::cout << "test failed: length error\n";
        throw; // rethrow
    }
    catch (std::bad_alloc& err) { // handle memory exhaustion
        std::terminate(); // terminate the program
    }
}
```

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:

```cpp
void test()
{
    try {
        Vector v(-27);
    }
    catch (std::length_error&) { // do something and rethrow
        std::cout << "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 errors 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:

```cpp
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:

```cpp
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:

```cpp
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:

```cpp
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:

```cpp
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:

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

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

```cpp
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:

```cpp
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);
';  // 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:

```cpp
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 ith 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:

```cpp
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 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] \texttt{#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 \texttt{module}s over headers (where \texttt{module}s are supported); §3.3.
[6] Use namespaces to express logical structure; §3.4; [CG: SF.20].
[7] Use \texttt{using}-directives for transition, for foundational libraries (such as \texttt{std}), or within a local scope; §3.4; [CG: SF.6] [CG: SF.7].
[8] Don’t put a \texttt{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 \texttt{try}-blocks; §3.5.1, §3.5.2; [CG: E.6].
[18] If your function may not throw, declare it \texttt{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].
[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.
Knowledge is of two kinds.
We know a subject ourselves,
or we know where
we can find information on it.
– Samuel Johnson
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 RAII, resource 164
adaptor, lambda as 180
address, memory 16
address-of operator & 11
adjacent_difference() 189
aims, C++ 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
alpha, 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
C99 and C99 218
C99, C89 and 218
call operator () 85
callback 181
capacity() 139, 147
capture list 87 2
carries_dependency 215
cast 53
catch
class 36
every exception 141
catch(...) 141
cell() 188
char 5
class abstract 54
base and derived 55
hierarchy 57
Classic C 219
C-library header 110
clock timing 200
<cmath> 109, 188
error handling 188
string 13

D
id, regex 119
d, regex 119
\d, regex 119
data race 196
data(), array 171
D&E 208
deallocation 51
debugging template 100
declaration 5
  function 4
  in condition 61
  interface 29
-declaration, using 34
declarator operator 12
declopye 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 39
dictionary – see map
difference
  from C 218
  from C void * assignment 221
digit, \[[:digit:]\] 119
digit, regex 119

E
EDOM 188
element requirements 140
eission, copy 66
emplace_back() 147
empty() 147
enable_if 184
eapsulation, complete 66
delet() 75, 143, 147, 150
eine, random 191
enum, bitset and 172
equal operator == 6
equality preserving 159
EqualityComparable 158
eual_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
esential 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
expf() 188
expression
  constant 10
  lambda 87
extern template 215

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
trecht() 188
strftime() 109
__func__ 215
function
  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
getc<i> 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
Index 235

- N -

navigation, hierarchy 61
new
  container allocator 178
  naked 52
  operator 51
newline \ 3
nothrow 37
nothrow() 215
non-memory resource 73
non-standard library 107
notreturn 215
normal_distribution 191
notation, regular expression 117
not-equal operator != 6
notify_one(), condition_variable 202
NULL, 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

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 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() 180
reference 17
  &x, rvalue 71
  to & 12
regex
  + 117
  - 117
  = 117
  { 117
  ( 117
  $ 117
  } 117
  ) 117
  | 117
alnum 119
alpha 119
blank 119
cntl 119
ID 119
'v 119
d 119
digit 119
digit 119
digit 119
digit 119
digit 119
digit 119
digit 119
digit 119
digit 119
digit 119
digit 119
digit 119
digit 119
digit 119
digit 119
<regex> 109, 116
  repetition 118
  repetition 118
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
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
<stdio.h> 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
subclass and subclass 55
suport, 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
tan() 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
<string> 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, RAII and 40
try-statement, RAII 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  
<typename_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  
\nhash< > 76  
\nmap and 146  
<unordered_map> 109  
unordered_multimap 146  
unordered_multiset 146  
unordered_set 146  
unsigned 5  
UnsignedIntegral 158  
upper, regex 119  
user-defined  
\nliteral 75, 215  
\noperator 51  
using  
\nalias 90  
\ndeclaration 34  
\ndirective 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  
\nexample 141  
rage checking 140  
vector arithmetic 192  
vector 138, 146  
\narray vs. 171  
<vector> 109  
<vector<bool>> 170  
vectored 161  
View 160  
virtual 54  
\ndestructor 59  
\nfunction, implementation of 56  
function table vtbl 56  
\npure 54  
void  
\n* 221  
\n* assignment, difference from C 221  
\nreturn type 3  
vtbl, virtual function table 56

W
w, regex 119  
w\w, regex 119  
w\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 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 141: “It was new. It was singular. It was simple. It must succeed!”, Horatio Nelson.
Page 167: “The time you enjoy wasting is not wasted time.” Bertrand Russell.
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.