

Updated for C++ 20



A Tour of C++

Third Edition

Bjarne Stroustrup



C++ In-Depth Series

Bjarne Stroustrup

FREE SAMPLE CHAPTER |



**A Tour of C++
Third Edition**

C++ In-Depth Series

Bjarne Stroustrup, Series Editor



Visit informit.com/series/indepth for a complete list of available publications.

The **C++ In-Depth Series** is a collection of concise and focused books that provide real-world programmers with reliable information about the C++ programming language.

Selected by the designer and original implementor of C++, Bjarne Stroustrup, and written by carefully chosen experts in the field, each book in this series presents either a single topic, at a technical level appropriate to that topic, or a fast-paced overview, for a quick understanding of broader language features. In either case, the series' practical approach is designed to lift professionals (and aspiring professionals) to the next level of programming skill or knowledge.



Make sure to connect with us!
informit.com/socialconnect

A Tour of C++

Third 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

Cover photo by: Marco Pregolato (Unsplash.com: @marco_pregolato).

Author photo courtesy of Bjarne Stroustrup.

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: 2022938722

Copyright © 2023 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.pearson.com/permissions.

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

ISBN-13: 978-0-13-681648-5

ISBN-10: 0-13-681648-7

First printing, October 2022

ScoutAutomatedPrintCode

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	10
1.7 Pointers, Arrays, and References	11
1.8 Tests	14
1.9 Mapping to Hardware	16
1.10 Advice	19
2 User-Defined Types	21
2.1 Introduction	21
2.2 Structures	22
2.3 Classes	23
2.4 Enumerations	25
2.5 Unions	27
2.6 Advice	28

3 Modularity	29
3.1 Introduction	29
3.2 Separate Compilation	30
3.3 Namespaces	35
3.4 Function Arguments and Return Values	37
3.5 Advice	42
4 Error Handling	43
4.1 Introduction	43
4.2 Exceptions	44
4.3 Invariants	45
4.4 Error-Handling Alternatives	47
4.5 Assertions	48
4.6 Advice	51
5 Classes	53
5.1 Introduction	53
5.2 Concrete Types	54
5.3 Abstract Types	60
5.4 Virtual Functions	62
5.5 Class Hierarchies	63
5.6 Advice	69
6 Essential Operations	71
6.1 Introduction	71
6.2 Copy and Move	74
6.3 Resource Management	78
6.4 Operator Overloading	80
6.5 Conventional Operations	81
6.6 User-Defined Literals	84
6.7 Advice	85
7 Templates	87
7.1 Introduction	87
7.2 Parameterized Types	88
7.3 Parameterized Operations	93
7.4 Template Mechanisms	99
7.5 Advice	102

8 Concepts and Generic Programming	103
8.1 Introduction	103
8.2 Concepts	104
8.3 Generic Programming	112
8.4 Variadic Templates	114
8.5 Template Compilation Model	117
8.6 Advice	117
9 Library Overview	119
9.1 Introduction	119
9.2 Standard-Library Components	120
9.3 Standard-Library Organization	121
9.4 Advice	124
10 Strings and Regular Expressions	125
10.1 Introduction	125
10.2 Strings	125
10.3 String Views	128
10.4 Regular Expressions	130
10.5 Advice	136
11 Input and Output	137
11.1 Introduction	137
11.2 Output	138
11.3 Input	139
11.4 I/O State	141
11.5 I/O of User-Defined Types	141
11.6 Output Formatting	143
11.7 Streams	146
11.8 C-style I/O	149
11.9 File System	150
11.10 Advice	154
12 Containers	157
12.1 Introduction	157
12.2 vector	158
12.3 list	162
12.4 forward_list	164
12.5 map	164

12.6	<code>unordered_map</code>	165
12.7	Allocators	167
12.8	Container Overview	168
12.9	Advice	170
13	Algorithms	173
13.1	Introduction	173
13.2	Use of Iterators	175
13.3	Iterator Types	178
13.4	Use of Predicates	181
13.5	Algorithm Overview	181
13.6	Parallel Algorithms	183
13.7	Advice	183
14	Ranges	185
14.1	Introduction	185
14.2	Views	186
14.3	Generators	188
14.4	Pipelines	188
14.5	Concepts Overview	190
14.6	Advice	194
15	Pointers and Containers	195
15.1	Introduction	195
15.2	Pointers	196
15.3	Containers	201
15.4	Alternatives	208
15.5	Advice	212
16	Utilities	213
16.1	Introduction	213
16.2	Time	214
16.3	Function Adaption	216
16.4	Type Functions	217
16.5	<code>source_location</code>	222
16.6	<code>move()</code> and <code>forward()</code>	223
16.7	Bit Manipulation	224
16.8	Exiting a Program	225
16.9	Advice	225

17 Numerics	227
17.1 Introduction	227
17.2 Mathematical Functions	228
17.3 Numerical Algorithms	229
17.4 Complex Numbers	230
17.5 Random Numbers	231
17.6 Vector Arithmetic	233
17.7 Numeric Limits	234
17.8 Type Aliases	234
17.9 Mathematical Constants	234
17.10 Advice	235
18 Concurrency	237
18.1 Introduction	237
18.2 Tasks and threads	238
18.3 Sharing Data	241
18.4 Waiting for Events	243
18.5 Communicating Tasks	245
18.6 Coroutines	250
18.8 Advice	253
19 History and Compatibility	255
19.1 History	255
19.2 C++ Feature Evolution	263
19.3 C/C++ Compatibility	268
19.4 Bibliography	271
19.5 Advice	274
Module std	277
A.1 Introduction	277
A.2 Use What Your Implementation Offers	278
A.3 Use Headers	278
A.4 Make Your Own module std	278
A.5 Advice	279
Index	281

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 or C++11. Furthermore, the resulting programs are better checked by the compiler and run faster.

This book gives an overview of C++ as defined by C++20, the current ISO C++ standard, and implemented by the major C++ suppliers. In addition, it mentions a couple library components in current use, but not scheduled for inclusion into the standard until C++23.

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 day or two but, obviously, there is much more to writing good C++ than can be learned in that amount of time. 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; some stories may sound strange or even implausible. 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. I assume some technical maturity from my readers. 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. The reader is encouraged to search out additional and supporting material as needed. There is far more to the C++ ecosystem than just the facilities offered by ISO standard (e.g., libraries, build systems, analysis tools, and development environments). There is an enormous amount of material (of varying quality) available on the Web. Most readers will find useful tutorial and overview videos from conferences such as CppCon and Meeting C++. For technical details of the language and library offered by the ISO C++ standard, I recommend [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, I rarely identify language features as present in C, C++98, or later ISO standards. Such information can be found in Chapter 19 (History and Compatibility). I focus on fundamentals and try to be brief, but I have not completely resisted the temptation to overrepresent novel features, such as modules (§3.2.2), concepts (§8.2), and coroutines (§18.6). Slightly favoring recent developments 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

Thanks to all who helped complete and correct the earlier editions of *A Tour of C++*, especially to the students in my “Design Using C++” course at Columbia University. Thanks to Morgan Stanley for giving me time to write this third edition. Thanks to Chuck Allison, Guy Davidson, Stephen Dewhurst, Kate Gregory, Danny Kalev, Gor Nishanov, and J.C. van Winkel for reviewing the book and suggesting many improvements.

This book was set using troff by the author using macros originating from Brian Kernighan.

Manhattan, New York

Bjarne Stroustrup

Containers

*It was new. It was singular. It was simple.
It must succeed!
– H. Nelson*

- Introduction
- **vector**
 - Elements; Range Checking
- **list**
- **forward_list**
- **map**
- **unordered_map**
- Allocators
- Container Overview
- Advice

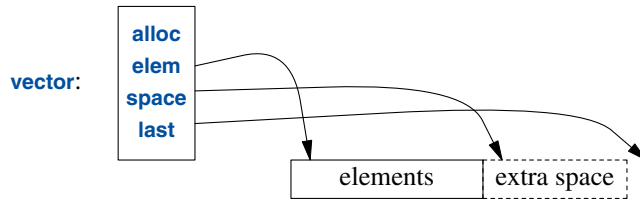
12.1 Introduction

Most computing involves creating collections of values and then manipulating such collections. Reading characters into a **string** and printing out the **string** is a simple example. A class with the main purpose of holding objects is commonly called a *container*. Providing suitable containers for a given task and supporting them with useful fundamental operations are important steps in the construction of any program.

To illustrate the standard-library containers, consider a simple program for keeping names and telephone numbers. This is the kind of program for which different approaches appear “simple and obvious” to people of different backgrounds. The **Entry** class from §11.5 can be used to hold a simple phone book entry. Here, we deliberately ignore many real-world complexities, such as the fact that many phone numbers do not have a simple representation as a 32-bit **int**.

12.2 vector

The most useful standard-library container is **vector**. A **vector** is a sequence of elements of a given type. The elements are stored contiguously in memory. A typical implementation of **vector** (§5.2.2, §6.2) will consist of a handle holding pointers to the first element, one-past-the-last element, and one-past-the-last allocated space (§13.1) (or the equivalent information represented as a pointer plus offsets):



In addition, it holds an allocator (here, **alloc**), from which the **vector** can acquire memory for its elements. The default allocator uses **new** and **delete** to acquire and release memory (§12.7). Using a slightly advanced implementation technique, we can avoid storing any data for simple allocators in a **vector** object.

We can initialize a **vector** with a set of values of its element type:

```
vector<Entry> phone_book = {
    {"David Hume",123456},
    {"Karl Popper",234567},
    {"Bertrand Arthur William Russell",345678}
};
```

Elements can be accessed through subscripting. So, assuming that we have defined `<<` for **Entry**, we can write:

```
void print_book(const vector<Entry>& book)
{
    for (int i = 0; i!=book.size(); ++i)
        cout << book[i] << '\n';
}
```

As usual, indexing starts at **0** so that `book[0]` holds the entry for **David Hume**. The **vector** member function `size()` gives the number of elements.

