The C++ Standard Library
Second Edition
This page intentionally left blank
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.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsonTechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Josuttis, Nicolai M.

The C++ standard library : a tutorial and reference / Nicolai M. Josuttis.—2nd ed.
p. cm.
Includes bibliographical references and index.
1. C++ (Computer program language) I. Title.
QA76.73.C153J69 2012
005.13’3-dc23

Copyright © 2012 Pearson Education, Inc.

This book was typeset by the author using the \LaTeX document processing system.

All rights reserved. Printed in the United States of America. 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. To obtain permission to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to (201) 236-3290.


Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan.

Third Printing, July 2013
To those who care
for people and mankind
This page intentionally left blank
Contents

Preface to the Second Edition xxiii
Acknowledgments for the Second Edition xxiv
Preface to the First Edition xxv
Acknowledgments for the First Edition xxvi

1 About This Book 1
  1.1 Why This Book ..................................... 1
  1.2 Before Reading This Book .............................. 2
  1.3 Style and Structure of the Book ........................ 2
  1.4 How to Read This Book ................................ 4
  1.5 State of the Art .................................... 5
  1.6 Example Code and Additional Information ............... 5
  1.7 Feedback ......................................... 5

2 Introduction to C++ and the Standard Library 7
  2.1 History of the C++ Standards ........................... 7
     2.1.1 Common Questions about the C++11 Standard ........ 8
     2.1.2 Compatibility between C++98 and C++11 ............ 9
  2.2 Complexity and Big-O Notation .......................... 10

3 New Language Features 13
  3.1 New C++11 Language Features .......................... 13
     3.1.1 Important Minor Syntax Cleanups ................... 13
     3.1.2 Automatic Type Deduction with auto ................. 14
     3.1.3 Uniform Initialization and Initializer Lists ........ 15
     3.1.4 Range-Based for Loops ............................ 17
     3.1.5 Move Semantics and Rvalue References .............. 19
3.1.6 New String Literals ..................................... 23
3.1.7 Keyword noexcept ..................................... 24
3.1.8 Keyword constexpr .................................... 26
3.1.9 New Template Features ................................... 26
3.1.10 Lambdas ............................................... 28
3.1.11 Keyword decltype ..................................... 32
3.1.12 New Function Declaration Syntax ....................... 32
3.1.13 Scoped Enumerations ................................ 32
3.1.14 New Fundamental Data Types .......................... 33
3.2 Old “New” Language Features .............................. 33
3.2.1 Explicit Initialization for Fundamental Types ............ 37
3.2.2 Definition of main() .................................... 37

4 General Concepts ............................................ 39
4.1 Namespace std ............................................ 39
4.2 Header Files ............................................... 40
4.3 Error and Exception Handling .............................. 41
4.3.1 Standard Exception Classes ............................ 41
4.3.2 Members of Exception Classes .......................... 44
4.3.3 Passing Exceptions with Class exception_ptr .......... 52
4.3.4 Throwing Standard Exceptions .......................... 53
4.3.5 Deriving from Standard Exception Classes ............... 54
4.4 Callable Objects .......................................... 54
4.5 Concurrency and Multithreading ........................... 55
4.6 Allocators .................................................. 57

5 Utilities ...................................................... 59
5.1 Pairs and Tuples .......................................... 60
5.1.1 Pairs .................................................. 60
5.1.2 Tuples ................................................ 68
5.1.3 I/O for Tuples ........................................ 74
5.1.4 Conversions between tuples and pairs ................... 75
5.2 Smart Pointers ............................................. 76
5.2.1 Class shared_ptr ....................................... 76
5.2.2 Class weak_ptr ......................................... 84
5.2.3 Misusing Shared Pointers ............................... 89
5.2.4 Shared and Weak Pointers in Detail ...................... 92
5.2.5 Class unique_ptr ....................................... 98
5.2.6 Class unique_ptr in Detail ........................................ 110
5.2.7 Class auto_ptr .................................................. 113
5.2.8 Final Words on Smart Pointers .............................. 114

5.3 Numeric Limits .................................................. 115

5.4 Type Traits and Type Utilities ............................... 122
5.4.1 Purpose of Type Traits ........................................... 122
5.4.2 Type Traits in Detail ........................................... 125
5.4.3 Reference Wrappers .............................................. 132
5.4.4 Function Type Wrappers ........................................ 133

5.5 Auxiliary Functions ............................................ 134
5.5.1 Processing the Minimum and Maximum .................... 134
5.5.2 Swapping Two Values .......................................... 136
5.5.3 Supplementary Comparison Operators .................... 138

5.6 Compile-Time Fractional Arithmetic with Class ratio<> ............. 140

5.7 Clocks and Timers ............................................. 143
5.7.1 Overview of the Chrono Library ............................ 143
5.7.2 Durations .................................................... 144
5.7.3 Clocks and Timepoints ...................................... 149
5.7.4 Date and Time Functions by C and POSIX ............... 157
5.7.5 Blocking with Timers ........................................ 160

5.8 Header Files <cstddef>, <cstdlib>, and <cstring> ............ 161
5.8.1 Definitions in <cstddef> .................................. 161
5.8.2 Definitions in <cstdlib> .................................. 162
5.8.3 Definitions in <cstring> .................................. 163

6 The Standard Template Library ..................................... 165

6.1 STL Components ............................................... 165
6.2 Containers ................................................... 167
6.2.1 Sequence Containers ....................................... 169
6.2.2 Associative Containers ..................................... 177
6.2.3 Unordered Containers ....................................... 180
6.2.4 Associative Arrays .......................................... 185
6.2.5 Other Containers ............................................ 187
6.2.6 Container Adapters ......................................... 188

6.3 Iterators ...................................................... 188
6.3.1 Further Examples of Using Associative and Unordered Containers .... 193
6.3.2 Iterator Categories ........................................ 198
6.4 Algorithms ........................................... 199
  6.4.1 Ranges ........................................ 203
  6.4.2 Handling Multiple Ranges .......................... 207

6.5 Iterator Adapters ..................................... 210
  6.5.1 Insert Iterators ................................ 210
  6.5.2 Stream Iterators ................................ 212
  6.5.3 Reverse Iterators ............................... 214
  6.5.4 Move Iterators ................................ 216

6.6 User-Defined Generic Functions ......................... 216

6.7 Manipulating Algorithms .............................. 217
  6.7.1 “Removing” Elements ............................ 218
  6.7.2 Manipulating Associative and Unordered Containers .. 221
  6.7.3 Algorithms versus Member Functions ................. 223

6.8 Functions as Algorithm Arguments ..................... 224
  6.8.1 Using Functions as Algorithm Arguments .......... 224
  6.8.2 Predicates ................................... 226

6.9 Using Lambdas ....................................... 229

6.10 Function Objects ................................... 233
  6.10.1 Definition of Function Objects ................ 233
  6.10.2 Predefined Function Objects .................... 239
  6.10.3 Binders ................................... 241
  6.10.4 Function Objects and Binders versus Lambdas ...... 243

6.11 Container Elements .................................. 244
  6.11.1 Requirements for Container Elements ............ 244
  6.11.2 Value Semantics or Reference Semantics ......... 245

6.12 Errors and Exceptions inside the STL .................. 245
  6.12.1 Error Handling ................................ 246
  6.12.2 Exception Handling ............................ 248

6.13 Extending the STL .................................. 250
  6.13.1 Integrating Additional Types ................... 250
  6.13.2 Deriving from STL Types ....................... 251

7 STL Containers ........................................ 253

7.1 Common Container Abilities and Operations .............. 254
  7.1.1 Container Abilities ............................. 254
  7.1.2 Container Operations ............................ 254
  7.1.3 Container Types ................................ 260
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2</td>
<td>Arrays</td>
<td>261</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Abilities of Arrays</td>
<td>261</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Array Operations</td>
<td>263</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Using arrays as C-Style Arrays</td>
<td>267</td>
</tr>
<tr>
<td>7.2.4</td>
<td>Exception Handling</td>
<td>268</td>
</tr>
<tr>
<td>7.2.5</td>
<td>Tuple Interface</td>
<td>268</td>
</tr>
<tr>
<td>7.2.6</td>
<td>Examples of Using Arrays</td>
<td>268</td>
</tr>
<tr>
<td>7.3</td>
<td>Vectors</td>
<td>270</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Abilities of Vectors</td>
<td>270</td>
</tr>
<tr>
<td>7.3.2</td>
<td>Vector Operations</td>
<td>273</td>
</tr>
<tr>
<td>7.3.3</td>
<td>Using Vectors as C-Style Arrays</td>
<td>278</td>
</tr>
<tr>
<td>7.3.4</td>
<td>Exception Handling</td>
<td>278</td>
</tr>
<tr>
<td>7.3.5</td>
<td>Examples of Using Vectors</td>
<td>279</td>
</tr>
<tr>
<td>7.3.6</td>
<td>Class vector&lt;bool&gt;</td>
<td>281</td>
</tr>
<tr>
<td>7.4</td>
<td>Deques</td>
<td>283</td>
</tr>
<tr>
<td>7.4.1</td>
<td>Abilities of Deques</td>
<td>284</td>
</tr>
<tr>
<td>7.4.2</td>
<td>Deque Operations</td>
<td>285</td>
</tr>
<tr>
<td>7.4.3</td>
<td>Exception Handling</td>
<td>288</td>
</tr>
<tr>
<td>7.4.4</td>
<td>Examples of Using Deques</td>
<td>288</td>
</tr>
<tr>
<td>7.5</td>
<td>Lists</td>
<td>290</td>
</tr>
<tr>
<td>7.5.1</td>
<td>Abilities of Lists</td>
<td>290</td>
</tr>
<tr>
<td>7.5.2</td>
<td>List Operations</td>
<td>291</td>
</tr>
<tr>
<td>7.5.3</td>
<td>Exception Handling</td>
<td>296</td>
</tr>
<tr>
<td>7.5.4</td>
<td>Examples of Using Lists</td>
<td>298</td>
</tr>
<tr>
<td>7.6</td>
<td>Forward Lists</td>
<td>300</td>
</tr>
<tr>
<td>7.6.1</td>
<td>Abilities of Forward Lists</td>
<td>300</td>
</tr>
<tr>
<td>7.6.2</td>
<td>Forward List Operations</td>
<td>302</td>
</tr>
<tr>
<td>7.6.3</td>
<td>Exception Handling</td>
<td>311</td>
</tr>
<tr>
<td>7.6.4</td>
<td>Examples of Using Forward Lists</td>
<td>312</td>
</tr>
<tr>
<td>7.7</td>
<td>Sets and Multisets</td>
<td>314</td>
</tr>
<tr>
<td>7.7.1</td>
<td>Abilities of Sets and Multisets</td>
<td>315</td>
</tr>
<tr>
<td>7.7.2</td>
<td>Set and Multiset Operations</td>
<td>316</td>
</tr>
<tr>
<td>7.7.3</td>
<td>Exception Handling</td>
<td>325</td>
</tr>
<tr>
<td>7.7.4</td>
<td>Examples of Using Sets and Multisets</td>
<td>325</td>
</tr>
<tr>
<td>7.7.5</td>
<td>Example of Specifying the Sorting Criterion at Runtime</td>
<td>328</td>
</tr>
</tbody>
</table>
7.8 Maps and Multimaps ........................................ 331
  7.8.1 Abilities of Maps and Multimaps ...................... 332
  7.8.2 Map and Multimap Operations .......................... 333
  7.8.3 Using Maps as Associative Arrays .................... 343
  7.8.4 Exception Handling .................................. 345
  7.8.5 Examples of Using Maps and Multimaps ............... 345
  7.8.6 Example with Maps, Strings, and Sorting Criterion at Runtime .... 351

7.9 Unordered Containers ..................................... 355
  7.9.1 Abilities of Unordered Containers .................... 357
  7.9.2 Creating and Controlling Unordered Containers ....... 359
  7.9.3 Other Operations for Unordered Containers .......... 367
  7.9.4 The Bucket Interface ................................ 374
  7.9.5 Using Unordered Maps as Associative Arrays .......... 374
  7.9.6 Exception Handling .................................. 375
  7.9.7 Examples of Using Unordered Containers ............. 375