The elements of a **vector** constitute a range, so we can use a range-**for** loop (§1.7):

```
void print_book(const vector<Entry>& book)
{
    for (const auto& x : book)    // for "auto" see §1.4
        cout << x << '\n';
}
```

When we define a **vector**, we give it an initial size (initial number of elements):

```

vector<int> v1 = {1, 2, 3, 4};           // size is 4
vector<string> v2;                     // size is 0
vector<Shape*> v3(23);                 // size is 23; initial element value: nullptr
vector<double> v4(32,9.9);            // size is 32; initial element value: 9.9

```

An explicit size is enclosed in ordinary parentheses, for example, (23), and by default, the elements are initialized to the element type's default value (e.g., `nullptr` for pointers and `0` for numbers). If you don't want the default value, you can specify one as a second argument (e.g., 9.9 for the 32 elements of `v4`).

The initial size can be changed. One of the most useful operations on a `vector` is `push_back()`, which adds a new element at the end of a `vector`, increasing its size by one. For example, assuming that we have defined `>>` for `Entry`, we can write:

```

void input()
{
    for (Entry e; cin>>e; )
        phone_book.push_back(e);
}

```

This reads `Entries` from the standard input into `phone_book` until either the end-of-input (e.g., the end of a file) is reached or the input operation encounters a format error.

The standard-library `vector` is implemented so that growing a `vector` by repeated `push_back()`s is efficient. To show how, consider an elaboration of the simple `Vector` from Chapter 5 and Chapter 7 using the representation indicated in the diagram above:

```

template<typename T>
class Vector {
    allocator<T> alloc; // standard-library allocator of space for Ts
    T* elem;           // pointer to first element
    T* space;          // pointer to first unused (and uninitialized) slot
    T* last;           // pointer to last slot
public:
    // ...
    int size() const { return space - elem; } // number of elements
    int capacity() const { return last - elem; } // number of slots available for elements
    // ...
    void reserve(int newsz); // increase capacity() to newsz
    // ...
    void push_back(const T& t); // copy t into Vector
    void push_back(T&& t); // move t into Vector
};

```

The standard-library `vector` has members `capacity()`, `reserve()`, and `push_back()`. The `reserve()` is used by users of `vector` and other `vector` members to make room for more elements. It may have to allocate new memory and when it does, it moves the elements to the new allocation. When `reserve()` moves elements to a new and larger allocation, any pointers to those elements will now point to the wrong location; they have become *invalidated* and should not be used.

Given `capacity()` and `reserve()`, implementing `push_back()` is trivial:


```

template<typename T>
void Vector<T>::push_back(const T& t)
{
    if (capacity()<=size())           // make sure we have space for t
        reserve(size()+0?8:2*size()); // double the capacity
    construct_at(space,t);           // initialize *space to t ("place t at space")
    ++space;
}

```

Now allocation and relocation of elements happens only infrequently. I used to use `reserve()` to try to improve performance, but that turned out to be a waste of effort: the heuristic used by `vector` is on average better than my guesses, so now I only explicitly use `reserve()` to avoid reallocation of elements when I want to use pointers to elements.

A `vector` can be copied in assignments and initializations. For example:

```
vector<Entry> book2 = phone_book;
```

Copying and moving `vectors` are implemented by constructors and assignment operators as described in §6.2. Assigning a `vector` involves copying its elements. Thus, after the initialization of `book2`, `book2` and `phone_book` hold separate copies of every `Entry` in the phone book. When a `vector` holds many elements, such innocent-looking assignments and initializations can be expensive. Where copying is undesirable, references or pointers (§1.7) or move operations (§6.2.2) should be used.

The standard-library `vector` is very flexible and efficient. Use it as your default container; that is, use it unless you have a solid reason to use some other container. If you avoid `vector` because of vague concerns about “efficiency,” measure. Our intuition is most fallible in matters of the performance of container uses.

12.2.1 Elements

Like all standard-library containers, `vector` is a container of elements of some type `T`, that is, a `vector<T>`. Just about any type qualifies as an element type: built-in numeric types (such as `char`, `int`, and `double`), user-defined types (such as `string`, `Entry`, `list<int>`, and `Matrix<double,2>`), and pointers (such as `const char*`, `Shape*`, and `double*`). When you insert a new element, its value is copied into the container. For example, when you put an integer with the value `7` into a container, the resulting element really has the value `7`. The element is not a reference or a pointer to some object containing `7`. This makes for nice, compact containers with fast access. For people who care about memory sizes and run-time performance this is critical.

If you have a class hierarchy (§5.5) that relies on `virtual` functions to get polymorphic behavior, do not store objects directly in a container. Instead store a pointer (or a smart pointer; §15.2.1). For example:

```

vector<Shape> vs;           // No, don't - there is no room for a Circle or a Smiley (§5.5)
vector<Shape*> vps;        // better, but see §5.5.3 (don't leak)
vector<unique_ptr<Shape>> vups; // OK

```

12.2.2 Range Checking

The standard-library `vector` does not guarantee range checking. For example:

```
void silly(vector<Entry>& book)
{
    int i = book[book.size()].number;    // book.size() is out of range
    // ...
}
```

That initialization is likely to place some random value in `i` rather than giving an error. This is undesirable, and out-of-range errors are a common problem. Consequently, I often use a simple range-checking adaptation of `vector`:

```
template<typename T>
struct Vec : std::vector<T> {
    using vector<T>::vector;           // use the constructors from vector (under the name Vec)

    T& operator[](int i) { return vector<T>::at(i); }           // range check
    const T& operator[](int i) const { return vector<T>::at(i); } // range check const objects; §5.2.1

    auto begin() { return Checked_iter<vector<T>>{*this}; }     // see §13.1
    auto end() { return Checked_iter<vector<T>>{*this, vector<T>::end()}; }
};
```

`Vec` inherits everything from `vector` except for the subscript operations that it redefines to do range checking. The `at()` operation is a `vector` subscript operation that throws an exception of type `out_of_range` if its argument is out of the `vector`'s range (§4.2).

For `Vec`, an out-of-range access will throw an exception that the user can catch. For example:

```
void checked(Vec<Entry>& book)
{
    try {
        book[book.size()] = {"Joe", 999999};    // will throw an exception
        // ...
    }
    catch (out_of_range&) {
        cerr << "range error\n";
    }
}
```

The exception will be thrown, and then caught (§4.2). If the user doesn't catch an exception, the program will terminate in a well-defined manner rather than proceeding or failing in an undefined manner. One way to minimize surprises from uncaught exceptions is to use a `main()` with a `try`-block as its body. For example:

```
int main()
try {
    // your code
}
```

```

catch (out_of_range&) {
    cerr << "range error\n";
}
catch (...) {
    cerr << "unknown exception thrown\n";
}

```

This provides default exception handlers so that if we fail to catch some exception, an error message is printed on the standard error-diagnostic output stream `cerr` (§11.2).

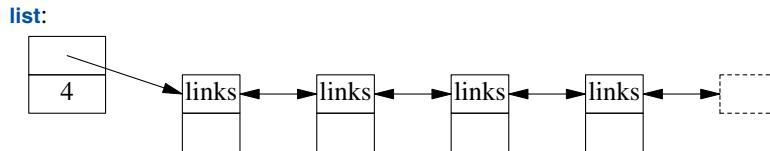
Why doesn't the standard guarantee range checking? Many performance-critical applications use `vectors` and checking all subscripting implies a cost on the order of 10%. Obviously, that cost can vary dramatically depending on hardware, optimizers, and an application's use of subscripting. However, experience shows that such overhead can lead people to prefer the far more unsafe built-in arrays. Even the mere fear of such overhead can lead to disuse. At least `vector` is easily range checked at debug time and we can build checked versions on top of the unchecked default.

A range-`for` avoids range errors at no cost by implicitly accessing all elements in the range. As long as their arguments are valid, the standard-library algorithms do the same to ensure the absence of range errors.

If you use `vector::at()` directly in your code, you don't need my `Vec` workaround. Furthermore, some standard libraries have range-checked `vector` implementations that offer more complete checking than `Vec`.

12.3 list

The standard library offers a doubly-linked list called `list`:



We use a `list` for sequences where we want to insert and delete elements without moving other elements. Insertion and deletion of phone book entries could be common, so a `list` could be appropriate for representing a simple phone book. For example:

```

list<Entry> phone_book = {
    {"David Hume",123456},
    {"Karl Popper",234567},
    {"Bertrand Arthur William Russell",345678}
};

```

When we use a linked list, we tend not to access elements using subscripting the way we commonly do for vectors. Instead, we might search the list looking for an element with a given value. To do this, we take advantage of the fact that a `list` is a sequence as described in Chapter 13:

```

int get_number(const string& s)
{
    for (const auto& x : phone_book)
        if (x.name==s)
            return x.number;
    return 0; // use 0 to represent "number not found"
}

```

The search for **s** starts at the beginning of the list and proceeds until **s** is found or the end of **phone_book** is reached.

Sometimes, we need to identify an element in a **list**. For example, we may want to delete an element or insert a new element before it. To do that we use an *iterator*: a **list** iterator identifies an element of a **list** and can be used to iterate through a **list** (hence its name). Every standard-library container provides the functions **begin()** and **end()**, which return an iterator to the first and to one-past-the-last element, respectively (§13.1). Using iterators explicitly, we can – less elegantly – write the **get_number()** function like this:

```

int get_number(const string& s)
{
    for (auto p = phone_book.begin(); p!=phone_book.end(); ++p)
        if (p->name==s)
            return p->number;
    return 0; // use 0 to represent "number not found"
}

```

In fact, this is roughly the way the terser and less error-prone range-**for** loop is implemented by the compiler. Given an iterator **p**, ***p** is the element to which it refers, **++p** advances **p** to refer to the next element, and when **p** refers to a class with a member **m**, then **p->m** is equivalent to **(*p).m**.

Adding elements to a **list** and removing elements from a **list** is easy:

```

void f(const Entry& ee, list<Entry>::iterator p, list<Entry>::iterator q)
{
    phone_book.insert(p,ee); // add ee before the element referred to by p
    phone_book.erase(q); // remove the element referred to by q
}

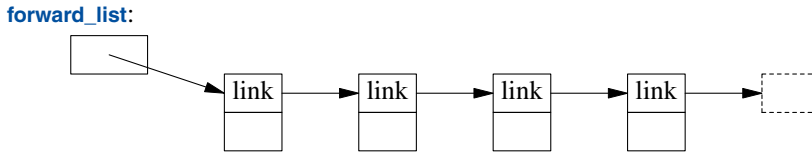
```

For a **list**, **insert(p,elem)** inserts an element with a copy of the value **elem** before the element pointed to by **p**. Here, **p** may be an iterator pointing one-beyond-the-end of the **list**. Conversely, **erase(p)** removes the element pointed to by **p** and destroys it.