7.10 Other STL Containers .................................... 385
  7.10.1 Strings as STL Containers .......................... 385
  7.10.2 Ordinary C-Style Arrays as STL Containers ........ 386

7.11 Implementing Reference Semantics ....................... 388

7.12 When to Use Which Container ............................ 392

8 STL Container Members in Detail ......................... 397
  8.1 Type Definitions ...................................... 397
  8.2 Create, Copy, and Destroy Operations .................. 400
  8.3 Nonmodifying Operations ................................ 403
    8.3.1 Size Operations .................................. 403
    8.3.2 Comparison Operations ............................ 404
    8.3.3 Nonmodifying Operations for Associative and Unordered Containers .... 404
  8.4 Assignments ............................................ 406
  8.5 Direct Element Access .................................. 408
  8.6 Operations to Generate Iterators ....................... 410
  8.7 Inserting and Removing Elements ....................... 411
    8.7.1 Inserting Single Elements ........................ 411
    8.7.2 Inserting Multiple Elements ....................... 416
    8.7.3 Removing Elements ................................ 417
    8.7.4 Resizing ........................................... 420
8.8 Special Member Functions for Lists and Forward Lists ................. 420
  8.8.1 Special Member Functions for Lists (and Forward Lists) ........... 420
  8.8.2 Special Member Functions for Forward Lists Only ................. 423
8.9 Container Policy Interfaces .................................. 427
  8.9.1 Nonmodifying Policy Functions ................................ 427
  8.9.2 Modifying Policy Functions .................................. 428
  8.9.3 Bucket Interface for Unordered Containers .................... 429
8.10 Allocator Support .......................................... 430
  8.10.1 Fundamental Allocator Members ................................ 430
  8.10.2 Constructors with Optional Allocator Parameters .......... 430

9 STL Iterators .............................................. 433
  9.1 Header Files for Iterators .................................. 433
  9.2 Iterator Categories .................................... 433
    9.2.1 Output Iterators ................................... 433
    9.2.2 Input Iterators .................................... 435
    9.2.3 Forward Iterators .................................. 436
    9.2.4 Bidirectional Iterators .............................. 437
    9.2.5 Random-Access Iterators ............................. 438
    9.2.6 The Increment and Decrement Problem of Vector Iterators ... 440
  9.3 Auxiliary Iterator Functions ................................ 441
    9.3.1 advance() ......................................... 441
    9.3.2 next() and prev() .................................. 443
    9.3.3 distance() ....................................... 445
    9.3.4 iter_swap() ....................................... 446
  9.4 Iterator Adapters ....................................... 448
    9.4.1 Reverse Iterators .................................. 448
    9.4.2 Insert Iterators .................................... 454
    9.4.3 Stream Iterators .................................. 460
    9.4.4 Move Iterators ..................................... 466
  9.5 Iterator Traits ......................................... 466
    9.5.1 Writing Generic Functions for Iterators .................. 468
  9.6 Writing User-Defined Iterators ................................ 471
10 STL Function Objects and Using Lambdas 475
10.1 The Concept of Function Objects 475
   10.1.1 Function Objects as Sorting Criteria 476
   10.1.2 Function Objects with Internal State 478
   10.1.3 The Return Value of for_each() 482
   10.1.4 Predicates versus Function Objects 483
10.2 Predefined Function Objects and Binders 486
   10.2.1 Predefined Function Objects 486
   10.2.2 Function Adapters and Binders 487
   10.2.3 User-Defined Function Objects for Function Adapters 495
   10.2.4 Deprecated Function Adapters 497
10.3 Using Lambdas 499
   10.3.1 Lambdas versus Binders 499
   10.3.2 Lambdas versus Stateful Function Objects 500
   10.3.3 Lambdas Calling Global and Member Functions 502
   10.3.4 Lambdas as Hash Function, Sorting, or Equivalence Criterion 504

11 STL Algorithms 505
11.1 Algorithm Header Files 505
11.2 Algorithm Overview 505
   11.2.1 A Brief Introduction 506
   11.2.2 Classification of Algorithms 506
11.3 Auxiliary Functions 517
11.4 The for_each() Algorithm 519
11.5 Nonmodifying Algorithms 524
   11.5.1 Counting Elements 524
   11.5.2 Minimum and Maximum 525
   11.5.3 Searching Elements 528
   11.5.4 Comparing Ranges 542
   11.5.5 Predicates for Ranges 550
11.6 Modifying Algorithms 557
   11.6.1 Copying Elements 557
   11.6.2 Moving Elements 561
   11.6.3 Transforming and Combining Elements 563
   11.6.4 Swapping Elements 566
   11.6.5 Assigning New Values 568
   11.6.6 Replacing Elements 571
11.7 Removing Algorithms ........................................ 575
  11.7.1 Removing Certain Values .......................... 575
  11.7.2 Removing Duplicates ............................. 578

11.8 Mutating Algorithms ..................................... 583
  11.8.1 Reversing the Order of Elements .................... 583
  11.8.2 Rotating Elements ................................. 584
  11.8.3 Permuting Elements ............................... 587
  11.8.4 Shuffling Elements ................................. 589
  11.8.5 Moving Elements to the Front ...................... 592
  11.8.6 Partition into Two Subranges ....................... 594

11.9 Sorting Algorithms ...................................... 596
  11.9.1 Sorting All Elements ............................. 596
  11.9.2 Partial Sorting ................................ 599
  11.9.3 Sorting According to the n
nth Element ................... 602
  11.9.4 Heap Algorithms ................................. 604

11.10 Sorted-Range Algorithms ............................... 608
  11.10.1 Searching Elements ............................. 608
  11.10.2 Merging Elements ............................... 614

11.11 Numeric Algorithms .................................. 623
  11.11.1 Processing Results ............................. 623
  11.11.2 Converting Relative and Absolute Values .......... 627

12 Special Containers ......................................... 631
  12.1 Stacks ............................................... 632
    12.1.1 The Core Interface ............................. 633
    12.1.2 Example of Using Stacks ......................... 633
    12.1.3 A User-Defined Stack Class ...................... 635
    12.1.4 Class stack<> in Detail ......................... 637

  12.2 Queues ............................................. 638
    12.2.1 The Core Interface ............................. 639
    12.2.2 Example of Using Queues ......................... 640
    12.2.3 A User-Defined Queue Class ...................... 641
    12.2.4 Class queue<> in Detail ......................... 641

  12.3 Priority Queues ..................................... 641
    12.3.1 The Core Interface ............................. 643
    12.3.2 Example of Using Priority Queues ................. 643
    12.3.3 Class priority_queue<> in Detail ................. 644
12.4 Container Adapters in Detail .................................. 645
  12.4.1 Type Definitions .................................. 645
  12.4.2 Constructors .................................... 646
  12.4.3 Supplementary Constructors for Priority Queues ....... 646
  12.4.4 Operations ..................................... 647
12.5 Bitsets ........................................ 650
  12.5.1 Examples of Using Bitsets ......................... 651
  12.5.2 Class bitset in Detail ............................ 653

13 Strings ............................................ 655
  13.1 Purpose of the String Classes ............................. 656
    13.1.1 A First Example: Extracting a Temporary Filename .... 656
    13.1.2 A Second Example: Extracting Words and Printing Them Backward ...... 660
  13.2 Description of the String Classes ......................... 663
    13.2.1 String Types .................................. 663
    13.2.2 Operation Overview ................................ 666
    13.2.3 Constructors and Destructor ....................... 667
    13.2.4 Strings and C-Strings ............................ 668
    13.2.5 Size and Capacity ................................ 669
    13.2.6 Element Access .................................. 671
    13.2.7 Comparisons .................................... 672
    13.2.8 Modifiers ....................................... 673
    13.2.9 Substrings and String Concatenation ............... 676
    13.2.10 Input/Output Operators ......................... 677
    13.2.11 Searching and Finding ........................... 678
    13.2.12 The Value npos ................................ 680
    13.2.13 Numeric Conversions ............................. 681
    13.2.14 Iterator Support for Strings ..................... 684
    13.2.15 Internationalization ............................. 689
    13.2.16 Performance ................................... 692
    13.2.17 Strings and Vectors ............................. 692
  13.3 String Class in Detail ................................ 693
    13.3.1 Type Definitions and Static Values ................. 693
    13.3.2 Create, Copy, and Destroy Operations ............... 694
    13.3.3 Operations for Size and Capacity ................... 696
    13.3.4 Comparisons .................................... 697
    13.3.5 Character Access ................................ 699
    13.3.6 Generating C-Strings and Character Arrays .......... 700
13.3.7 Modifying Operations ........................................ 700
13.3.8 Searching and Finding ..................................... 708
13.3.9 Substrings and String Concatenation ....................... 711
13.3.10 Input/Output Functions ................................... 712
13.3.11 Numeric Conversions ..................................... 713
13.3.12 Generating Iterators .................................... 714
13.3.13 Allocator Support ...................................... 715

14 Regular Expressions 717
14.1 The Regex Match and Search Interface ....................... 717
14.2 Dealing with Subexpressions ................................. 720
14.3 Regex Iterators ............................................ 726
14.4 Regex Token Iterators .................................... 727
14.5 Replacing Regular Expressions ............................. 730
14.6 Regex Flags ............................................... 732
14.7 Regex Exceptions ......................................... 735
14.8 The Regex ECMAScript Grammar ............................ 738
14.9 Other Grammars ........................................... 739
14.10 Basic Regex Signatures in Detail ......................... 740

15 Input/Output Using Stream Classes 743
15.1 Common Background of I/O Streams ......................... 744
15.1.1 Stream Objects ....................................... 744
15.1.2 Stream Classes ....................................... 744
15.1.3 Global Stream Objects ................................. 745
15.1.4 Stream Operators .................................... 745
15.1.5 Manipulators ......................................... 746
15.1.6 A Simple Example .................................... 746
15.2 Fundamental Stream Classes and Objects ................. 748
15.2.1 Classes and Class Hierarchy ........................... 748
15.2.2 Global Stream Objects ................................. 751
15.2.3 Header Files .......................................... 752
15.3 Standard Stream Operators << and >> .................... 753
15.3.1 Output Operator << .................................... 753
15.3.2 Input Operator >> ..................................... 754
15.3.3 Input/Output of Special Types ......................... 755
15.4 State of Streams ................................................................. 758
  15.4.1 Constants for the State of Streams ................................. 758
  15.4.2 Member Functions Accessing the State of Streams ............... 759
  15.4.3 Stream State and Boolean Conditions .............................. 760
  15.4.4 Stream State and Exceptions .................................... 762

15.5 Standard Input/Output Functions ........................................ 767
  15.5.1 Member Functions for Input ....................................... 768
  15.5.2 Member Functions for Output .................................... 771
  15.5.3 Example Uses ...................................................... 772
  15.5.4 Sentry Objects .................................................... 772

15.6 Manipulators ................................................................. 774
  15.6.1 Overview of All Manipulators .................................. 774
  15.6.2 How Manipulators Work .......................................... 776
  15.6.3 User-Defined Manipulators .................................... 777

15.7 Formatting .................................................................. 779
  15.7.1 Format Flags ....................................................... 779
  15.7.2 Input/Output Format of Boolean Values ........................ 781
  15.7.3 Field Width, Fill Character, and Adjustment .................. 781
  15.7.4 Positive Sign and Uppercase Letters ........................... 784
  15.7.5 Numeric Base ...................................................... 785
  15.7.6 Floating-Point Notation ........................................ 787
  15.7.7 General Formatting Definitions ................................ 789

15.8 Internationalization ...................................................... 790

15.9 File Access ................................................................. 791
  15.9.1 File Stream Classes .............................................. 791
  15.9.2 Rvalue and Move Semantics for File Streams ................... 795
  15.9.3 File Flags .......................................................... 796
  15.9.4 Random Access .................................................... 799
  15.9.5 Using File Descriptors ......................................... 801

15.10 Stream Classes for Strings .............................................. 802
  15.10.1 String Stream Classes ........................................... 802
  15.10.2 Move Semantics for String Streams ............................ 806
  15.10.3 char* Stream Classes ......................................... 807

15.11 Input/Output Operators for User-Defined Types ..................... 810
  15.11.1 Implementing Output Operators ................................ 810
  15.11.2 Implementing Input Operators ................................ 812
  15.11.3 Input/Output Using Auxiliary Functions ....................... 814
15.11.4 User-Defined Format Flags ................................. 815
15.11.5 Conventions for User-Defined Input/Output Operators ............................ 818

15.12 Connecting Input and Output Streams ........................................ 819
15.12.1 Loose Coupling Using tie() .................................... 819
15.12.2 Tight Coupling Using Stream Buffers ........................................ 820
15.12.3 Redirecting Standard Streams ........................................ 822
15.12.4 Streams for Reading and Writing ........................................ 824

15.13 The Stream Buffer Classes ........................................ 826
15.13.1 The Stream Buffer Interfaces ........................................ 826
15.13.2 Stream Buffer Iterators ........................................ 828
15.13.3 User-Defined Stream Buffers ........................................ 832

15.14 Performance Issues ........................................ 844
15.14.1 Synchronization with C’s Standard Streams ........................................ 845
15.14.2 Buffering in Stream Buffers ........................................ 845
15.14.3 Using Stream Buffers Directly ........................................ 846

16 Internationalization ........................................ 849
16.1 Character Encodings and Character Sets ........................................ 850
16.1.1 Multibyte and Wide-Character Text ........................................ 850
16.1.2 Different Character Sets ........................................ 851
16.1.3 Dealing with Character Sets in C++ ........................................ 852
16.1.4 Character Traits ........................................ 853
16.1.5 Internationalization of Special Characters ........................................ 857

16.2 The Concept of Locales ........................................ 857
16.2.1 Using Locales ........................................ 858
16.2.2 Locale Facets ........................................ 864

16.3 Locales in Detail ........................................ 866

16.4 Facets in Detail ........................................ 869
16.4.1 Numeric Formatting ........................................ 870
16.4.2 Monetary Formatting ........................................ 874
16.4.3 Time and Date Formatting ........................................ 884
16.4.4 Character Classification and Conversion ........................................ 891
16.4.5 String Collation ........................................ 904
16.4.6 Internationalized Messages ........................................ 905
17 Numerics

17.1 Random Numbers and Distributions ................................................................. 907
  17.1.1 A First Example ................................................................................. 908
  17.1.2 Engines ............................................................................................ 912
  17.1.3 Engines in Detail ................................................................................ 915
  17.1.4 Distributions ..................................................................................... 917
  17.1.5 Distributions in Detail ......................................................................... 921

17.2 Complex Numbers ......................................................................................... 925
  17.2.1 Class complex<> in General ................................................................. 925
  17.2.2 Examples Using Class complex<> ....................................................... 926
  17.2.3 Operations for Complex Numbers ...................................................... 928
  17.2.4 Class complex<> in Detail .................................................................. 935

17.3 Global Numeric Functions ........................................................................... 941

17.4 Valarrays ...................................................................................................... 943

18 Concurrency

18.1 The High-Level Interface: async() and Futures ............................................ 946
  18.1.1 A First Example Using async() and Futures ...................................... 946
  18.1.2 An Example of Waiting for Two Tasks ................................................ 955
  18.1.3 Shared Futures .................................................................................... 960

18.2 The Low-Level Interface: Threads and Promises .......................................... 964
  18.2.1 Class std::thread .............................................................................. 964
  18.2.2 Promises ........................................................................................... 969
  18.2.3 Class packaged_task<> ..................................................................... 972

18.3 Starting a Thread in Detail ............................................................................ 973
  18.3.1 async() in Detail ............................................................................... 974
  18.3.2 Futures in Detail ............................................................................... 975
  18.3.3 Shared Futures in Detail .................................................................... 976
  18.3.4 Class std::promise in Detail .............................................................. 977
  18.3.5 Class std::packaged_task in Detail ................................................. 977
  18.3.6 Class std::thread in Detail ............................................................... 979
  18.3.7 Namespace this_thread ..................................................................... 981

18.4 Synchronizing Threads, or the Problem of Concurrency .................................. 982
  18.4.1 Beware of Concurrency! ..................................................................... 982
  18.4.2 The Reason for the Problem of Concurrent Data Access .................... 983
  18.4.3 What Exactly Can Go Wrong (the Extent of the Problem) .................. 983
  18.4.4 The Features to Solve the Problems ................................................... 987
18.5 Mutexes and Locks ........................................ 989
  18.5.1 Using Mutexes and Locks ............................. 989
  18.5.2 Mutexes and Locks in Detail .......................... 998
  18.5.3 Calling Once for Multiple Threads ...................... 1000
18.6 Condition Variables ...................................... 1003
  18.6.1 Purpose of Condition Variables ......................... 1003
  18.6.2 A First Complete Example for Condition Variables ........ 1004
  18.6.3 Using Condition Variables to Implement a Queue for Multiple Threads 1006
  18.6.4 Condition Variables in Detail ......................... 1009
18.7 Atomics ............................................. 1012
  18.7.1 Example of Using Atomics ............................. 1012
  18.7.2 Atomics and Their High-Level Interface in Detail ......... 1016
  18.7.3 The C-Style Interface of Atomics ....................... 1019
  18.7.4 The Low-Level Interface of Atomics .................... 1019

19 Allocators .................................................. 1023
  19.1 Using Allocators as an Application Programmer ............ 1023
  19.2 A User-Defined Allocator ................................. 1024
  19.3 Using Allocators as a Library Programmer .................. 1026

Bibliography ..................................................... 1031
  Newsgroups and Forums ....................................... 1031
  Books and Web Sites ......................................... 1032

Index ............................................................ 1037
This page intentionally left blank
Preface to the Second Edition

I never thought that the first edition of this book would sell so long. But now, after twelve years, it’s time for a new edition that covers C++11, the new C++ standard.

Note that this means more than simply adding new libraries. C++ has changed. Almost all typical applications of parts of the library look a bit different now. This is not the result of a huge language change. It’s the result of many minor changes, such as using rvalue references and move semantics, range-based for loops, auto, and new template features. Thus, besides presenting new libraries and supplementary features of existing libraries, almost all of the examples in this book were rewritten at least partially. Nevertheless, to support programmers who still use “old” C++ environments, this book will describe differences between C++ versions whenever they appear.

I learned C++11 the hard way. Because I didn’t follow the standardization as it was happening I started to look at C++11 about two years ago. I really had trouble understanding it. But the people on the standardization committee helped me to describe and present the new features as they are intended to be used now.

Note, finally, that this book now has a problem: Although the book’s size grew from about 800 to more than 1,100 pages, I still can’t present the C++ standard library as a whole. The library part of the new C++11 standard alone now has about 750 pages, written in very condensed form without much explanation. For this reason, I had to decide which features to describe and in how much detail. Again, many people in the C++ community helped me to make this decision. The intent was to concentrate on what the average application programmer needs. For some missing parts, I provide a supplementary chapter on the Web site of this book, http://www.cppstdlib.com, but you still will find details not mentioned here in the standard.

The art of teaching is not the art of presenting everything. It’s the art of separating the wheat from the chaff so that you get the most out of it. May the exercise succeed.
This book presents ideas, concepts, solutions, and examples from many sources. Over the past several years, the C++ community introduced many ideas, concepts, proposals, and enhancements to C++ that became part of C++11. Thus, again I’d like to thank all the people who helped and supported me while preparing this new edition.

First, I’d like to thank everyone in the C++ community and on the C++ standardization committee. Besides all the work to add new language and library features, they had a hard time explaining everything to me, but they did so with patience and enthusiasm.

Scott Meyers and Anthony Williams allowed me to use their teaching material and book manuscripts so that I could find many useful examples not yet publicly available.

I’d also like to thank everyone who reviewed this book and gave valuable feedback and clarifications: Dave Abrahams, Alberto Ganesh Barbati, Pete Becker, Thomas Becker, Hans Boehm, Walter E. Brown, Paolo Carlini, Lawrence Crowl, Beman Dawes, Doug Gregor, David Grigsby, Pablo Halpern, Howard Hinnant, John Lakos, Bronek Kozicki, Dietmar Kühl, Daniel Krügler, Mat Marcus, Jens Maurer, Alisdair Meredith, Bartosz Milewski, P. J. Plauger, Tobias Schüle, Peter Sommerlad, Jonathan Wakely, and Anthony Williams.

There is one person who did an especially outstanding job. Whenever I had a question, Daniel Krügler answered almost immediately with incredible accurateness and knowledge. Everyone in the standardization process know that he treats everybody this way. Without him, both the C++ standard and this book would not have the quality they have now.

Many thanks to my editor Peter Gordon, Kim Boedigheimer, John Fuller, and Anna Popick from Addison-Wesley. Besides their support, they found the right balance between patience and pressure. The copy editor Evelyn Pyle and the proofreader Diane Freed did an incredible job translating my German English into American English. In addition, thanks to Frank Mittelbach for solving my LATEX issues.

Last but not least, all my thanks go to Jutta Eckstein. Jutta has the wonderful ability to force and support people in their ideals, ideas, and goals. While most people experience this only when working with her, I have the honor to benefit in my day-to-day life.
Preface to the First Edition

In the beginning, I only planned to write a small German book (400 pages or so) about the C++ standard library. That was in 1993. Now, in 1999 you see the result — a book in English with more than 800 pages of facts, figures, and examples. My goal is to describe the C++ standard library so that all (or almost all) your programming questions are answered before you think of the question. Note, however, that this is not a complete description of all aspects of the C++ standard library. Instead, I present the most important topics necessary for learning and programming in C++ by using its standard library.

Each topic is described based on the general concepts; this discussion then leads to the specific details needed to support everyday programming tasks. Specific code examples are provided to help you understand the concepts and the details.

That’s it — in a nutshell. I hope you get as much pleasure from reading this book as I did from writing it. Enjoy!
Acknowledgments for the First Edition

This book presents ideas, concepts, solutions, and examples from many sources. In a way it does not seem fair that my name is the only name on the cover. Thus, I’d like to thank all the people and companies who helped and supported me during the past few years.

First, I’d like to thank Dietmar Kühl. Dietmar is an expert on C++, especially on input/output streams and internationalization (he implemented an I/O stream library just for fun). He not only translated major parts of this book from German to English, he also wrote sections of this book using his expertise. In addition, he provided me with invaluable feedback over the years.

Second, I’d like to thank all the reviewers and everyone else who gave me their opinion. These people endow the book with a quality it would never have had without their input. (Because the list is extensive, please forgive me for any oversight.) The reviewers for the English version of this book included Chuck Allison, Greg Comeau, James A. Crottinger, Gabriel Dos Reis, Alan Ezust, Nathan Myers, Werner Mossner, Todd Veldhuizen, Chichiang Wan, Judy Ward, and Thomas Wickhult. The German reviewers included Ralf Boecker, Dirk Herrmann, Dietmar Kühl, Edda Lörke, Herbert Scheubner, Dominik Strasser, and Martin Weitzel. Additional input was provided by Matt Austern, Valentin Bonnard, Greg Colvin, Beman Dawes, Bill Gibbons, Lois Goldthwaite, Andrew Koenig, Steve Rumsby, Bjarne Stroustrup, and David Vandevoorde.

Special thanks to Dave Abrahams, Janet Cocker, Catherine Ohala, and Maureen Willard who reviewed and edited the whole book very carefully. Their feedback was an incredible contribution to the quality of this book.

A special thanks goes to my “personal living dictionary” — Herb Sutter — the author of the famous “Guru of the Week” (a regular series of C++ programming problems that is published on the comp.lang.c++.moderated Internet newsgroup).

I’d also like to thank all the people and companies who gave me the opportunity to test my examples on different platforms with different compilers. Many thanks to Steve Adamczyk, Mike Anderson, and John Spicer from EDG for their great compiler and their support. It was a big help during the standardization process and the writing of this book. Many thanks to P. J. Plauger and Dinkumware, Ltd, for their early standard-conforming implementation of the C++ standard library. Many thanks to Andreas Hommel and Metrowerks for an evaluative version of their CodeWarrior Programming Environment. Many thanks to all the developers of the free GNU and egcs compilers. Many thanks to Microsoft for an evaluative version of Visual C++. Many thanks to Roland Hartinger.
from Siemens Nixdorf Informations Systems AG for a test version of their C++ compiler. Many thanks to Topjects GmbH for an evaluative version of the ObjectSpace library implementation.