These **list** examples could be written identically using **vector** and (surprisingly, unless you understand machine architecture) often perform better with a **vector** than with a **list**. When all we want is a sequence of elements, we have a choice between using a **vector** and a **list**. Unless you have a reason not to, use a **vector**. A **vector** performs better for traversal (e.g., **find()** and **count()**) and for sorting and searching (e.g., **sort()** and **equal_range()**; §13.5, §15.3.3).

12.4 forward_list

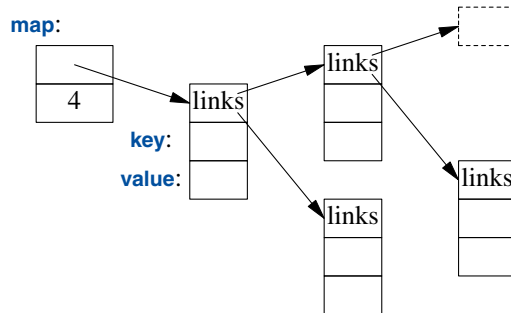
The standard library also offers a singly-linked list called `forward_list`:



A `forward_list` differs from a (doubly-linked) `list` by only allowing forward iteration. The point of that is to save space. There is no need to keep a predecessor pointer in each link and the size of an empty `forward_list` is just one pointer. A `forward_list` doesn't even keep its number of elements. If you need the number of elements, count. If you can't afford to count, you probably shouldn't use a `forward_list`.

12.5 map

Writing code to look up a name in a list of $(name, number)$ pairs is quite tedious. In addition, a linear search is inefficient for all but the shortest lists. The standard library offers a balanced binary search tree (usually a red-black tree) called `map`:



In other contexts, a `map` is known as an associative array or a dictionary.

The standard-library `map` is a container of pairs of values optimized for lookup and insertion. We can use the same initializer as for `vector` and `list` (§12.2, §12.3):

```
map<string,int> phone_book {
    {"David Hume",123456},
    {"Karl Popper",234567},
    {"Bertrand Arthur William Russell",345678}
};
```

When indexed by a value of its first type (called the *key*), a `map` returns the corresponding value of the second type (called the *value* or the *mapped type*). For example:

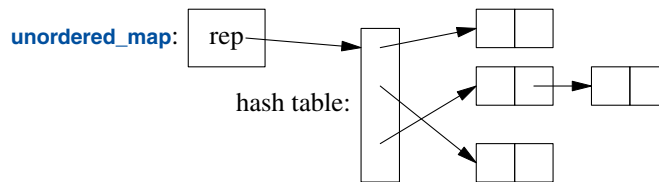
```
int get_number(const string& s)
{
    return phone_book[s];
}
```

In other words, subscripting a `map` is essentially the lookup we called `get_number()`. If a `key` isn't found, it is entered into the `map` with a default value for its `value`. The default value for an integer type is `0` and that just happens to be a reasonable value to represent an invalid telephone number.

If we wanted to avoid entering invalid numbers into our phone book, we could use `find()` and `insert()` (§12.8) instead of `[]`.

12.6 unordered_map

The cost of a `map` lookup is $O(\log(n))$ where `n` is the number of elements in the `map`. That's pretty good. For example, for a `map` with 1,000,000 elements, we perform only about 20 comparisons and indirections to find an element. However, in many cases, we can do better by using a hashed lookup rather than a comparison using an ordering function, such as `<`. The standard-library hashed containers are referred to as “unordered” because they don't require an ordering function:



For example, we can use an `unordered_map` from `<unordered_map>` for our phone book:

```
unordered_map<string,int> phone_book {
    {"David Hume",123456},
    {"Karl Popper",234567},
    {"Bertrand Arthur William Russell",345678}
};
```

Like for a `map`, we can subscript an `unordered_map`:

```
int get_number(const string& s)
{
    return phone_book[s];
}
```

The standard library provides a default hash function for `strings` as well as for other built-in and standard-library types. If necessary, we can provide our own. Possibly, the most common need for a custom hash function comes when we want an unordered container of one of our own types. A hash function is often implemented as a function object (§7.3.2). For example:

```

struct Record {
    string name;
    int product_code;
    // ...
};

struct Rhash {           // a hash function for Record
    size_t operator()(const Record& r) const
    {
        return hash<string>(r.name) ^ hash<int>(r.product_code);
    }
};

unordered_set<Record,Rhash> my_set; // set of Records using Rhash for lookup

```

Designing good hash functions is an art and often requires knowledge of the data to which it will be applied. Creating a new hash function by combining existing hash functions using exclusive-or (^) is simple and often very effective. However, be careful to ensure that every value that takes part in the hash really helps to distinguish the values. For example, unless you can have several names for the same product code (or several product codes for the same name), combining the two hashes provides no benefits.

We can avoid explicitly passing the `hash` operation by defining it as a specialization of the standard-library `hash`:

```

namespace std {           // make a hash function for Record

    template<> struct hash<Record> {
        using argument_type = Record;
        using result_type = size_t;

        result_type operator()(const Record& r) const
        {
            return hash<string>(r.name) ^ hash<int>(r.product_code);
        }
    };
}

```

Note the differences between a `map` and an `unordered_map`:

- A `map` requires an ordering function (the default is `<`) and yields an ordered sequence.
- A `unordered_map` requires an equality function (the default is `==`); it does not maintain an order among its elements.

Given a good hash function, an `unordered_map` is much faster than a `map` for large containers. However, the worst-case behavior of an `unordered_map` with a poor hash function is far worse than that of a `map`.

12.7 Allocators

By default, standard-library containers allocate space using `new`. Operators `new` and `delete` provide a general free store (also called dynamic memory or heap) that can hold objects of arbitrary size and user-controlled lifetime. This implies time and space overheads that can be eliminated in many special cases. Therefore, the standard-library containers offer the opportunity to install allocators with specific semantics where needed. This has been used to address a wide variety of concerns related to performance (e.g., pool allocators), security (allocators that clean-up memory as part of deletion), per-thread allocation, and non-uniform memory architectures (allocating in specific memories with pointer types to match). This is not the place to discuss these important, but very specialized and often advanced techniques. However, I will give one example motivated by a real-world problem for which a pool allocator was the solution.

An important, long-running system used an event queue (see §18.4) using `vectors` as events that were passed as `shared_ptrs`. That way, the last user of an event implicitly deleted it:

```
struct Event {
    vector<int> data = vector<int>(512);
};

list<shared_ptr<Event>> q;

void producer()
{
    for (int n = 0; n!=LOTS; ++n) {
        lock_guard lk {m};           // m is a mutex; see §18.3
        q.push_back(make_shared<Event>());
        cv.notify_one();           // cv is a condition_variable; see §18.4
    }
}
```

From a logical point of view this worked nicely. It is logically simple, so the code is robust and maintainable. Unfortunately, this led to massive fragmentation. After 100,000 events had been passed among 16 producers and 4 consumers, more than 6GB of memory had been consumed.

The traditional solution to fragmentation problems is to rewrite the code to use a pool allocator. A pool allocator is an allocator that manages objects of a single fixed size and allocates space for many objects at a time, rather than using individual allocations. Fortunately, C++ offers direct support for that. The pool allocator is defined in the `pmr` (“polymorphic memory resource”) subnamespace of `std`:

```
pmr::synchronized_pool_resource pool;           // make a pool

struct Event {
    vector<int> data = vector<int>(512,&pool);   // let Events use the pool
};

list<shared_ptr<Event>> q {&pool};             // let q use the pool
```



```

void producer()
{
    for (int n = 0; n!=LOTS; ++n) {
        scoped_lock lk {m}; // m is a mutex (§18.3)
        q.push_back(allocate_shared<Event,pmr::polymorphic_allocator<Event>>(&pool));
        cv.notify_one();
    }
}

```

Now, after 100,000 events had been passed among 16 producers and 4 consumers, less than 3MB of memory had been consumed. That's about a 2000-fold improvement! Naturally, the amount of memory actually in use (as opposed to memory wasted to fragmentation) is unchanged. After eliminating fragmentation, memory use was stable over time so the system could run for months.

Techniques like this have been applied with good effects from the earliest days of C++, but generally they required code to be rewritten to use specialized containers. Now, the standard containers optionally take allocator arguments. The default is for the containers to use `new` and `delete`. Other polymorphic memory resources include

- `unsynchronized_polymorphic_resource`; like `polymorphic_resource` but can only be used by one thread.
- `monotonic_polymorphic_resource`; a fast allocator that releases its memory only upon its destruction and can only be used by one thread.

A polymorphic resource must be derived from `memory_resource` and define members `allocate()`, `deallocate()`, and `is_equal()`. The idea is for users to build their own resources to tune code.

12.8 Container Overview

The standard library provides some of the most general and useful container types to allow the programmer to select a container that best serves the needs of an application:

Standard Container Summary	
<code>vector<T></code>	A variable-size vector (§12.2)
<code>list<T></code>	A doubly-linked list (§12.3)
<code>forward_list<T></code>	A singly-linked list
<code>deque<T></code>	A double-ended queue
<code>map<K,V></code>	An associative array (§12.5)
<code>multimap<K,V></code>	A map in which a key can occur many times
<code>unordered_map<K,V></code>	A map using a hashed lookup (§12.6)
<code>unordered_multimap<K,V></code>	A multimap using a hashed lookup
<code>set<T></code>	A set (a map with just a key and no value)
<code>multiset<T></code>	A set in which a value can occur many times
<code>unordered_set<T></code>	A set using a hashed lookup
<code>unordered_multiset<T></code>	A multiset using a hashed lookup

The unordered containers are optimized for lookup with a key (often a string); in other words, they are hash tables.

The containers are defined in namespace `std` and presented in headers `<vector>`, `<list>`, `<map>`, etc. (§9.3.4). In addition, the standard library provides container adaptors `queue<T>`, `stack<T>`, and `priority_queue<T>`. Look them up if you need them. The standard library also provides more specialized container-like types, such as `array<T,N>` (§15.3.1) and `bitset<N>` (§15.3.2).