Many thanks to everyone from Addison Wesley Longman who worked with me. Among others this includes Janet Cocker, Mike Hendrickson, Debbie Lafferty, Marina Lang, Chanda Leary, Catherine Ohala, Marty Rabinowitz, Susanne Spitzer, and Maureen Willard. It was fun.

In addition, I’d like to thank the people at BREDEX GmbH and all the people in the C++ community, particularly those involved with the standardization process, for their support and patience (sometimes I ask really silly questions).

Last but not least, many thanks and kisses for my family: Ulli, Lucas, Anica, and Frederic. I definitely did not have enough time for them due to the writing of this book.

Have fun and be human!
This page intentionally left blank
5.6 Compile-Time Fractional Arithmetic with Class ratio<>

Since C++11, the C++ standard library provides an interface to specify compile-time fractions and to perform compile-time arithmetic with them. To quote [N2661:Chrono] (with minor modifications):\(^{26}\)

> The ratio utility is a general purpose utility inspired by Walter E. Brown allowing one to easily and safely compute rational values at compile time. The ratio class catches all errors (such as divide by zero and overflow) at compile time. It is used in the duration and time_point libraries [see Section 5.7, page 143] to efficiently create units of time. It can also be used in other “quantity” libraries (both standard-defined and user-defined), or anywhere there is a rational constant which is known at compile time. The use of this utility can greatly reduce the chances of runtime overflow because a ratio and any ratios resulting from ratio arithmetic are always reduced to lowest terms.

The ratio utility is provided in <ratio>, with class ratio<> defined as follows:

```cpp
class ratio {
  public:
    typedef ratio<num, den> type;
    static constexpr intmax_t num;
    static constexpr intmax_t den;
}
```

intmax_t designates a signed integer type capable of representing any value of any signed integer type. It is defined in <cstdint> or <stdint.h> with at least 64 bits. Numerator and denominator are both public and are automatically reduced to the lowest terms. For example:

```
// util/ratio1.cpp
#include <ratio>
#include <iostream>
using namespace std;

int main()
{
    typedef ratio<5,3> FiveThirds;
    cout << FiveThirds::num << "/" << FiveThirds::den << endl;
}
```

\(^{26}\) Thanks to Walter E. Brown, Howard Hinnant, Jeff Garland, and Marc Paterno for their friendly permission to quote [N2661:Chrono] here and in the following section covering the chrono library.
typedef ratio<25,15> AlsoFiveThirds;
cout << AlsoFiveThirds::num << "/" << AlsoFiveThirds::den << endl;

type<42,42> one;
cout << one.num << "/" << one.den << endl;

type<0> zero;
cout << zero.num << "/" << zero.den << endl;

type<7,-3> Neg;
cout << Neg::num << "/" << Neg::den << endl;
}

The program has the following output:
5/3
5/3
1/1
0/1
-7/3

Table 5.19 lists the compile-time operations defined for ratio types. The four basic arithmetic compile-time operations +, -, *, and / are defined as ratio_add, ratio_subtract, ratio_multiply, and ratio_divide. The resulting type is a ratio<>, so the static member type yields the corresponding type. For example, the following expression yields std::ratio<13,21> (computed as $\frac{6}{21} + \frac{7}{21}$):

\[
\text{std::ratio_add<std::ratio<2,7>,std::ratio<2,6>>::type}
\]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>ratio_add</td>
<td>Reduced sum of ratios</td>
<td>ratio&lt;&gt;</td>
</tr>
<tr>
<td>ratio_subtract</td>
<td>Reduced difference of ratios</td>
<td>ratio&lt;&gt;</td>
</tr>
<tr>
<td>ratio_multiply</td>
<td>Reduced product of ratios</td>
<td>ratio&lt;&gt;</td>
</tr>
<tr>
<td>ratio_divide</td>
<td>Reduced quotient of ratios</td>
<td>ratio&lt;&gt;</td>
</tr>
<tr>
<td>ratio_equal</td>
<td>Checks for ==</td>
<td>true_type or false_type</td>
</tr>
<tr>
<td>ratio_not_equal</td>
<td>Checks for !=</td>
<td>true_type or false_type</td>
</tr>
<tr>
<td>ratio_less</td>
<td>Checks for &lt;</td>
<td>true_type or false_type</td>
</tr>
<tr>
<td>ratio_less_equal</td>
<td>Checks for &lt;=</td>
<td>true_type or false_type</td>
</tr>
<tr>
<td>ratio_greater</td>
<td>Checks for &gt;</td>
<td>true_type or false_type</td>
</tr>
<tr>
<td>ratio_greater_equal</td>
<td>Checks for &gt;=</td>
<td>true_type or false_type</td>
</tr>
</tbody>
</table>

*Table 5.19. Operations of ratio<> Types*
In addition, you can compare two ratio types with ratio_equal, ratio_not_equal, ratio_less, ratio_less_equal, ratio_greater, or ratio_greater_equal. As with type traits, the resulting type is derived from true_type or false_type (see Section 5.4.2, page 125), so its member value yields true or false:

\[
\text{ratio_equal}(<\text{ratio}<5,3>,\text{ratio}<25,15>):\text{value} \quad //\text{yields true}
\]

As written, class ratio catches all errors, such as divide by zero and overflow, at compile time. For example,

\[
\text{ratio_multiply}<\text{ratio}<1,\text{numeric_limits<long long>::max()}>,,\text{ratio}<1,2>:\text{type}
\]

won’t compile, because \(\frac{1}{\text{max}}\) times \(\frac{1}{2}\) results in an overflow, with the resulting value of the denominator exceeding the limit of its type.

Similarly, the following expression won’t compile, because this is a division by zero:

\[
\text{ratio_divide}<\text{fiveThirds},\text{zero}:\text{type}
\]

Note, however, that the following expression will compile because the invalid value is detected when member type, num, or den are evaluated:

\[
\text{ratio_divide}<\text{fiveThirds},\text{zero}>
\]

<table>
<thead>
<tr>
<th>Name</th>
<th>Unit</th>
</tr>
</thead>
</table>
| yocto   | \(1,000,000,000,000,000,000,000,000,000,000,000\)
| zepto   | \(1,000,000,000,000,000,000,000,000,000,000,000\) (optional) |
| atto    | \(1,000,000,000,000,000,000,000,000,000,000,000\) (optional) |
| femto   | \(1,000,000,000,000,000,000,000,000,000,000,000\) |
| pico    | \(1,000,000,000,000,000,000,000,000,000,000,000\) |
| nano    | \(1,000,000,000,000,000,000,000,000,000,000,000\) |
| micro   | \(1,000,000,000,000,000,000,000,000,000,000,000\) |
| milli   | \(1,000,000,000,000,000,000,000,000,000,000,000\) |
| centi   | \(1,000,000,000,000,000,000,000,000,000,000,000\) |
| deci    | \(1,000,000,000,000,000,000,000,000,000,000,000\) |
| deca    | \(1,000,000,000,000,000,000,000,000,000,000,000\) |
| hecto   | \(1,000,000,000,000,000,000,000,000,000,000,000\) |
| kilo    | \(1,000,000,000,000,000,000,000,000,000,000,000\) |
| mega    | \(1,000,000,000,000,000,000,000,000,000,000,000\) |
| giga    | \(1,000,000,000,000,000,000,000,000,000,000,000\) |
| tera    | \(1,000,000,000,000,000,000,000,000,000,000,000\) |
| peta    | \(1,000,000,000,000,000,000,000,000,000,000,000\) |
| exa     | \(1,000,000,000,000,000,000,000,000,000,000,000\) |
| zetta   | \(1,000,000,000,000,000,000,000,000,000,000,000\) (optional) |
| yotta   | \(1,000,000,000,000,000,000,000,000,000,000,000\) (optional) |

Table 5.20. Predefined ratio Units
Predefined ratios make it more convenient to specify large or very small numbers (see Table 5.20). They allow you to specify large numbers without the inconvenient and error-prone listing of zeros. For example,

```
std::nano
```

is equivalent to

```
std::ratio<1,1000000000LL>
```

which makes it more convenient to specify, for example, nanoseconds (see Section 5.7.2, page 145). The units marked as “optional” are defined only if they are representable by `intmax_t`.

## 5.7 Clocks and Timers

One of the most obvious libraries a programming language should have is one to deal with date and time. However, experience shows that such a library is harder to design than it sounds. The problem is the amount of flexibility and precision the library should provide. In fact, in the past, the interfaces to system time provided by C and POSIX switched from seconds to milliseconds, then to microseconds, and finally to nanoseconds. The problem was that for each switch, a new interface was provided. For this reason, a precision-neutral library was proposed for C++11. This library is usually called the `chrono library` because its features are defined in `<chrono>`.

In addition, the C++ standard library provides the basic C and POSIX interfaces to deal with calendar time. Finally, you can use the thread library, provided since C++11, to wait for a thread or the program (the main thread) for a period of time.

### 5.7.1 Overview of the Chrono Library

The chrono library was designed to be able to deal with the fact that timers and clocks might be different on different systems and improve over time in precision. To avoid having to introduce a new time type every 10 years or so — as happened with the POSIX time libraries, for example — the goal was to provide a precision-neutral concept by separating duration and point of time (“timepoint”) from specific clocks. As a result, the core of the chrono library consists of the following types or concepts, which serve as abstract mechanisms to specify and deal with points in and durations of time:

- A **duration** of time is defined as a specific number of ticks over a time unit. One example is a duration such as “3 minutes” (3 ticks of a “minute”). Other examples are “42 milliseconds” or “86,400 seconds,” which represents the duration of 1 day. This concept also allows specifying something like “1.5 times a third of a second,” where 1.5 is the number of ticks and “a third of a second” the time unit used.

- A **timepoint** is defined as combination of a duration and a beginning of time (the so-called **epoch**). A typical example is a timepoint that represents “New Year’s Midnight 2000,” which is described as “1,262,300,400 seconds since January 1, 1970” (this day is the epoch of the system clock of UNIX and POSIX systems).
The concept of a timepoint, however, is parametrized by a clock, which is the object that defines the epoch of a timepoint. Thus, different clocks have different epochs. In general, operations dealing with multiple timepoints, such as processing the duration/difference between two timepoints, require using the same epoch/clock. A clock also provides a convenience function to yield the timepoint of now.

In other words, timepoint is defined as a duration before or after an epoch, which is defined by a clock (see Figure 5.4).

![Figure 5.4. Epoch, Durations, and Timepoints](image)

For more details about the motivation and design of these classes, see [N2661:Chrono]. Let’s look into these types and concepts in detail.

Note that all identifiers of the chrono library are defined in namespace std::chrono.

### 5.7.2 Durations

A duration is a combination of a value representing the number of ticks and a fraction representing the unit in seconds. Class ratio is used to specify the fraction (see Section 5.6, page 140). For example:

```cpp
std::chrono::duration<int> twentySeconds(20);
std::chrono::duration<double, std::ratio<60>> halfAMinute(0.5);
std::chrono::duration<long, std::ratio<1,1000>> oneMillisecond(1);
```

Here, the first template argument defines the type of the ticks, and the optional second template argument defines the unit type in seconds. Thus, the first line uses seconds as unit type, the second line uses minutes (“60 seconds”), and the third line uses milliseconds (“1/1000 of a second”).

For more convenience, the C++ standard library provides the following type definitions:

---

27 I use some quotes of [N2661:Chrono] in this book with friendly permission by the authors.
namespace std {
    namespace chrono {
        typedef duration<signed int-type == 64 bits, nano> nanoseconds;
        typedef duration<signed int-type == 55 bits, micro> microseconds;
        typedef duration<signed int-type == 45 bits, milli> milliseconds;
        typedef duration<signed int-type == 35 bits> seconds;
        typedef duration<signed int-type == 29 bits, ratio<60>> minutes;
        typedef duration<signed int-type == 23 bits, ratio<3600>> hours;
    }
}

With them, you can easily specify typical time periods:

```cpp
std::chrono::seconds twentySeconds(20);
std::chrono::hours aDay(24);
std::chrono::milliseconds oneMillisecond(1);
```

### Arithmetic Duration Operations

You can compute with durations in the expected way (see Table 5.21):

- You can process the sum, difference, product, or quotient of two durations.
- You can add or subtract ticks or other durations.
- You can compare two durations.

The important point here is that the unit type of two durations involved in such an operation might be different. Due to a provided overloading of `common_type<>` (see Section 5.4.1, page 124) for durations, the resulting duration will have a unit that is the greatest common divisor of the units of both operands.

For example, after

```cpp
chrono::seconds d1(42);  // 42 seconds
chrono::milliseconds d2(10);  // 10 milliseconds
```

the expression

```cpp
d1 - d2
```

yields a duration of 41,990 ticks of unit type milliseconds (\(\frac{1}{1000}\) seconds).

Or, more generally, after

```cpp
chrono::duration<int, ratio<1,3>> d1(1);  // 1 tick of 1/3 second
chrono::duration<int, ratio<1,5>> d2(1);  // 1 tick of 1/5 second
```

the expression

```cpp
d1 + d2
```

yields 8 ticks of \(\frac{1}{15}\) second and

```cpp
d1 < d2
```

yields `false`. In both cases, `d1` gets expanded to 5 ticks of \(\frac{1}{15}\) second, and `d2` gets expanded to 3 ticks of \(\frac{1}{15}\) second. So the sum of 3 and 5 is 8, and 5 is not less than 3.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>( d1 + d2 )</td>
<td>Process sum of durations ( d1 ) and ( d2 )</td>
</tr>
<tr>
<td>( d1 - d2 )</td>
<td>Process difference of durations ( d1 ) and ( d2 )</td>
</tr>
<tr>
<td>( d * val )</td>
<td>Return result of ( val ) times duration ( d )</td>
</tr>
<tr>
<td>( val * d )</td>
<td>Return result of ( val ) times duration ( d )</td>
</tr>
<tr>
<td>( d / val )</td>
<td>Return of the duration ( d ) divided by value ( val )</td>
</tr>
<tr>
<td>( d1 / d2 )</td>
<td>Compute factor between durations ( d1 ) and ( d2 )</td>
</tr>
<tr>
<td>( d % val )</td>
<td>Result of duration ( d ) modulo value ( val )</td>
</tr>
<tr>
<td>( d % d2 )</td>
<td>Result of duration ( d ) modulo the value of ( d2 )</td>
</tr>
<tr>
<td>( d1 == d2 )</td>
<td>Return whether duration ( d1 ) is equal to duration ( d2 )</td>
</tr>
<tr>
<td>( d1 != d2 )</td>
<td>Return whether duration ( d1 ) differs from duration ( d2 )</td>
</tr>
<tr>
<td>( d1 &lt; d2 )</td>
<td>Return whether duration ( d1 ) is shorter than duration ( d2 )</td>
</tr>
<tr>
<td>( d1 &lt;= d2 )</td>
<td>Return whether duration ( d1 ) is not longer than duration ( d2 )</td>
</tr>
<tr>
<td>( d1 &gt; d2 )</td>
<td>Return whether duration ( d1 ) is longer than duration ( d2 )</td>
</tr>
<tr>
<td>( d1 &lt;= d2 )</td>
<td>Return whether duration ( d1 ) is not shorter than duration ( d2 )</td>
</tr>
<tr>
<td>( ++d )</td>
<td>Increment duration ( d ) by 1 tick</td>
</tr>
<tr>
<td>( d++ )</td>
<td>Increment duration ( d ) by 1 tick</td>
</tr>
<tr>
<td>( --d )</td>
<td>Decrement duration ( d ) by 1 tick</td>
</tr>
<tr>
<td>( d-- )</td>
<td>Decrement duration ( d ) by 1 tick</td>
</tr>
<tr>
<td>( d += d1 )</td>
<td>Extend the duration ( d ) by the duration ( d1 )</td>
</tr>
<tr>
<td>( d -= d1 )</td>
<td>Shorten the duration ( d ) by the duration ( d1 )</td>
</tr>
<tr>
<td>( d *= val )</td>
<td>Multiply the duration ( d ) by ( val )</td>
</tr>
<tr>
<td>( d /= val )</td>
<td>Divide the duration ( d ) by ( val )</td>
</tr>
<tr>
<td>( d %= val )</td>
<td>Process duration ( d ) modulo ( val )</td>
</tr>
<tr>
<td>( d %= d2 )</td>
<td>Process duration ( d ) modulo the value of ( d2 )</td>
</tr>
</tbody>
</table>

Table 5.21. Arithmetic Operations of durations

You can also convert durations into durations of different units, as long as there is an implicit type conversion. Thus, you can convert hours into seconds but not the other way around. For example:

```cpp
std::chrono::seconds twentySeconds(20); // 20 seconds
std::chrono::hours aDay(24); // 24 hours

std::chrono::milliseconds ms; // 0 milliseconds
ms += twentySeconds + aDay; // 86,400,000 milliseconds
--ms; // 86,399,999 milliseconds
ms *= 2; // 172,839,998 milliseconds
std::cout << ms.count() << " ms" << std::endl;
std::cout << std::chrono::nanoseconds(ms).count() << " ns" << std::endl;
```

These conversions result in the following output:

```
172839998 ms
172839998000000 ns
```
Other Duration Operations

In the preceding example, we use the member `count()` to yield the current number of ticks, which is one of the other operations provided for durations. Table 5.22 lists all operations, members, and types available for durations besides the arithmetic operations of Table 5.21. Note that the default constructor default-initializes (see Section 3.2.1, page 37) its value, which means that for fundamental representation types, the initial value is undefined.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>duration d</code></td>
<td>Default constructor; creates duration (default-initialized)</td>
</tr>
<tr>
<td><code>duration d(d2)</code></td>
<td>Copy constructor; copies duration ((d2) might have a different unit type)</td>
</tr>
<tr>
<td><code>duration d(val)</code></td>
<td>Creates duration of (val) ticks of (ds) unit type</td>
</tr>
<tr>
<td><code>d = d2</code></td>
<td>Assigns duration (d2) to (d) (implicit conversion possible)</td>
</tr>
<tr>
<td><code>d.count()</code></td>
<td>Returns ticks of the duration (d)</td>
</tr>
<tr>
<td><code>duration_cast&lt;D&gt;(d)</code></td>
<td>Returns duration (d) explicitly converted into type (D)</td>
</tr>
<tr>
<td><code>duration::zero()</code></td>
<td>Yields duration of zero length</td>
</tr>
<tr>
<td><code>duration::max()</code></td>
<td>Yields maximum possible duration of this type</td>
</tr>
<tr>
<td><code>duration::min()</code></td>
<td>Yields minimum possible duration of this type</td>
</tr>
<tr>
<td><code>duration::rep</code></td>
<td>Yields the type of the ticks</td>
</tr>
<tr>
<td><code>duration::period</code></td>
<td>Yields the type of the unit type</td>
</tr>
</tbody>
</table>

Table 5.22. Other Operations and Types of durations

You can use these members to define a convenience function for the output operator \(\ll\) for durations:

```cpp
template<typename V, typename R>
ostream& operator<<(ostream& s, const std::chrono::duration<V,R>& d)
{
    s << "[" << d.count() << " of " << R::num << "/"
        << R::den << "]";
    return s;
}
```

Here, after printing the number of ticks with `count()`, we print the numerator and denominator of the unit type used, which is a ratio processed at compile time (see Section 5.6, page 140). For example,

```cpp
std::chrono::milliseconds d(42);
std::cout << d << std::endl;
```

will then print:

```
[42 of 1/1000]
```

---

28 Note that this output operator does not work where ADL (argument-dependent lookup) does not work (see Section 15.11.1, page 812, for details).
As we have seen, implicit conversions to a more precise unit type are always possible. However, conversions to a coarser unit type are not, because you might lose information. For example, when converting an integral value of 42,010 milliseconds into seconds, the resulting integral value, 42, means that the precision of having a duration of 10 milliseconds over 42 seconds gets lost. But you can still explicitly force such a conversion with a `duration_cast`. For example:

```cpp
std::chrono::seconds sec(55);
std::chrono::minutes m1 = sec; // ERROR
std::chrono::minutes m2 =
    std::chrono::duration_cast<std::chrono::minutes>(sec); // OK
```

As another example, converting a duration with a floating-point tick type also requires an explicit cast to convert it into an integral duration type:

```cpp
std::chrono::duration<double,std::ratio<60>> halfMin(0.5);
std::chrono::seconds s1 = halfMin; // ERROR
std::chrono::seconds s2 =
    std::chrono::duration_cast<std::chrono::seconds>(halfMin); // OK
```

A typical example is code that segments a duration into different units. For example, the following code segments a duration of milliseconds into the corresponding hours, minutes, seconds, and milliseconds (to output the first line starting with `raw:` we use the output operator just defined):

```cpp
using namespace std;
using namespace std::chrono;
milliseconds ms(7255042);
// split into hours, minutes, seconds, and milliseconds
hours hh = duration_cast<hours>(ms);
minutes mm = duration_cast<minutes>(ms % chrono::hours(1));
seconds ss = duration_cast<seconds>(ms % chrono::minutes(1));
milliseconds msec = duration_cast<milliseconds>(ms % chrono::seconds(1));
// and print durations and values:
cout << "raw: " << hh << ":" << mm << ":" << ss << ":" << msec << endl;
```

Here, the cast

```cpp
std::chrono::duration_cast<std::chrono::hours>(ms)
```

converts the milliseconds into hours, where the values are truncated, not rounded. Thanks to the modulo operator `%`, for which you can even pass a duration as second argument, you can easily
process the remaining milliseconds with `ms % std::chrono::hours(1)`, which is then converted into minutes. Thus, the output of this code will be as follows:

```
raw: [2 of 3600/1]: [0 of 60/1]: [55 of 1/1]: [42 of 1/1000]
02:00:55:042
```

Finally, class `duration` provides three static functions: `zero()`, which yields a duration of 0 seconds, as well as `min()` and `max()`, which yield the minimum and maximum value a duration can have.

### 5.7.3 Clocks and Timepoints

The relationships between timepoints and clocks are a bit tricky:

- A **clock** defines an epoch and a tick period. For example, a clock might tick in milliseconds since the UNIX epoch (January 1, 1970) or tick in nanoseconds since the start of the program. In addition, a clock provides a type for any timepoint specified according to this clock.

  The interface of a clock provides a function `now()` to yield an object for the current point in time.

- A **timepoint** represents a specific point in time by associating a positive or negative duration to a given clock. Thus, if the duration is “10 days” and the associated clock has the epoch of January 1, 1970, the timepoint represents January 11, 1970.

  The interface of a timepoint provides the ability to yield the epoch, minimum and maximum timepoints according to the clock, and timepoint arithmetic.

### Clocks

Table 5.23 lists the type definitions and static members required for each clock.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>clock::duration</code></td>
<td>Yields the duration type of the clock</td>
</tr>
<tr>
<td><code>clock::rep</code></td>
<td>Yields the type of the ticks (equivalent to <code>clock::duration::rep</code>)</td>
</tr>
<tr>
<td><code>clock::period</code></td>
<td>Yields the type of the unit type (equivalent to <code>clock::duration::period</code>)</td>
</tr>
<tr>
<td><code>clock::time_point</code></td>
<td>Yields the timepoint type of the clock</td>
</tr>
<tr>
<td><code>clock::is_steady</code></td>
<td>Yields true if the clock is steady</td>
</tr>
<tr>
<td><code>clock::now()</code></td>
<td>Yields a <code>time_point</code> for the current point in time</td>
</tr>
</tbody>
</table>

*Table 5.23. Operations and Types of Clocks*

The C++ standard library provides three clocks, which provide this interface:

1. The **`system_clock`** represents timepoints associated with the usual real-time clock of the current system. This clock also provides convenience functions `to_time_t()` and `from_time_t()`
1. To convert between any timepoint and the C system time type `time_t`, which means that you can convert into and from calendar times (see Section 5.7.4, page 158).

2. The **steady_clock** gives the guarantee that it never gets adjusted.\(^\text{29}\) Thus, timepoint values never decrease as the physical time advances, and they advance at a steady rate relative to real time.