The standard containers and their basic operations are designed to be similar from a notational point of view. Furthermore, the meanings of the operations are equivalent for the various containers. Basic operations apply to every kind of container for which they make sense and can be efficiently implemented:

Standard Container Operations (partial)	
<code>value_type</code>	The type of an element
<code>p=c.begin()</code> <code>p=c.end()</code>	<code>p</code> points to first element of <code>c</code> ; also <code>cbegin()</code> for an iterator to <code>const</code> <code>p</code> points to one-past-the-last element of <code>c</code> ; also <code>cend()</code> for an iterator to <code>const</code>
<code>k=c.size()</code> <code>c.empty()</code>	<code>k</code> is the number of elements in <code>c</code> Is <code>c</code> empty?
<code>k=c.capacity()</code> <code>c.reserve(k)</code> <code>c.resize(k)</code>	<code>k</code> is the number of elements that <code>c</code> can hold without a new allocation Increase the capacity to <code>k</code> ; if <code>k<=c.capacity()</code> , <code>c.reserve(k)</code> does nothing Make the number of elements <code>k</code> ; added elements have the default value <code>value_type{}</code>
<code>c[k]</code> <code>c.at(k)</code> <code>c.push_back(x)</code> <code>c.emplace_back(a)</code> <code>q=c.insert(p,x)</code> <code>q=c.erase(p)</code>	The <code>k</code> th element of <code>c</code> ; zero-based; no range guaranteed checking The <code>k</code> th element of <code>c</code> ; if out of range, throw <code>out_of_range</code> Add <code>x</code> at the end of <code>c</code> ; increase the size of <code>c</code> by one Add <code>value_type{a}</code> at the end of <code>c</code> ; increase the size of <code>c</code> by one Add <code>x</code> before <code>p</code> in <code>c</code> Remove element at <code>p</code> from <code>c</code>
<code>c=c2</code> <code>b=(c==c2)</code> <code>x=(c<=>c2)</code>	Assignment: copy all elements from <code>c2</code> to get <code>c==c2</code> Equality of all elements of <code>c</code> and <code>c2</code> ; <code>b==true</code> if equal Lexicographical order of <code>c</code> and <code>c2</code> : <code>x<0</code> if <code>c</code> is less than <code>c2</code> , <code>x==0</code> if equal, and <code>0<x</code> if greater than. <code>!=</code> , <code><</code> , <code><=</code> , <code>></code> , and <code>>=</code> are generated from <code><=></code>

This notational and semantic uniformity enables programmers to provide new container types that can be used in a very similar manner to the standard ones. The range-checked vector, `Vector` (§4.3, Chapter 5), is an example of that. The uniformity of container interfaces allows us to specify algorithms independently of individual container types. However, each has strengths and weaknesses. For example, subscripting and traversing a `vector` is cheap and easy. On the other hand, `vector` elements are moved to different locations when we insert or remove elements; `list` has exactly the opposite properties. Please note that a `vector` is usually more efficient than a `list` for short sequences of small elements (even for `insert()` and `erase()`). I recommend the standard-library `vector` as the default type for sequences of elements: you need a reason to choose another.

Consider the singly-linked list, `forward_list`, a container optimized for the empty sequence (§12.3). An empty `forward_list` occupies just one word, whereas an empty `vector` occupies three. Empty sequences, and sequences with only an element or two, are surprisingly common and useful.

An `emplace` operation, such as `emplace_back()` takes arguments for an element's constructor and builds the object in a newly allocated space in the container, rather than copying an object into the container. For example, for a `vector<pair<int,string>>` we could write:

```
v.push_back(pair{1,"copy or move"}); // make a pair and move it into v
v.emplace_back(1,"build in place"); // build a pair in v
```

For simple examples like this, optimizations can result in equivalent performance for both calls.

12.9 Advice

- [1] An STL container defines a sequence; §12.2.
- [2] STL containers are resource handles; §12.2, §12.3, §12.5, §12.6.
- [3] Use `vector` as your default container; §12.2, §12.8; [CG: SL.con.2].
- [4] For simple traversals of a container, use a range-`for` loop or a begin/end pair of iterators; §12.2, §12.3.
- [5] Use `reserve()` to avoid invalidating pointers and iterators to elements; §12.2.
- [6] Don't assume performance benefits from `reserve()` without measurement; §12.2.
- [7] Use `push_back()` or `resize()` on a container rather than `realloc()` on an array; §12.2.
- [8] Don't use iterators into a resized `vector`; §12.2 [CG: ES.65].
- [9] Do not assume that `[]` range checks; §12.2.
- [10] Use `at()` when you need guaranteed range checks; §12.2; [CG: SL.con.3].
- [11] Use range-`for` and standard-library algorithms for cost-free avoidance of range errors; §12.2.2.
- [12] Elements are copied into a container; §12.2.1.
- [13] To preserve polymorphic behavior of elements, store pointers (built-in or user-defined); §12.2.1.
- [14] Insertion operations, such as `insert()` and `push_back()`, are often surprisingly efficient on a `vector`; §12.3.
- [15] Use `forward_list` for sequences that are usually empty; §12.8.
- [16] When it comes to performance, don't trust your intuition: measure; §12.2.
- [17] A `map` is usually implemented as a red-black tree; §12.5.
- [18] An `unordered_map` is a hash table; §12.6.
- [19] Pass a container by reference and return a container by value; §12.2.
- [20] For a container, use the `()`-initializer syntax for sizes and the `{}`-initializer syntax for sequences of elements; §5.2.3, §12.2.
- [21] Prefer compact and contiguous data structures; §12.3.
- [22] A `list` is relatively expensive to traverse; §12.3.
- [23] Use unordered containers if you need fast lookup for large amounts of data; §12.6.
- [24] Use ordered containers (e.g., `map` and `set`) if you need to iterate over their elements in order; §12.5.
- [25] Use unordered containers (e.g., `unordered_map`) for element types with no natural order (i.e., no reasonable `<`); §12.5.
- [26] Use associative containers (e.g., `map` and `list`) when you need pointers to elements to be stable as the size of the container changes; §12.8.

- [27] Experiment to check that you have an acceptable hash function; §12.6.
- [28] A hash function obtained by combining standard hash functions for elements using the exclusive-or operator (^) is often good; §12.6.
- [29] Know your standard-library containers and prefer them to handcrafted data structures; §12.8.
- [30] If your application is suffering performance problems related to memory, minimize free store use and/or consider using a specialized allocator; §12.7.

This page intentionally left blank

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 169
not-equal operator 7

"

literal operator 85
string literal 3

\$, [regex](#) 131

%

modulus operator 7
remainder operator 7

%=, operator 7

&

address-of operator 11
pointer 11
reference to 12

&&, rvalue reference 77

,

char 8
digit separator 6

(, [regex](#) 131

()), call operator 94

(?: pattern 134

), [regex](#) 131

*

contents-of operator 11
iterator 179
multiply operator 7
operator 220
pointer 11
[regex](#) 131

*=, scaling operator 7

*? lazy 132

+

iterator 192
plus operator 7
[regex](#) 131
[string](#) concatenation 125

++

increment operator 7
iterator 179, 192

+=

iterator 192
operator 7
[string](#) append 126

+? lazy 132

-

iterator 192
minus operator 7

--

decrement operator 7
iterator 192

- =, iterator 192
- >
 - member access 23
 - operator 220
 - return type 40
- - member access 23
 - regex 131
- ..., variadic **template** 114
- /, divide operator 7
- // comment 2
- /=, scaling operator 7
- :**public** 61
- <<
 - output operator 3, 84
 - output **ostream** 138
- <=
 - container 169
 - less-than-or-equal operator 7
- <=>
 - container 169
 - spaceship operator 81
- <
 - container 169
 - less-than operator 7
- =
 - 0 60
 - and == 7
 - assignment 17
 - auto** 8
 - container 169
 - default 56
 - initializer 7
 - initializer narrowing 8
 - string** assignment 126
- ==
 - = and 7
 - container 169
 - equal operator 7
 - iterator 192
 - string** 126
- >
 - container 169
 - greater-than operator 7
- >=
 - container 169
 - greater-than-or-equal operator 7
- >>
 - input **istream** 139
 - input operator 84
- ?, **regex** 131
- ?: operator 82
- ?? lazy 132
- [, **regex** 131
- [&] 95
- [=] 95

- [[[]] attribute syntax 263
- []
 - array 11
 - array** 203
 - auto** 41
 - iterator 192
 - string** 126
 - subscript operator 25
 - subscripting 169
- \, backslash 3
-], **regex** 131
- ^, **regex** 131
- {, **regex** 131
- { }
 - format()** argument 144
 - grouping 2
 - initializer 8
- { }? lazy 132
- |
 - pipeline 188
 - regex** 131
- }, **regex** 131
- ~, destructor 57
- 0
 - = 60
 - nullptr NULL** 14

A

- :%A, **format()** 146
- abort()** 225
- abs()** 228
- abstract **class** 60
- access
 - ., member 23
 - >, member 23
- accumulate()** 229
- acquisition RAII, resource 197
- Ada 208
- adaptor
 - function 216
 - lambda as 216
 - range** 187
- address, memory 16
- address-of operator **&** 11
- adjacent_difference()** 229
- aims, C++11 261
- Alexander Fraser 259
- algorithm 173
 - container 175
 - lifting 113
 - numerical 229
 - parallel 183
 - standard library 181
- <**algorithm**> 123, 182
- alias