3. The **high_resolution_clock** represents a clock with the shortest tick period possible on the current system.

Note that the standard does not provide requirements for the precision, the epoch, and the range (minimum and maximum timepoints) of these clocks. For example, your system clock might have the UNIX epoch (January 1, 1970) as epoch, but this is not guaranteed. If you require a specific epoch or care for timepoints that might not be covered by the clock, you have to use convenience functions to find it out.

For example, the following function prints the properties of a clock:

```cpp
#include <chrono>
#include <iostream>
#include <iomanip>

template <typename C>
void printClockData ()
{
    using namespace std;

    cout << "- precision: ";
    // if time unit is less or equal one millisecond
    typedef typename C::period P;  // type of time unit
    if (ratio_less_equal<P,milli>::value) {
        // convert to and print as milliseconds
        typedef typename ratio_multiply<P,kilo>::type TT;
        cout << fixed << double(TT::num)/TT::den
            << " milliseconds" << endl;
    }
    else {
        // print as seconds
        cout << fixed << double(P::num)/P::den << " seconds" << endl;
    }
    cout << "- is_steady: " << boolalpha << C::is_steady << endl;
}
```

We can call this function for the various clocks provided by the C++ standard library:

\(^{29}\) The `steady_clock` was initially proposed as `monotonic_clock`. 
The program might, for example, have the following output:

```
    system_clock:
    - precision: 0.000100 milliseconds
    - is_steady: false

    high_resolution_clock:
    - precision: 0.000100 milliseconds
    - is_steady: true

    steady_clock:
    - precision: 1.000000 milliseconds
    - is_steady: true
```

Here, for example, the system and the high-resolution clock have the same precision of 100 nanoseconds, whereas the steady clock uses milliseconds. You can also see that both the steady clock and high-resolution clock can’t be adjusted. Note, however, that this might be very different on other systems. For example, the high-resolution clock might be the same as the system clock.

The steady_clock is important to compare or compute the difference of two times in your program, where you processed the current point in time. For example, after

```
auto system_start = chrono::system_clock::now();
```

a condition to check whether the program runs more than one minute:

```
if (chrono::system_clock::now() > system_start + minutes(1))
```

might not work, because if the clock was adjusted in the meantime, the comparison might yield false, although the program did run more than a minute. Similarly, processing the elapsed time of a program:

```
auto diff = chrono::system_clock::now() - system_start;
auto sec = chrono::duration_cast<chrono::seconds>(diff);
cout << "this program runs: " << s.count() << " seconds" << endl;
```
might print a negative duration if the clock was adjusted in the meantime. For the same reason, using timers with other than the steady_clock might change their duration when the system clock gets adjusted (see Section 5.7.5, page 160, for details).

**Timepoints**

With any of these clocks — or even with user-defined clocks — you can deal with timepoints. Class time_point provides the corresponding interface, parametrized by a clock:

```cpp
namespace std {
    namespace chrono {
        template<typename Clock,
            typename Duration=typename Clock::duration>
        class time_point;
    }
}
```

Four specific timepoints play a special role:

1. The epoch, which the default constructor of class time_point yields for each clock.
2. The current time, which the static member function now() of each clock yields (see Section 5.7.3, page 149).
3. The minimum timepoint, which the static member function min() of class time_point yields for each clock.
4. The maximum timepoint, which the static member function max() of class time_point yields for each clock.

For example, the following program assigns these timepoints to tp and prints them converted into a calendar notation:

```cpp
// util/chrono1.cpp
#include <chrono>
#include <ctime>
#include <string>
#include <iostream>

std::string asString (const std::chrono::system_clock::time_point& tp) {
    // convert to system time:
    std::time_t t = std::chrono::system_clock::to_time_t(tp);
    std::string ts = std::ctime(&t); // convert to calendar time
    ts.resize(ts.size()-1); // skip trailing newline
    return ts;
}

int main()
{

// print the epoch of this system clock:
std::chrono::system_clock::time_point tp;
std::cout << "epoch: " << asString(tp) << std::endl;

// print current time:
tp = std::chrono::system_clock::now();
std::cout << "now: " << asString(tp) << std::endl;

// print minimum time of this system clock:
tp = std::chrono::system_clock::time_point::min();
std::cout << "min: " << asString(tp) << std::endl;

// print maximum time of this system clock:
tp = std::chrono::system_clock::time_point::max();
std::cout << "max: " << asString(tp) << std::endl;

After including <chrono>, we first declare a convenience function asString(), which converts a
timepoint of the system clock into the corresponding calendar time. With

    std::time_t t = std::chrono::system_clock::to_time_t(tp);

we use the static convenience function to_time_t(), which converts a timepoint into an object of
the traditional time type of C and POSIX, type time_t, which usually represents the number of
seconds since the UNIX epoch, January 1, 1970 (see Section 5.7.4, page 157). Then,

    std::string ts = std::ctime(&t);

uses ctime() to convert this into a calendar notation, for which

    ts.resize(ts.size()-1);

removes the trailing newline character.

Note that ctime() takes the local time zone into account, which has consequences we will dis-
cuss shortly. Note also that this convenience function probably will work only for system_clocks,
the only clocks that provide an interface for conversions to and from time_t. For other clocks, such
an interface might also work but is not portable, because the other clocks are not required to have
epoch of the system time as their internal epoch.

Note also that the output format for timepoints might better get localized by using the time_put
facet. See Section 16.4.3, page 884, for details, and page 886 for an example.

Inside main(), the type of tp, declared as

    std::chrono::system_clock::time_point

is equivalent to:30

    std::chrono::time_point< std::chrono::system_clock>

30 According to the standard, a system_clock::time_point could also be identical to
time_point<C2,system_clock::duration>, where C2 is a different clock but has the same epoch
as system_clock.
Thus, tp is declared as the timepoint of the system_clock. Having the clock as template argument ensures that only timepoint arithmetic with the same clock (epoch) is possible.

The program might have the following output:

epoch: Thu Jan  1 01:00:00 1970
now: Sun Jul 24 19:40:46 2011
min: Sat Mar 5 18:27:38 1904
max: Mon Oct 29 07:32:22 2035

Thus, the default constructor, which yields the epoch, creates a timepoint, which asString() converts into

Thu Jan  1 01:00:00 1970

Note that it’s 1 o’clock rather than midnight. This may look a bit surprising, but remember that the conversion to the calendar time with ctime() inside asString() takes the time zone into account. Thus, the UNIX epoch used here — which, again, is not always guaranteed to be the epoch of the system time — started at 00:00 in Greenwich, UK. In my time zone, Germany, it was 1 a.m. at that moment, so in my time zone the epoch started at 1 a.m. on January 1, 1970. Accordingly, if you start this program, your output is probably different, according to your time zone, even if your system uses the same epoch in its system clock.

To have the universal time (UTC) instead, you should use the following conversion rather than calling ctime(), which is a shortcut for asctime(localtime(...)) (see Section 5.7.4, page 157):

```
std::string ts = std::asctime(gmtime(&t));
```

In that case, the output of the program would be:

epoch: Thu Jan  1 00:00:00 1970
now: Sun Jul 24 17:40:46 2011
min: Sat Mar 5 17:27:38 1904
max: Mon Oct 29 06:32:22 2035

Yes, here, the difference is 2 hours for now(), because this timepoint is when summer time is used, which leads to a 2-hour difference to UTC in Germany.

In general, time_point objects have only one member, the duration, which is relative to the epoch of the associated clock. The timepoint value can be requested by time_since_epoch(). For timepoint arithmetic, any useful combination of a timepoint and another timepoint or duration is provided (see Table 5.24).

Although the interface uses class ratio (see Section 5.6, page 140), which ensures that overflows by the duration units yield a compile-time error, overflows on the duration values are possible. Consider the following example:

```
// util/chrono2.cpp
#include <chrono>
#include <ctime>
#include <iostream>
#include <string>
using namespace std;
```
<table>
<thead>
<tr>
<th>Operation</th>
<th>Yields</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>timepoint ( t )</td>
<td>timepoint</td>
<td>Default constructor; creates a timepoint representing the epoch</td>
</tr>
<tr>
<td>timepoint ( t(tp2) )</td>
<td>timepoint</td>
<td>Creates a timepoint equivalent to ( tp2 ) (the duration unit might be finer grained)</td>
</tr>
<tr>
<td>timepoint ( t(d) )</td>
<td>timepoint</td>
<td>Creates a timepoint having duration ( d ) after the epoch</td>
</tr>
<tr>
<td>\text{time_point_cast(&lt;C,D&gt;}(tp)</td>
<td>timepoint</td>
<td>Converts ( tp ) into a timepoint with clock ( C ) and duration ( D ) (which might be more coarse grained)</td>
</tr>
<tr>
<td>( tp += d )</td>
<td>timepoint</td>
<td>Adds duration ( d ) to the current timepoint ( tp )</td>
</tr>
<tr>
<td>( tp -= d )</td>
<td>timepoint</td>
<td>Subtracts duration ( d ) from the current timepoint ( tp )</td>
</tr>
<tr>
<td>( tp + d )</td>
<td>timepoint</td>
<td>Returns a new timepoint of ( tp ) with duration ( d ) added</td>
</tr>
<tr>
<td>( d + tp )</td>
<td>timepoint</td>
<td>Returns a new timepoint of ( tp ) with duration ( d ) added</td>
</tr>
<tr>
<td>( tp - d )</td>
<td>timepoint</td>
<td>Returns a new timepoint of ( tp ) with duration ( d ) subtracted</td>
</tr>
<tr>
<td>( tp1 - tp2 )</td>
<td>duration</td>
<td>Returns the duration between timepoints ( tp1 ) and ( tp2 )</td>
</tr>
<tr>
<td>( tp1 == tp2 )</td>
<td>bool</td>
<td>Returns whether timepoint ( tp1 ) is equal to timepoint ( tp2 )</td>
</tr>
<tr>
<td>( tp1 != tp2 )</td>
<td>bool</td>
<td>Returns whether timepoint ( tp1 ) differs from timepoint ( tp2 )</td>
</tr>
<tr>
<td>( tp1 &lt; tp2 )</td>
<td>bool</td>
<td>Returns whether timepoint ( tp1 ) is before timepoint ( tp2 )</td>
</tr>
<tr>
<td>( tp1 &lt;= tp2 )</td>
<td>bool</td>
<td>Returns whether timepoint ( tp1 ) is not after timepoint ( tp2 )</td>
</tr>
<tr>
<td>( tp1 &gt; tp2 )</td>
<td>bool</td>
<td>Returns whether timepoint ( tp1 ) is after timepoint ( tp2 )</td>
</tr>
<tr>
<td>( tp1 &gt;= tp2 )</td>
<td>bool</td>
<td>Returns whether timepoint ( tp1 ) is not before timepoint ( tp2 )</td>
</tr>
<tr>
<td>( tp\text{.time_since_epoch()} )</td>
<td>duration</td>
<td>Returns the duration between the epoch and timepoint ( tp )</td>
</tr>
<tr>
<td>\text{timepoint}::\text{min()}</td>
<td>timepoint</td>
<td>Returns the first possible timepoint of type ( \text{timepoint} )</td>
</tr>
<tr>
<td>\text{timepoint}::\text{max()}</td>
<td>timepoint</td>
<td>Returns the last possible timepoint of type ( \text{timepoint} )</td>
</tr>
</tbody>
</table>

Table 5.24. Operations of time_points
string asString (const chrono::system_clock::time_point& tp)
{
    time_t t = chrono::system_clock::to_time_t(tp);   // convert to system time
    string ts = ctime(&t);                            // convert to calendar time
    ts.resize(ts.size()-1);                          // skip trailing newline
    return ts;
}

int main()
{
    // define type for durations that represent day(s):
    typedef chrono::duration<int, ratio<3600*24>> Days;

    // process the epoch of this system clock
    chrono::time_point<chrono::system_clock> tp;
    cout << "epoch: " << asString(tp) << endl;

    // add one day, 23 hours, and 55 minutes
    tp += Days(1) + chrono::hours(23) + chrono::minutes(55);
    cout << "later: " << asString(tp) << endl;

    // process difference from epoch in minutes and days:
    auto diff = tp - chrono::system_clock::time_point();
    cout << "diff: " << asString(diff) << endl;
    Days days = chrono::duration_cast<Days>(diff);
    cout << "diff: " << days.count() << " day(s)" << endl;

    // subtract one year (hoping it is valid and not a leap year)
    tp -= chrono::hours(24*365);
    cout << "-1 year: " << asString(tp) << endl;

    // subtract 50 years (hoping it is valid and ignoring leap years)
    tp -= chrono::duration<int, ratio<3600*24*365>>(50);
    cout << "-50 years: " << asString(tp) << endl;

    // subtract 50 years (hoping it is valid and ignoring leap years)
    tp -= chrono::duration<int, ratio<3600*24*365>>(50);
    cout << "-50 years: " << asString(tp) << endl;
}
First, expressions, such as
\[
\text{tp} = \text{tp} + \text{Days}(1) + \text{chrono}::\text{hours}(23) + \text{chrono}::\text{minutes}(55);
\]
or
\[
\text{tp} -= \text{chrono}::\text{hours}(24*365);
\]
allow adjusting timepoints by using timepoint arithmetic.

Because the precision of the system clock usually is better than minutes and days, you have to explicitly cast the difference between two timepoints to become days:

\[
\text{auto \ diff = tp - chrono::system\_clock::time\_point();}
\]
\[
\text{Days days = chrono::duration\_cast<Days>(diff);}\]

Note, however, that these operation do not check whether a combination performs an overflow. On my system, the output of the program is as follows:

epoch: Thu Jan 1 01:00:00 1970
later: Sat Jan 3 00:55:00 1970
diff: 2875 minute(s)
diff: 1 day(s)
-1 year: Fri Jan 3 00:55:00 1969
-50 years: Thu Jan 16 00:55:00 1919
-50 years: Sat Mar 5 07:23:16 2005

You can see the following:

- The cast uses \text{static\_cast<>} for the destination unit, which for ordinary integral unit types means that values are truncated instead of rounded. For this reason, a duration of 47 hours and 55 minutes converts into 1 day.
- Subtracting 50 years of 365 days does not take leap years into account, so the resulting day is January 16 instead of January 3.
- When deducting another 50 years the timepoint goes below the minimum timepoint, which is March 5, 1904 on my system (see Section 5.7.3, page 152), so the result is the year 2005. No error processing is required by the C++ standard library in this case.

This demonstrates that \text{chrono} is a duration and a timepoint but not a date/time library. You can compute with durations and timepoints but still have to take epoch, minimum and maximum timepoints, leap years, and leap seconds into account.

### 5.7.4 Date and Time Functions by C and POSIX

The C++ standard library also provides the standard C and POSIX interfaces to deal with date and time. In \text{<ctime>}, the macros, types, and functions of \text{<time.h>} are available in namespace std. The types and functions are listed in Table 5.25. In addition, the macro \text{CLOCKS\_PER\_SEC} defines the unit type of \text{clock()} (which returns the elapsed CPU time in \( \frac{1}{\text{CLOCKS\_PER\_SEC}} \) seconds).

See Section 16.4.3, page 884, for some more details and examples using these time functions and types.
<table>
<thead>
<tr>
<th>Identifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock_t</td>
<td>Type of numeric values of elapsed CPU time returned by clock()</td>
</tr>
<tr>
<td>time_t</td>
<td>Type of numeric values representing timepoints</td>
</tr>
<tr>
<td>struct tm</td>
<td>Type of “broken down” calendar time</td>
</tr>
<tr>
<td>clock()</td>
<td>Yields the elapsed CPU time in $\frac{1}{\text{CLOCKS}_s \text{PER_SEC}}$ seconds</td>
</tr>
<tr>
<td>time()</td>
<td>Yields the current time as numeric value</td>
</tr>
<tr>
<td>difftime()</td>
<td>Yields the difference of two time_t in seconds as double</td>
</tr>
<tr>
<td>localtime()</td>
<td>Converts a time_t into a struct tm taking time zone into account</td>
</tr>
<tr>
<td>gmtime()</td>
<td>Converts a time_t into a struct tm not taking time zone into account</td>
</tr>
<tr>
<td>asctime()</td>
<td>Converts a struct tm into a standard calendar time string</td>
</tr>
<tr>
<td>strftime()</td>
<td>Converts a struct tm into a user-defined calendar time string</td>
</tr>
<tr>
<td>ctime()</td>
<td>Converts a time_t into a standard calendar time string taking time zone into account (shortcut for asctime(localtime(t)))</td>
</tr>
<tr>
<td>mktime()</td>
<td>Converts a struct tm into a time_t and queries weekday and day of the year</td>
</tr>
</tbody>
</table>

Table 5.25. Definitions in <ctime>

Note that time_t usually is just the number of seconds since the UNIX epoch, which is January 1, 1970. However, according to the C and C++ standard, this is not guaranteed.

Conversions between Timepoints and Calendar Time

The convenience function to transfer a timepoint to a calendar time string was already discussed in Section 5.7.3, page 153. Here is a header file that also allows converting calendar times into timepoints:

```cpp
// util/timepoint.hpp
#include <chrono>
#include <ctime>
#include <string>

// convert timepoint of system clock to calendar time string
inline
std::string asString (const std::chrono::system_clock::time_point& tp) {
    // convert to system time:
    std::time_t t = std::chrono::system_clock::to_time_t(tp);
    std::string ts = ctime(&t);   // convert to calendar time
    ts.resize(ts.size()-1);       // skip trailing newline
    return ts;
}
```
# convert calendar time to timepoint of system clock

```cpp
inline
std::chrono::system_clock::time_point
makeTimePoint (int year, int mon, int day,
    int hour, int min, int sec=0)
{
    struct std::tm t;
    t.tm_sec = sec;         // second of minute (0 .. 59 and 60 for leap seconds)
    t.tm_min = min;         // minute of hour (0 .. 59)
    t.tm_hour = hour;       // hour of day (0 .. 23)
    t.tm_mday = day;        // day of month (0 .. 31)
    t.tm_mon = mon-1;       // month of year (0 .. 11)
    t.tm_year = year-1900;  // year since 1900
    t.tm_isdst = -1;        // determine whether daylight saving time
    std::time_t tt = std::mktime(&t);
    if (tt == -1) {
        throw "no valid system time";
    }
    return std::chrono::system_clock::from_time_t(tt);
}
```

The following program demonstrates these convenience functions:

```cpp
// util/timepoint1.cpp
#include <chrono>
#include <iostream>
#include "timepoint.hpp"

int main()
{
    auto tp1 = makeTimePoint(2010,01,01,00,00);
    std::cout << asString(tp1) << std::endl;

    auto tp2 = makeTimePoint(2011,05,23,13,44);
    std::cout << asString(tp2) << std::endl;
}
```

The program has the following output:

```
Fri Jan  1 00:00:00 2010
Mon May 23 13:44:00 2011
```

Note again that both `makeTimePoint()` and `asString()` take the local time zone into account. For this reason, the date passed to `makeTimePoint()` matches the output with `asString()`. Also, it doesn’t matter whether daylight saving time is used (passing a negative value to `t.tm_isdst` in
makeTimePoint() causes mktime() to attempt to determine whether daylight saving time is in effect for the specified time).

Again, to let asString() use the universal time UTC instead, use asctime(gmtime(...)) rather than ctime(...). For mktime(), there is no specified way to use UTC, so makeTimePoint() always takes the current time zone into account.

Section 16.4.3, page 884, demonstrates how to use locales to internationalize the reading and writing of time data.

## 5.7.5 Blocking with Timers

Durations and timepoints can be used to block threads or programs (i.e., the main thread). These blocks can be conditionless or can be used to specify a maximum duration when waiting for a lock, a condition variable, or another thread to end (see Chapter 18):

- sleep_for() and sleep_until() are provided by this_thread to block threads (see Section 18.3.7, page 981).
- try_lock_for() and try_lock_until() are provided to specify a maximum interval when waiting for a mutex (see Section 18.5.1, page 994).
- wait_for() and wait_until() are provided to specify a maximum interval when waiting for a condition variable or a future (see Section 18.1.1, page 953 or Section 18.6.4, page 1010).

All the blocking functions that end with ..._for() use a duration, whereas all functions that end with ..._until() use a timepoint as argument. For example,

```cpp
this_thread::sleep_for(chrono::seconds(10));
```
blocks the current thread, which might be the main thread, for 10 seconds, whereas