- template 100
 - type 234
 - using 100
 - alignas 263
 - alignof 263
 - allocation 57
 - allocator **new**, container 167
 - almost container 201
 - alnum, **regex** 133
 - alpha, **regex** 133
 - [[[:alpha:]] letter 133
 - alternatives, error handling 47
 - Annemarie 127
 - ANSI C++ 260
 - any 211
 - Anya 208
 - append +=, **string** 126
 - argument
 - {}, **format()** 144
 - constrained 89
 - constrained **template** 90
 - default function 38
 - default **template** 108
 - function 37
 - lambda as 96
 - order, **format()** 145
 - passing, function 72
 - type 90
 - value 90
 - arithmetic
 - conversions, usual 7
 - operator 7
 - vector 233
 - Arithmetic example 108, 219
 - ARM 260
 - array
 - [] 11
 - array vs. 203
 - array 202
 - [] 203
 - data()** 203
 - initialize 202
 - size()** 203
 - vs. array 203
 - vs. **vector** 203
 - <array> 123
 - asin() 228
 - assembler 257
 - assert(), assertion 49
 - assertion
 - assert() 49
 - expect() 48
 - static_assert 50
 - assignable_from, concept 190
 - assignment
 - = 17
 - =, **string** 126
 - copy 72, 75
 - initialization and 18
 - move 72, 78
 - assignment-to-string-literal, removed 267
 - associate type 222
 - associative array – see **map**
 - async() launch 247
 - at() 161
 - atan() 228
 - atan2() 228
 - atexit() 225
 - atomic 243
 - AT&T Bell Laboratories 260
 - attribute
 - [[carries_dependency]] 263
 - [[deprecated]] 264
 - [[fallthrough]] 264
 - [[likely]] 265
 - [[maybe_unused]] 264
 - [[nodiscard]] 98, 264
 - [[noreturn]] 263
 - [[no_unique_address]] 265
 - syntax, [[]] 263
 - [[unlikely]] 265
 - auto
 - [] 41
 - = 8
 - concept and 110
 - return type 40
 - auto_ptr, removed 267
- ## B
- :%B, **format()** 146
 - :b, **format()** 145
 - back_inserter() 175
 - backslash \ 3
 - bad_variant_access 210
 - base
 - and derived **class** 61
 - destructor for 65
 - basic_string 128
 - BCPL 269
 - begin() 83, 163, 169, 175
 - beginner, book for 1
 - Bell Laboratories, AT&T 260
 - beta() 228
 - bibliography 271
 - bidirectional_iterator, concept 192
 - bidirectional_range, concept 193
 - binary search 182
 - binding, structured 41
 - bit manipulation 224
 - bit_cast 224
 - bit-field, **bitset** and 204

- bitset 204
 - and bit-field 204
 - and `enum` 204
 - blank, `regex` 133
 - block
 - as function body, `try` 161
 - `try` 44
 - body, function 2
 - book for beginner 1
 - bool 6
 - Boolean, concept 191
 - bounded_range, concept 193
 - break 15
 - Brian Kernighan 259
 - buffer overrun 200
 - built-in type 21
 - byte, `std::byte` 224
- ## C
- C 257
 - and C++ compatibility 268
 - Classic 269
 - difference from 268
 - K&R 269
 - style 269
 - with Classes 256
 - with Classes language features 258
 - with Classes standard library 259
 - C++
 - ANSI 260
 - compatibility, C and 268
 - Core Guidelines 262
 - core language 2
 - evolution 256
 - history 255
 - ISO 260
 - meaning 257
 - model 262
 - modern 262
 - pronunciation 257
 - standard, ISO 2
 - standard library 2
 - standardization 260
 - style 269
 - timeline 256
 - use 262
 - users, number of 262
 - C++03 260
 - C++0x, C++11 257, 260
 - C++11
 - aims 261
 - C++0x 257, 260
 - language features 263
 - library components 265
 - C++14 261
 - language features 264
 - library components 266
 - C++17 261
 - language features 264
 - library components 266
 - C++20 1, 185, 261
 - concept 104
 - language features 265
 - library components 266
 - module 33
 - C++98 260
 - standard library 259
 - C11 268
 - C+23, `spanstream` 149
 - C89 and C99 268
 - C99, C89 and 268
 - calendar 214
 - call operator () 94
 - callback 217
 - `capacity()` 159, 169
 - capture list 95
 - `[[carries_dependency]]` attribute 263
 - cast 59
 - catch
 - clause 44
 - every exception 161
 - `catch(...)` 161
 - `cbegin()` 83
 - `ceil()` 228
 - `cend()` 83
 - `cerr` 138
 - `char` 6
 - , 8
 - character sets, multiple 128
 - check
 - compile-time 50
 - run-time 48
 - `Checked_iter` example 174
 - checking
 - cost of range 162
 - template definition 109
 - `chrono::namespace` 214
 - `<chrono>` 123, 214, 243
 - `cin` 139
 - class 23, 54
 - abstract 60
 - and `struct` 25
 - base and derived 61
 - concrete 54
 - hierarchy 63
 - interface 23
 - member 23
 - scope 9
 - template 88
 - Classic C 269
 - clause, `requires` 105

- clear(), `iostream` 141
- C-library header 123
- clock 214
- clock timing 243
- <cmath> 123, 228
- cntrl, `regex` 133
- code complexity, function and 5
- comment, // 2
- common_reference_with, concept 190
- common_type_t 190, 221
- common_view 187
- common_with, concept 190
- communication, task 245
- comparison operator 7, 81
- compatibility, C and C++ 268
- compilation
 - model, `template` 117
 - separate 30
- compiler 2
- compile-time
 - check 50
 - computation 218
 - evaluation 10
 - if 101
- complex 55, 230
- <complex> 123, 228, 230
- complexity, function and code 5
- components
 - C++11 library 265
 - C++14 library 266
 - C++17 library 266
 - C++20 library 266
- computation, compile-time 218
- concatenation +, `string` 125
- concept 89
 - assignable_from 190
 - bidirectional_iterator 192
 - bidirectional_range 193
 - Boolean 191
 - bounded_range 193
 - common_reference_with 190
 - common_with 190
 - constructible_from 191
 - contiguous_iterator 192
 - contiguous_range 193
 - convertible_to 190
 - copy_constructible 191
 - default_initializable 191
 - derived_from 190
 - destructible 191
 - equality_comparable 190
 - equality_comparable_with 190
 - equivalence_relation 191
 - floating_point 190
 - forward_iterator 192
 - forward_range 193
 - input_iterator 192
 - input_or_output_iterator 192
 - input_range 193
 - integral 190
 - invocable 191
 - mergeable 192
 - mopyable 191
 - movable 191
 - move_constructible 191
 - output_iterator 192
 - output_range 193
 - permutable 192
 - predicate 191
 - random_access_iterator 192
 - random_access_range 193
 - range 185
 - range 193
 - regular 191
 - regular_invocable 191
 - relation 191
 - same_as 190
 - semiregular 191
 - sentinel_for 192
 - signed_integral 190
 - sized_range 193
 - sized_sentinel_for 192
 - sortable 192
 - strict_weak_order 191
 - swappable 190
 - swappable_with 190
 - three_way_comparable 190
 - three_way_comparable_with 190
 - totally_ordered 190
 - totally_ordered_with 190
 - unsigned_integral 190
 - view 193
- concept 104
 - and auto 110
 - and type 111
 - and variable 111
 - based overloading 106
 - C++20 104
 - definition of 107
 - in <concepts> 190
 - in <iterator> 190
 - in <ranges> 190
 - static_assert and 108
 - use 104
- concepts
 - iterator 192
 - range 193
 - type 190
- <concepts> 123
 - concept in 190
- concrete
 - class 54

- type 54
 - concurrency 237
 - condition, declaration in 67
 - `condition_variable` 244
 - `notify_one()` 245
 - `wait()` 244
 - `<condition_variable>` 244
 - `const`
 - immutability 10
 - member function 56
 - constant
 - expression 10
 - mathematical 234
 - `const_cast` 59
 - `constexpr`, immutability 10
 - `constexpr`
 - function 10
 - `if` 101
 - immutability 10
 - `const_iterator` 179
 - constrained
 - argument 89
 - `template` 90
 - `template` argument 90
 - `constructible_from`, concept 191
 - construction, order of 67
 - constructor 24
 - and destructor 258
 - copy 72, 75
 - default 56
 - delegating 264
 - `explicit` 73
 - inherited 161
 - inheriting 264
 - initializer-list 58
 - invariant and 45
 - move 72, 77
 - `consumer()` example, `producer()` 244
 - container 57, 88, 157
 - `>=` 169
 - `==` 169
 - `>` 169
 - `=` 169
 - `<` 169
 - `<=` 169
 - `<=>` 169
 - `!=` 169
 - algorithm 175
 - allocator `new` 167
 - almost 201
 - object in 160
 - operation 83
 - overview 168
 - `return` 176
 - specialized 201
 - standard library 168
 - contents-of operator * 11
 - `contiguous_iterator`, concept 192
 - `contiguous_range`, concept 193
 - conventional operation 81
 - conversion 73
 - explicit type 59
 - narrowing 8
 - conversions, usual arithmetic 7
 - `convertible_to`, concept 190
 - cooperative multitasking example 251
 - copy 74
 - assignment 72, 75
 - constructor 72, 75
 - cost of 76
 - elision 40, 72
 - elision 78
 - memberwise 72
 - `copy()` 182
 - `copy_constructible`, concept 191
 - `copy_if()` 182
 - Core Guidelines, C++ 262
 - core language, C++ 2
 - `co_return` 250
 - coroutine 250, 259
 - generator 250
 - `promise_type` 253
 - `cos()` 228
 - `cosh()` 228
 - cost
 - of copy 76
 - of range checking 162
 - `count()` 182
 - `count_if()` 181–182
 - Courtney 208
 - `cout` 138
 - output 3
 - `co_yield` 250
 - `<cstdlib>` 123
 - C-style
 - error handling 228
 - I/O 149
 - string 13
 - CTAD 93
- ## D
- `:d, format()` 145
 - `\d, regex` 133
 - `\D, regex` 133
 - `d, regex` 133
 - dangling pointer 196
 - data
 - member 23
 - race 239
 - `data(), array` 203
 - D&E 256

deadlock 242
 deallocation 57
 debugging [template](#) 113
 declaration 6
 function 4
 in condition 67
 interface 29
 [using](#) 36
 declarator operator 12
[decltype](#) 263
[decltype\(\)](#) 218
 decrement operator `--` 7
 deduction
 guide 210
 guide 92
 [return](#) type 40
 default
 = 56
 constructor 56
 function argument 38
 member initializer 74
 operation 72
 [template](#) argument 108
[=default](#) 72
[defaultfloat](#) 143
[default_initializable](#), concept 191
 definition
 checking, [template](#) 109
 implementation 30
 of [concept](#) 107
 delegating constructor 264
[=delete](#) 73
[delete](#)
 naked 58
 operator 57
[delete\[\]](#), operator 57
 Dennis Ritchie 259
 deprecated
 feature 267
 [stringstream](#) 148, 267
[\[\[deprecated\]\]](#) attribute 264
[deque](#) 168
 derived [class](#), base and 61
[derived_from](#), concept 190
[destructible](#), concept 191
 destruction, order of 67
 destructor 57, 72
 ~ 57
 constructor and 258
 for base 65
 for member 65
 [virtual](#) 65
 dictionary – see [map](#)
 difference from C
 digit
 [\[\[:digit:\]\]](#) 133

 separator ' 6
[digit, regex](#) 133
[\[\[:digit:\]\]](#) digit 133
[directive, using](#) 36
[directory_iterator](#) 151–152
 distribution, random 231
 divide operator / 7
 domain error 228
[double](#) 6
 double-checked locking 243
 Doug McIlroy 259
[drop_view](#) 187
 duck typing 117
[duration](#) 214
[duration_cast](#) 214
 dynamic memory 57
[dynamic_cast](#) 67
 is instance of 67
 is kind of 67

E

[e](#) 234
[EDOM](#) macro 228
 element requirements 160
 elision, copy 40, 72
[emplace_back\(\)](#) 169
[empty\(\)](#) 169
[enable_if](#) 221
[enable_if_t](#) 221
 encapsulation 72
[end\(\)](#) 83, 163, 169, 175
[endl](#) 154
 engine, random 231
[enum](#)
 [bitset](#) and 204
 [class](#) enumeration 26
 enumeration 25
 [using](#) 26
 enumeration
 [enum](#) 25
 [enum class](#) 26
 equal operator `==` 7
 equality preserving 192
[Equality_comparable](#) example 108
[equality_comparable](#), concept 190
[equality_comparable_with](#), concept 190
[equal_range\(\)](#) 182
[equivalence_relation](#), concept 191
[ERANGE](#) macro 228
[erase\(\)](#) 163, 169
[errno](#) 228
 error
 domain 228
 handling 43
 handling alternatives 47

- handling, C-style 228
 - range 200, 228
 - recovery 47
 - run-time 44
- error-code, exception vs 47
- `error_code` 153
- essential operation 72
- evaluation
 - compile-time 10
 - order of 8
- event driven simulation example 251
- evolution, C++ 256
- Example, `expect()` 48
- example
 - `Arithmetic` 108, 219
 - `Checked_iter` 174
 - cooperative multitasking 251
 - `Equality_comparable` 108
 - event driven simulation 251
 - `finally()` 98
 - `find_all()` 176
 - Hello, World! 2
 - `Number` 108
 - `producer() consumer()` 244
 - `Rand_int` 231
 - `Sentinel` 193
 - `Sequence` 109
 - `sum()` 104
 - `task` 253
 - `tau` 235
 - `Value_type` 109
 - `Vec` 161
 - `Vector` 22–23, 29, 33–34, 57–58, 73, 75, 77,
- 88–89, 91–92, 100
- exception 44
 - and `main()` 161
 - `catch` every 161
 - specification, removed 267
 - vs error-code 47
- `exclusive_scan()` 229
- execution policy 183
- `exists()` 150
- `exit`, program 225
- `exit()` termination 225
- `exp()` 228
- `exp2()` 228
- `expect()`
 - assertion 48
 - Example 48
- explicit type conversion 59
- `explicit` constructor 73
- `exponential_distribution` 231
- `export`
 - `module` 33
 - removed 267
- expression

- constant 10
- fold 115
- lambda 95
- `requires` 106
- `extension()` 152
- extern template 264

F

- `fabs()` 228
- facilities, standard library 120
- `[[fallthrough]]` attribute 264
- feature
 - deprecated 267
 - removed 267
- features
 - C with Classes language 258
 - C++11 language 263
 - C++14 language 264
 - C++17 language 264
 - C++20 language 265
- file
 - header 31
 - open a 151
 - system operation 153
 - type 154
- `file_name()`, `source_location` 222
- `<filesystem>` 150
- `filesystem_error` 153
- `filter()` 189
- `filter_view` 186
- `final` 264
- `Final_action` 98
- `finally()` example 98
- `find()` 175, 182
- `find_all()` example 176
- `find_if()` 181–182
- `fixed` 143
- floating-point literal 6
- `floating_point`, concept 190
- `floor()` 228
- fold expression 115
- `for`
 - statement 12
 - statement, range 12
- format, output 143–144
- `format()`
 - `:%A` 146
 - argument {} 144
 - argument order 145
 - `:%B` 146
 - `:b` 145
 - `:d` 145
 - `:o` 145
 - precision 145
 - `:x` 145

- `<format>` 123, 144
- `forward()` 116, 223
- forwarding, perfect 224
- `forward_iterator`, concept 192
- `forward_list` 168
 - singly-linked list 164
- `<forward_list>` 123
- `forward_range`, concept 193
- Fraser, Alexander 259
- free store 57
- `friend` 193
- `<fstream>` 123, 147
- `__func__` 264
- function 2
 - adaptor 216
 - and code complexity 5
 - argument 37
 - argument, default 38
 - argument passing 72
 - body 2
 - body, try block as 161
 - `const` member 56
 - `constexpr` 10
 - declaration 4
 - implementation of `virtual` 62
 - mathematical 228
 - member 23
 - name 5
 - object 94
 - overloading 5
 - `return` value 37
 - `template` 93
 - type 217
 - value `return` 72
- `function` 217
 - and `nullptr` 217
- `<functional>` 123
- `function_name()`, `source_location` 222
- fundamental type 6
- `future`
 - and `promise` 245
 - member `get()` 245
- `<future>` 123, 245

G

- garbage collection 79
- Gavin 208
- `gcd()` 229
- generator
 - coroutine 250
 - type 221
- generic programming 103, 112, 258
- `get<>()`
 - by index 207
 - by type 207

- `get()`, `future` member 245
- `getline()` 140
- `graph`, `regex` 133
- greater-than operator `>` 7
- greater-than-or-equal operator `>=` 7
- greedy match 132, 135
- grouping, `{}` 2
- guide, deduction 92
- Guidelines, C++ Core 262

H

- half-open sequence 182
- `handle` 24, 58
 - resource 75, 198
- hardware, mapping to 16
- hash table 165
- `hash<>`, `unordered_map` 84
- header
 - C-library 123
 - file 31
 - problems 32
 - standard library 121, 123
 - unit 279
- heap 57
- `Hello, World!` example 2
- `hexfloat` 143
- hierarchy
 - `class` 63
 - navigation 67
- history, C++ 255
- HOPL 256

I

- `if`
 - compile-time 101
 - `constexpr` 101
 - statement 14
- `ifstream` 147
- immutability
 - `const` 10
 - `constexpr` 10
 - `constexpr` 10
- implementation
 - definition 30
 - inheritance 66
 - iterator 178
 - of `virtual` function 62
 - `push_back()` 159
 - `string` 127
- `import` 3
 - and `#include` 277
 - `#include` and 34
 - module 33

- in-class member initialization 264
 - `#include` 3, 31
 - and `import` 34
 - `import` and 277
 - `inclusive_scan()` 229
 - incompatibility, `void*` 270
 - increment operator `++` 7
 - index, `get<>()` by 207
 - infinite `range` 185
 - inheritance 61
 - implementation 66
 - interface 65
 - multiple 259
 - inherited constructor 161
 - inheriting constructor 264
 - initialization
 - and assignment 18
 - in-class member 264
 - initialize 58
 - `array` 202
 - initializer
 - `=` 7
 - `{}` 8
 - default member 74
 - lambda as 97–98
 - narrowing, `=` 8
 - initializer-list constructor 58
 - `initializer_list` 58
 - `inline` 55
 - `namespace` 264
 - inlining 55
 - `inner_product()` 229
 - input
 - `istream >>` 139
 - of user-defined type 141
 - operator `>>` 84
 - `string` 140
 - `input_iterator`, concept 192
 - `input_or_output_iterator`, concept 192
 - `input_range`, concept 193
 - `insert()` 163, 169
 - instantiation 89
 - instantiation time, `template` 117
 - instruction, machine 16
 - `int` 6
 - output bits of 204
 - `int32_t` 234
 - integer literal 6
 - `integral`, concept 190
 - interface
 - `class` 23
 - declaration 29
 - inheritance 65
 - invalidation 159
 - invariant 45
 - and constructor 45
 - `invocable`, concept 191
 - `invoke_result_t` 221
 - I/O 138
 - C-style 149
 - iterator and 179
 - state 141
 - `<iomanip>` 143
 - `<ios>` 123, 143
 - `iostream`
 - `clear()` 141
 - kinds of 146
 - `setstate()` 141
 - `unset()` 141
 - `<iostream>` 3, 123
 - `iota()` 229
 - `is`
 - instance of, `dynamic_cast` 67
 - kind of, `dynamic_cast` 67
 - `is_arithmetic_v` 218
 - `is_base_of_v` 218
 - `is_constructible_v` 218
 - `is_directory()` 151
 - `is_integral_v` 218
 - ISO
 - C++ 260
 - C++ standard 2
 - ISO-14882 260
 - `is_same_of_v` 218
 - `istream` 138
 - `>>`, input 139
 - `<istream>` 139
 - `istream_iterator` 179
 - `istringstream` 147
 - iterator 83–84, 175
 - `==` 192
 - `+` 192
 - `--` 192
 - `+=` 192
 - `-=` 192
 - `-` 192
 - `++` 179, 192
 - `*` 179
 - `[]` 192
 - and I/O 179
 - concepts 192
 - implementation 178
 - `iterator` 163, 179
 - `<iterator>`, concept in 190
 - `iterator_t` 109
 - `iter_value_t` 109
- ## J
- `join()`, `thread` 238
 - `join_view` 187

K

Kernighan, Brian 259
 key and value 164
 kinds of `iostream` 146
 K&R C 269

L

`\l`, `regex` 133
`\L`, `regex` 133
 lambda
 as adaptor 216
 as argument 96
 as initializer 97–98
 expression 95
 language
 and library 119
 features, C with Classes 258
 features, C++11 263
 features, C++14 264
 features, C++17 264
 features, C++20 265
 launch, `async()` 247
 lazy
 +? 132
 {}? 132
 ?? 132
 *? 132
 match 132, 135
`lcm()` 229
 leak, resource 67, 78, 197
 less-than operator < 7
 less-than-or-equal operator <= 7
 letter, `[[alpha:]]` 133
 library
 algorithm, standard 181
 C with Classes standard 259
 C++98 standard 259
 components, C++11 265
 components, C++14 266
 components, C++17 266
 components, C++20 266
 container, standard 168
 facilities, standard 120
 language and 119
 non-standard 119
 standard 119
 lifetime, scope and 9
 lifting algorithm 113
`[[likely]]` attribute 265
`<limits>` 217, 234
`line()`, `source_location` 222
 linker 2
 list
 capture 95

`forward_list` singly-linked 164
`list` 162, 168
 literal
 ", string 3
 floating-point 6
 integer 6
 operator " 85
 raw string 130
 suffix, `s` 127
 suffix, `sv` 129
 type of string 127
 UDL, user-defined 84
 user-defined 264
 literals
 `string_literals` 127
 `string_view_literals` 129
`ln10` 234
`ln2` 234
 local scope 9
 lock, reader-writer 242
 locking, double-checked 243
`log()` 228
`log10()` 228
`log2()` 228
`log2e` 234
`long long` 263
`lower`, `regex` 133

M

machine instruction 16
 macro
 EDOM 228
 ERANGE 228
`main()` 2
 exception and 161
`make_pair()` 207
`make_shared()` 199
`make_unique()` 199
 management, resource 78, 197
 manipulation, bit 224
 manipulator 143
`map` 164, 168
 and `unordered_map` 166
`<map>` 123
 mapped type, value 164
 mapping to hardware 16
 match
 greedy 132, 135
 lazy 132, 135
 mathematical
 constant 234
 function 228
 function, standard 228
 functions, special 228
`<math.h>` 228

- Max Munch rule 132
 - `[[maybe_unused]]` attribute 264
 - McIlroy, Doug 259
 - meaning, C++ 257
 - member
 - access . 23
 - access -> 23
 - `class` 23
 - data 23
 - destructor for 65
 - function 23
 - function, `const` 56
 - initialization, in-class 264
 - initializer, default 74
 - memberwise copy 72
 - `mem_fn()` 217
 - memory 79
 - address 16
 - dynamic 57
 - resource, polymorphic 167
 - safety 196
 - `<memory>` 123, 197, 199
 - `merge()` 182
 - `mergeable`, concept 192
 - `midpoint()` 229
 - minus operator - 7
 - model
 - C++ 262
 - `template` compilation 117
 - modern C++ 262
 - modularity 29
 - module
 - C++20 33
 - `export` 33
 - `import` 33
 - standard library 123
 - `std` 34, 277
 - `std.compat` 277
 - modulus operator % 7
 - `month` 214
 - `mopyable`, concept 191
 - `movable`, concept 191
 - `move` 72, 77
 - assignment 72, 78
 - constructor 72, 77
 - `move()` 78, 182, 223
 - `move_constructible`, concept 191
 - moved-from
 - object 78
 - state 224
 - move-only type 223
 - multi-line pattern 131
 - `multimap` 168
 - multiple
 - character sets 128
 - inheritance 259
 - `return` values 41
 - multiply operator * 7
 - `multiset` 168
 - `mutex` 241
 - `<mutex>` 241
- ## N
- `\n`, newline 3
 - naked
 - `delete` 58
 - `new` 58
 - name, function 5
 - namespace scope 9
 - `namespace` 35
 - `chrono` 214
 - `inline` 264
 - `pmr` 167
 - `std` 3, 36, 121
 - `views` 188
 - narrowing
 - = initializer 8
 - conversion 8
 - navigation, hierarchy 67
 - `new`
 - container allocator 167
 - naked 58
 - operator 57
 - newline `\n` 3
 - Nicholas 126
 - `[[nodiscard]]` attribute 98, 264
 - `noexcept` 50
 - `noexcept()` 263
 - nonhomogeneous operation 108
 - non-memory resource 79
 - non-standard library 119
 - Norah 208
 - `[[noreturn]]` attribute 263
 - `normal_distribution` 231
 - notation
 - regular expression 131
 - `template` 105
 - not-equal operator != 7
 - `notify_one()`, `condition_variable` 245
 - `[[no_unique_address]]` attribute 265
 - `now()` 214
 - `NULL 0`, `nullptr` 14
 - `nullptr` 13
 - function and 217
 - `NULL 0` 14
 - number
 - of C++ users 262
 - random 231
 - `Number` 108
 - example 108
 - `<numbers>` 234

`<numeric>` 229
 numerical algorithm 229
`numeric_limits` 234

O

`:o, format()` 145
 object 6
 function 94
 in container 160
 moved-from 78
 object-oriented programming 63, 258
`ofstream` 147
 open a file 151
 operation
 container 83
 conventional 81
 default 72
 essential 72
 file system 153
 nonhomogeneous 108
 path 152
 operator
 `->` 220
 `+=` 7
 `%=` 7
 `*` 220
 `?:` 82
 `&`, address-of 11
 `()`, call 94
 `*`, contents-of 11
 `--`, decrement 7
 `/`, divide 7
 `==`, equal 7
 `>`, greater-than 7
 `>=`, greater-than-or-equal 7
 `++`, increment 7
 `>>`, input 84
 `<`, less-than 7
 `<=`, less-than-or-equal 7
 `"`, literal 85
 `-`, minus 7
 `%`, modulus 7
 `*`, multiply 7
 `!=`, not-equal 7
 `<<`, output 3, 84
 `+`, plus 7
 `%`, remainder 7
 `*=`, scaling 7
 `/=`, scaling 7
 `<=>`, spaceship 81
 `[]`, subscript 25
 arithmetic 7
 comparison 7, 81
 declarator 12
 `delete[]` 57

`delete` 57
`new` 57
 overloaded 57
 overloading 80
 relational 81
 user-defined 57
 optimization, short-string 127
`optional` 210
 order
 `format()` argument 145
 of construction 67
 of destruction 67
 of evaluation 8
 of, `public private` 23
`ostream` 138
 `<<`, output 138
`<ostream>` 138
`ostream_iterator` 179
`ostringstream` 147
`out_of_range` 161
 output 138
 bits of `int` 204
 `cout` 3
 format 143–144
 of user-defined type 141
 operator `<<` 3, 84
 `ostream <<` 138
 string 140
`output_iterator`, concept 192
`output_range`, concept 193
 overloaded operator 57
`overloaded()` 210
 overloading
 concept based 106
 function 5
 operator 80
`override` 61
 overrun, buffer 200
 overview, container 168
 ownership 197
 owning 196

P

`packaged_task thread` 247
`par` 183
 parallel algorithm 183
 parameterized type 88
`partial_sum()` 229
`par_unseq` 183
 passing data to task 239
`path` 151
 operation 152
 pattern 130
 `(?:` 134
 multi-line 131

- perfect forwarding 224
- `permutable`, concept 192
- `phone_book` example 158
- `pi` 234
- pipeline | 188
- plus operator + 7
- `pmr, namespace` 167
- pointer 17
 - & 11
 - * 11
 - dangling 196
 - smart 84, 197, 220
- policy, execution 183
- polymorphic
 - memory resource 167
 - type 60
- `pow()` 228
- precision, `format()` 145
- `precision()` 143
- precondition 45
- predicate 94, 181
 - type 218
- `predicate`, concept 191
- `print, regex` 133
- `printf()` 149
- `private` order of, `public` 23
- problems, header 32
- procedural programming 2
- `producer()` `consumer()` example 244
- program 2
 - exit 225
- programming
 - generic 103, 112, 258
 - object-oriented 63, 258
 - procedural 2
- `promise`
 - `future` and 245
 - member `set_exception()` 245
 - member `set_value()` 245
- `promise_type`, coroutine 253
- pronunciation, C++ 257
- `ptrdiff_t` 234
- `public private` order of 23
- `punct, regex` 133
- pure `virtual` 60
- purpose, `template` 103
- `push_back()` 58, 163, 169
 - implementation 159
- `push_front()` 163

Q

- `quick_exit()` termination 225

R

- `R*` 130
- race, data 239
- RAII 58, 98, 259
 - and resource management 45
 - and `try-block` 48
 - and `try-statement` 45
 - resource acquisition 197
 - `scoped_lock` and 241–242
- `Rand_int` example 231
- random
 - distribution 231
 - engine 231
 - number 231
- `<random>` 123, 231
- `random_access_iterator`, concept 192
- `random_access_range`, concept 193
- `random_device` 233
- `random_engine seed()`
- range
 - checking, cost of 162
 - checking `Vec` 161
 - concepts 193
 - error 200, 228
 - `for` statement 12
- `range`
 - adaptor 187
 - concept 193
 - concept 185
 - infinite 185
- range-checking, `span` 200
- range-`for`, `span` and 200
- `<ranges>` 123, 185
 - concept in 190
- `range_value_t` 109
- raw string literal 130
- reader-writer lock 242
- recovery, error 47
- `recursive_directory_iterator` 152
- `reduce()` 229
- reference 18
 - &&, rvalue 77
 - rvalue 78
 - to & 12
- `regex`
 - * 131
 - [131
 - + 131
 - . 131
 - ? 131
 - ^ 131
 -] 131
 -) 131
 - (131
 - \$ 131
 - { 131

- } 131
- | 131
- alnum 133
- alpha 133
- blank 133
- cntrl 133
- \D 133
- \d 133
- d 133
- digit 133
- graph 133
- \L 133
- \l 133
- lower 133
- print 133
- punct 133
- regular expression 130
- repetition 132
- s 133
- \s 133
- \S 133
- space 133
- \u 133
- \U 133
- upper 133
- \W 133
- \w 133
- w 133
- xdigit 133
- <regex> 123, 130
 - regular expression 130
- regex_iterator 135
- regex_search 130
- register, removed 267
- regular
 - expression notation 131
 - expression `regex` 130
 - expression `<regex>` 130
- regular, concept 191
- regular_invocable, concept 191
- reinterpret_cast 59
- relation, concept 191
- relational operator 81
- remainder operator `%` 7
- remove_const_t 221
- removed
 - assignment-to-string-literal 267
 - auto_ptr 267
 - exception specification 267
 - export 267
 - feature 267
 - register 267
- repetition, `regex` 132
- replace() 182
 - string 126
- replace_if() 182

- request_stop() 249
- requirement, `template` 104
- requirements 105
 - element 160
- requires
 - clause 105
 - expression 106
- reserve() 159, 169
- resize() 169
- resource
 - acquisition RAII 197
 - handle 75, 198
 - leak 67, 78, 197
 - management 78, 197
 - management, RAII and 45
 - non-memory 79
 - retention 79
 - safe 262
 - safety 78
- rethrow 46
- return
 - container 176
 - function value 72
 - type `->` 40
 - type `auto` 40
 - type deduction 40
 - type, suffix 263
 - type suffix 40
 - type, `void` 4
 - value, function 37
 - values, multiple 41
- returning results from task 240
- reverse_view 187
- rieman_zeta() 228
- Ritchie, Dennis 259
- rule
 - Max Munch 132
 - of zero 73
- run-time
 - check 48
 - error 44
- rvalue
 - reference 78
 - reference `&&` 77

S

- s literal suffix 127
- \S, `regex` 133
- \s, `regex` 133
- s, `regex` 133
- safe
 - resource 262
 - type 262
- safety
 - memory 196

- resource 78
- `same_as`, concept 190
- saving space 27
- scaling
 - operator `/=` 7
 - operator `*=` 7
- `scanf()` 149
- scientific 143
- scope
 - and lifetime 9
 - class 9
 - local 9
 - namespace 9
- `scoped_lock` 197
 - and RAI 241–242
 - `unique_lock` and 245
- `scoped_lock()` 242
- `scope_exit` 98
- search, binary 182
- `seed()`, `random_engine`
- semiregular, concept 191
- Sentinel example 193
- `sentinel_for`, concept 192
- separate compilation 30
- sequence 174
 - half-open 182
- Sequence example 109
- `set` 168
- `<set>` 123
- `set_exception()`, `promise` member 245
- `setstate()`, `iostream` 141
- `set_value()`, `promise` member 245
- SFINAE 221
- `shared_lock` 242
- `shared_mutex` 242
- `shared_ptr` 197
- sharing data task 241
- short-string optimization 127
- sightseeing tour
- `signed_integral`, concept 190
- SIMD 183
- Simula 251, 255
- `sin()` 228
- singly-linked list, `forward_list` 164
- `sinh()` 228
- size of type 6
- `size()` 83, 169
 - array 203
- `sized_range`, concept 193
- `sized_sentinel_for`, concept 192
- `sizeof` 6
- `sizeof()` 217
- `size_t` 100, 234
- smart pointer 84, 197, 220
- `smatch` 130
- `sort()` 173, 182
- `sortable`, concept 192
- `source_location`
 - `file_name()` 222
 - `function_name()` 222
 - `line()` 222
- space, saving 27
- `space, regex` 133
- spaceship operator `<=>` 81
- `span`
 - and range-`for` 200
 - range-checking 200
 - `string_view` and 200
- `spanstream` C++23 149
- special mathematical functions 228
- specialization 89
- specialized container 201
- `sph_bessel()` 228
- `split_view` 187
- `sqrt()` 228
- `<sstream>` 123, 147
- standard
 - ISO C++ 2
 - library 119
 - library algorithm 181
 - library, C++ 2
 - library, C with Classes 259
 - library, C++98 259
 - library container 168
 - library facilities 120
 - library header 121, 123
 - library module 123
 - library `std` 121
 - library suffix 121
 - mathematical function 228
- standardization, C++ 260
- state
 - I/O 141
 - moved-from 224
- statement
 - `for` 12
 - `if` 14
 - range `for` 12
 - `switch` 15
 - `while` 14
- `static_assert` 234
 - and `concept` 108
 - assertion 50
- `static_cast` 59
- `std`
 - module 34, 277
 - namespace 3, 36, 121
 - standard library 121
 - sub-namespaces 121
- `std::byte` byte 224
- `std.compat`, module 277
- `<stdexcept>` 123

- std.h 278
- stem() 152
- STL 259
- stopping thread 248
- stop_requested() 249
- stop_source 249
- stop_token 248
- store, free 57
- strict_weak_order, concept 191
- string
 - C-style 13
 - literal " 3
 - literal, raw 130
 - literal template argument 91
 - literal, type of 127
 - Unicode 128
- string 125
 - [] 126
 - == 126
 - append += 126
 - assignment = 126
 - concatenation + 125
 - implementation 127
 - input 140
 - output 140
 - replace() 126
 - substr() 126
- <string> 123, 125
- string_literals, literals 127
- stringstream 147
- string_view 128
 - and span 200
- string_view_literals, literals 129
- strstream deprecated 148, 267
- struct 22
 - class and 25
 - union and 27
- structured binding 41
- style
 - C++ 269
 - C 269
- subclass, superclass and 61
- sub-namespaces, std 121
- subscript operator [] 25
- subscripting, [] 169
- substr(), string 126
- suffix 84
 - return type 263
 - return type 40
 - s literal 127
 - standard library 121
 - sv literal 129
 - time 214
- sum() example 104
- superclass and subclass 61
- sv literal suffix 129

- swap() 84
- swappable, concept 190
- swappable_with, concept 190
- switch statement 15
- synchronized_pool_resource 167
- syncstream 149
- sync_with_stdio() 149
- syntax, [[]] attribute 263
- system_clock 214

T

- table, hash 165
- tagged union 28
- take() 189
- take_view 186–187
- tanh() 228
- task
 - and thread 238
 - communication 245
 - passing data to 239
 - returning results from 240
 - sharing data 241
- task example 253
- tau example 235
- TC++PL 256
- template
 - argument, string literal 91
 - type safty 90
- template 87
 - ..., variadic 114
 - alias 100
 - argument, constrained 90
 - argument, default 108
 - class 88
 - compilation model 117
 - constrained 90
 - debugging 113
 - definition checking 109
 - extern 264
 - function 93
 - instantiation time 117
 - notation 105
 - purpose 103
 - requirement 104
 - variable 99
 - virtual 94
- terminate() termination 225
- termination 48
 - exit() 225
 - quick_exit() 225
 - terminate() 225
- this 76
- [this] and [*this] 95
- thread
 - join() 238

- `packaged_task` 247
 - stopping 248
 - task and 238
- `<thread>` 123, 238
- `thread_local` 264
- `three_way_comparable`, concept 190
- `three_way_comparable_with`, concept 190
- time 214
 - suffix 214
 - `template` instantiation 117
- timeline, C++ 256
- `time_point` 214
- `time_zone` 216
- timing, `clock` 243
- to hardware, mapping 16
- `totally_ordered`, concept 190
- `totally_ordered_with`, concept 190
- tour, sightseeing
- `transform_reduce()` 229
- `transform_view` 187
- translation unit 32
- `try`
 - block 44
 - block as function body 161
- `try-block`, RAII and 48
- `try-statement`, RAII and 45
- type 6
 - alias 234
 - argument 90
 - associate 222
 - built-in 21
 - `concept` and 111
 - concepts 190
 - concrete 54
 - conversion, explicit 59
 - file 154
 - function 217
 - fundamental 6
 - generator 221
 - `get<>()` by 207
 - input of user-defined 141
 - move-only 223
 - of string literal 127
 - output of user-defined 141
 - parameterized 88
 - polymorphic 60
 - predicate 218
 - safe 262
 - safety template 90
 - size of 6
 - user-defined 21
- `typename` 88, 177
- `<type_traits>` 218
- typing, duck 117

U

- `\U, regex` 133
- `\u, regex` 133
- UDL, user-defined literal 84
- `uint_least64_t` 234
- `unset(), iostream` 141
- Unicode string 128
- `uniform_int_distribution` 231
- uninitialized 8
- `union` 27
 - and `struct` 27
 - and `variant` 28
 - tagged 28
- `unique_copy()` 173, 182
- `unique_lock` 242, 244
 - and `scoped_lock` 245
- `unique_ptr` 68, 197
- `[[unlikely]]` attribute 265
- `unordered_map` 165, 168
 - `hash<>` 84
 - map and 166
- `<unordered_map>` 123
- `unordered_multimap` 168
- `unordered_multiset` 168
- `unordered_set` 168
- `unsigned` 6
- `unsigned_integral`, concept 190
- `upper, regex` 133
- use
 - C++ 262
 - `concept` 104
- user-defined
 - literal 264
 - literal UDL 84
 - operator 57
 - type 21
 - type, input of 141
 - type, output of 141
- `using`
 - alias 100
 - declaration 36
 - directive 36
 - `enum` 26
- usual arithmetic conversions 7
- `<utility>` 123, 206

V

- `valarray` 233
- `<valarray>` 233
- value 6
 - argument 90
 - key and 164
 - mapped type 164
 - `return`, function 72

- values, multiple [return](#) 41
- [Value_type](#) example 109
- [value_type](#) 100, 169
- variable 5–6
 - [concept](#) and 111
 - [template](#) 99
- variadic [template](#) ... 114
- [variant](#) 209
 - union and 28
- [Vec](#)
 - example 161
 - range checking 161
- vector arithmetic 233
- [Vector](#) example 22–23, 29, 33–34, 57–58, 73, 75, 77, 88–89, 91–92, 100
- [vector](#) 158, 168
 - array vs. 203
- [<vector>](#) 123
- [vector<bool>](#) 201
- vectorized 183
- [vformat\(\)](#) 146
- view 186
- [view](#), concept 193
- [views, namespace](#) 188
- [virtual](#) 60
 - destructor 65
 - function, implementation of 62
 - function table [vtbl](#) 62
 - pure 60
 - [template](#) 94
- [void*](#) incompatibility 270
- [void return](#) type 4
- [vtbl](#), [virtual](#) function table 62

W

- [w, regex](#) 133
- [\W, regex](#) 133
- [\w, regex](#) 133
- [wait\(\), condition_variable](#) 244
- [weekday](#) 214
- WG21 256
- [while](#) statement 14
- whitespace 139

X

- [:x, format\(\)](#) 145
- X3J16 260
- [xdigit, regex](#) 133

Y

- year 214

Z

- zero, rule of 73
- [zoned_time](#) 216