```cpp
this_thread::sleep_until(chrono::system_clock::now() + chrono::seconds(10));
```
blocks the current thread until the system clock has reached a timepoint 10 seconds later than now.

Although these calls look the same, they are not! For all ..._until() functions, where you pass a timepoint, time adjustments might have an effect. If, during the 10 seconds after calling sleep_until(), the system clock gets adjusted, the timeout will be adjusted accordingly. If, for example, we wind the system clock back 1 hour, the program will block for 60 minutes and 10 seconds. If, for example, we adjust the clock forward for more than 10 seconds, the timer will end immediately.

If you use a ..._for() function, such as sleep_for(), where you pass a duration, or if you use the steady_clock, adjustments of the system clock usually will have no effect on the duration of timers. However, on hardware where a steady clock is not available, and thus the platform gives no chance to count seconds independently of a possibly adjusted system time, time adjustments can also impact the ..._for() functions.

All these timers do not guarantee to be exact. For any timer, there will be a delay because the system only periodically checks for expired timers, and the handling of timers and interrupts takes some time. Thus, durations of timers will take their specified time plus a period that depends on the quality of implementation and the current situation.
Index

Note: Page numbers in **bold** indicate the location of the definition of the item. Page numbers in the normal type face are other pages of interest. If the entry appears in source code the page numbers are in the *italic* type face.

\&
   as capture 29
   &\&, |\=, ^= for atomics 1016
() as operator 233
   for locales 868
* for iterators 188, 191, 435
   for unique_ptrs 111
   for shared_ptrs 79, 94
+ for strings 676, 711
   for iterators 438
   for timepoints 155
   +, -, *, / for complex 933, 938
   for ratios 141
   +, -, *, /, % for durations 146
++, --  for atomics 1016
   for durations 146
   for iterators 188, 435, 437
   for iterators of arrays, vectors, strings 440
   += for strings 674, 702
   +=, -= for atomics 1016
   for iterators 438
   for timepoints 155
   +=, -=, *=, /= for complex 929, 933, 937
   +=, -=, *=, /=, %= for durations 146
-> for iterators 188, 191, 435
   for return types 32
   for unique_ptrs 111
   for shared_ptrs 94
<< 753, 754
   conventions 818
   for bitsets 652
   for complex 927, 933, 938
   for distributions 918
   for pairs 62
   for random-value engines 916
   for stream buffers 846
   for strings 677, 712
   for shared_ptrs 94
   user-defined 810
= as capture 29
   for arrays 265
   for containers 255, 258, 406
   for deques 287
= (continued)
  for durations 147
  for forward lists 303
  for iterators 188
  for lists 293
  for maps and multimaps 336
  for pairs 61
  for sets and multisets 321
  for strings 673
  for tuples 71
  for unique_ptrs 111
  for unordered containers 368
  for vectors 274
  for shared_ptrs 93, 97

==, !=
  != derived from == 138
  for allocators 1024
  for arrays 264
  for complex 932, 939
  for container adapters 649
  for containers 255, 259, 404
  for deques 286
  for distributions 918
  for error categories 49
  for forward lists 303
  for iterators 188, 435, 436
  for lists 292
  for locales 868
  for maps and multimaps 335
  for pairs 61
  for random-value engines 916
  for ratios 141
  for sets and multisets 318
  for strings 697
  for timepoints 155
  for tuples 71
  for unique_ptrs 111
  for vectors 273
  for shared_ptrs 94

>>
  and templates 13
  conventions 818
  for complex 927, 933, 938
  for distributions 918
  for random-value engines 916
  for stream buffers 846
  for strings 677, 712
  user-defined 810

?: 125

[]
  for unordered_maps 186
  for arrays 265, 408
  for deques 286, 408
  for iterators 438
  for maps 186, 343, 408
  for match results 720, 722
  for strings 408, 671
  for unique_ptrs 111
  for unordered maps 374, 408
  for vectors 274, 408

A

abort() 162
abs()
  for complex 931, 938
  global function 226, 942
absolute to relative values 516, 628
accumulate()  
  algorithm 375, 623
acos()   
  for complex 935, 940
  global function 941
acosh()  
  for complex 935, 940
  global function 941
adapter  
  for containers 188, 631
add_const trait 130
add_cv trait 130
add_lvalue_reference trait 130
add_pointer trait 130
address  
  I/O 756
address_family_not_supported 46
address_in_use 46
address_not_available 46
add_rvalue_reference trait 130
add_volatile trait 130
adjacent_difference()  
  algorithm 628, 630
adjacent_find()  
  algorithm 540
adjustfield 782
ADL 677, 812
adopt_lock 994, 999
  for unique_locks 1000
advance() 441, 442, 465, 586
algorithm 166, 199, 505
  absolute to relative values 516, 628
accumulate() 375, 623
adjacent_difference() 628, 630
adjacent_find() 540
all_of() 555
and lambda 206, 229
and maps 345
and regular expressions 727
any_of() 555
auxiliary functions 517
binary_search() 608
C++98/C++03 example 201
change order of elements 583
comparing 542
complexity 10
copy() 454, 557
copy and modify elements 563
copy_backward() 557
copy elements 557
copy_if() 557
copy_n() 557
count() 524, 831
count_if() 524
destination 217
equal() 542
equal_range() 613
fill() 568
fill_n() 568
find() 200, 528
find_end() 537
find_first_of() 538
find_if() 226, 350, 528
find_if_not() 528
for_each() 482, 519
for sorted ranges 515
function as argument 224
generate() 478, 569
generate_n() 478, 569
header file 505
heap 604
includes() 609
inner_product() 625
inplace_merge() 622
intersection 617
iota() 571
is_heap() 554
is_heap_until() 554
is_partitioned() 552
is_permutation() 544
is_sorted() 550
is_sorted_until() 550
lexicographical_compare() 548
lower_bound() 611
make_heap() 514, 604, 606, 644
manipulating 217
max_element() 200, 525
merge() 614
min_element() 200, 525
minmax_element() 526
algorithm (continued)

mismatch() 546
modifying 217, 509, 557, 568
move() 561
move_backward() 561
multiple ranges 207
mutating 511, 583
next_permutation() 587
none_of() 555
nonmodifying 507, 524
nth_element() 602
numeric 515, 623
overview 505
partial_sort() 514, 599
partial_sort_copy() 600
partial_sum() 627, 630
partition() 592
partition_copy() 594
partition_point() 552
pop_heap() 605, 606, 644
prev_permutation() 587
push_heap() 605, 606, 644
random_shuffle() 589
range predicates 550
ranges 203
relative to absolute values 516, 627
remove() 575
remove_copy() 577
remove_copy_if() 577
remove_if() 483, 575
removing duplicates 578
removing elements 218, 511, 575
replace() 571
replace_copy() 573
replace_copy_if() 573
replace_if() 571
result 479
reverse() 200, 583
reverse_copy() 583
rotate() 584
rotate_copy() 585
search() 534, 684
searching elements 507, 528
search_n() 531
search_n_if() 533
set_difference() 618
set_intersection() 617
set_symmetric_difference() 619
set_union() 616
shuffle() 589, 908
sort() 200, 228, 512, 596
sorted-ranges 608
sort_heap() 514, 605, 606
sorting 511, 596
stable_partition() 592
stable_sort() 514, 596
suffix_copy 507
suffix_if 507
swap_ranges() 566
transform() 225, 240, 563, 564, 684
union elements 616
unique() 578
unique_copy() 580
unordered comparison 544
upper_bound() 611
user-defined 308, 468
versus member functions 223
<algorithm> 59, 134, 136, 200, 505
algostuff.hpp 517
aliasing constructor
  for shared_ptrs 95
alias template 27, 1024
aligned_storage trait 131
aligned_union trait 131
alignment_of trait 131
allocate()
  for allocators 1024
allocate_shared()
  for shared_ptrs 93
allocator 57, 1023
  ==, != 1024
allocate() 1024
construct() 1024
deallocate() 1024
default 57
destroy() 1024
get_allocator() 1024
user-defined 1024
uses_allocator trait 128
value_type 1026
allocator_arg 977
allocator_type
  for containers 430
  for strings 715
all_of()
  algorithm 555
alnum
  for ctype_base 894
alpha
  for ctype_base 894
already_connected 46
always_noconv()
  for codecvt facets 898
amortized complexity 10
antisymmetric 315
any_of()
  algorithm 555
app stream flag 796
  for string streams 806
append()
  for strings 674, 702, 703
arg()
  for complex 931, 938
argc 37, 797
argument-dependent lookup 677, 812
argument_list_too_long 46
argument_out_of_domain 46
argv 37, 797
array
  ++, -- for iterators 440
  and range-based for loops 18
  associative 185, 343, 374
  as STL container 261, 267, 270, 278, 386
    see STL container 261, 267, 270, 278, 386
    see vector
    see begin() and end() 257, 386
fill() 407
traits 130
array 171, 261, 268
  see C-style array 267
at() 265
back() 265
begin() 266
cbegin() 266
cend() 266
constructor 263, 264
continuity of elements 267
crbegin() 266
crend() 266
data() 267
destructor 263
element access 265
empty() 264
derend() 266
exception handling 268
fill() 265
front() 265
get() 268
header file 261
initialization 262
iterators 266
max_size() 264
move semantics 263
rbegin() 266
rend() 266
size() 264
swap() 263, 265
tuple interface 268
zero sized 263
<array> 261
ASCII 851
asctime() 154, 158
as-if rule 983
asin()
  for complex 935, 940
  global function 941
asinh()
  for complex 935, 940
  global function 941
assign()
  for char_traits 854
  for containers 407
  for deques 287
  for forward lists 303
  for lists 293
  for strings 673, 700, 701
  for vectors 274
assignable 244
assignment
see =
movesemantics 21
associative array 185
with map 343, 346
with unordered map 185, 374
associative container 167, 177
modifying access 221
order of duplicates 180
sorting criterion 232, 933
terminology 168
unordered
see unordered container
user-defined inserter 471
async() 946, 947, 974
and exceptions 951
arguments 958
launch policy 951, 974
polling 954
async launch policy 951, 974
at()
for arrays 265, 408
for containers 408
for deques 286, 408
for maps 186, 343, 408
for strings 671, 699
for unordered maps 186, 374, 408
for vectors 274, 408
atan()
for complex 935, 940
global function 941
atan2()
global function 941
atanh()
for complex 935, 940
global function 941
ate stream flag 796
for string streams 806
atexit() 162
concurrency 56
atomic
&=, |=, ^= 1016
++,- - 1016
+=, -= 1016
CAS operations 1018
compare_exchange_strong() 1016
compare_exchange_weak() 1016
exchange() 1016
fetch_add() 1016
fetch_and() 1016
fetch_or() 1016
fetch_sub() 1016
fetch_xor() 1016
for shared_ptrs 96
is_lock_free() 1016
load() 1012, 1016
low-level interface 1019
memory order 1016, 1020
store() 1012, 1016
versus mutexes 1012
<atomic> 1013, 1016
atomic_bool 1019
atomic_exchange()
for shared_ptrs 97
atomic_init() 1013
atomic_int 1019
atomic_is_lock_free()
for shared_ptrs 97
atomic_load()
for shared_ptrs 97
atomics 1012
atomic_store()
for shared_ptrs 97
at_quick_exit() 162
concurrency 56
attemratio unit 142
auto 14, 192
auto_ptr 113
awk regex grammar 733, 739
B
back()
for arrays 265
for containers 409
for deques 286
for lists 293
for queues 648
for strings 671, 699
for vectors 274
back_inserter 211, 212, 455
backward compatibility 9
bad() for streams 759
bad_address 46
bad_alloc 41, 43
bad_array_new_length 41
badbit 758
bad_cast 41, 42, 43
bad_exception 41, 42
bad_file_descriptor 46
bad_function_call 41, 44, 133
bad_message 46
bad_typeid 41, 42
bad_weak_ptr 41, 44, 89
base() 452
basefield 785
basic_regex grammar 733, 739
basic_filebuf 791, 824
basic_fstream 791
basic_ifstream 791
basic_ios 748
basic_istream 748
basic_istringstream 802
basic_ofstream 791
basic_ostream 748
see output stream
basic_ostringstream 802
basic_regex<> 719
basic_streambuf 748, 832
see input buffer, output buffer
basic_string 655
see string
basic_stringbuf 802
basic_stringstream 802
before_begin() for forward lists 304, 307, 312, 423
beg stream position 800
begin() as global function 386
for arrays 266
for buckets 374, 380, 429
for containers 189, 256, 410
for C-style arrays 257, 386
for deques 286
for forward lists 304
for initializer lists 18
for lists 294
for maps and multimaps 337
for match results 723
for sets and multisets 321
for strings 684, 714
for unordered containers 369, 429
for vectors 276
bernoulli_distribution 917, 921
bibliography 1031
bidirectional iterator 198, 437
    advance() 441
    distance() 445
    next() 443
    prev() 443
bidirectional_iterator_tag 466
Big-O notation 10
binary stream flag 796
binary_function 497
binary numeric representation 652
binary predicate 228
binary_search()
    algorithm 608
    bind() 242, 487, 488, 496
        and pointers 493
        and references 491
        for data members 494
        versus lambdas 499
    bindist() 243, 497
    bind2nd() 241, 243, 497, 525
    binder 241, 487
        deprecated 497
    binomial_distribution 917, 922
    bit_and<>() 486
    bitfield
        and concurrency 982
    bitfield with dynamic size 281
        see vector<bool>
    bitfield with static size 650
        see bitset
    bit_or<>() 486
    bitset 650, 651
    << 652
bitset (continued)
  binary numeric representation 652
  header file 650
  to_string() 652
  to_ullong() 652
<bitset> 650
bit_xor<>() 486
blank
  for ctype_base 894
bool
  input 781
  I/O 755
  numeric limits 116
  output 781
bool()
  for exceptions 49
  for streams 760
  for unique_ptrs 100, 111
  for shared_ptrs 94
boolalpha manipulator 781
boolalpha stream flag 781
Boolean conditions
  in loops 760
  of streams 760
Boolean vector 281
broken_pipe 46
broken_promise 47
bucket() 374
  for unordered containers 429
bucket_count() 374
  for unordered containers 362, 380, 429
bucket interface 429
  for unordered containers 374
bucket_size() 374
  for unordered containers 429
byte order mark 851

C
"C" locale 861
C++
  C++03 7, 193, 202
  C++0x 7
  C++11 7
  C++98 7, 193, 202
  __cplusplus 9
  history 7
  TR1 7
callable object 54, 958
callback
  for streams 817
  call_once() 1000
capacity()
  for strings 427, 670, 696
  for vectors 270, 273, 427
capture group
  for regular expressions 719
capture of lambdas 28, 29
carray 261
case-sensitive search 732
CAS operations
  for atomics 1018
catalog
  for message_base 905
category
  of container iterators 410
  of iterators 198, 433, 469
category()
  for exceptions 49
cauchy_distribution 917, 923
cbefore_begin() 917, 923
cbegin() 192
  for arrays 266
  for vectors 270, 273, 427
  for strings 684, 714
  for unordered containers 369, 429
<ctctype> 896
ceil()
  global function 941
cend() 192
  for arrays 266
for buckets 374, 430
for containers 256, 410
for deques 286
for forward lists 304
for lists 294
for maps and multimaps 337
for match results 723
for sets and multisets 321
for strings 684, 714
for unordered containers 369, 430
for vectors 276

centi ratio unit 142
cerr 745, 751
redirecting 822
<cerrno> 45
<cfloat> 115, 116
char
  classification 891
  input 755
  numeric limits 116
char*
  input 755
char* stream 807
  freeze() 808
  str() 808
char16_t 33, 852
  numeric limits 116
char32_t 33, 852
  numeric limits 116
character
  classification 891
  encoding conversion 897
  sets 851
  traits 689, 853
char_traits 664, 749, 853
  assign() 854
  char_type 854
  compare() 689, 854
  copy() 854
  eof() 854
  eq() 689, 854
  eq_int_type() 854
  find() 689, 854
  int_type 854
  length() 854
lt() 689, 854
move() 854
not_eof() 854
off_type 854
pos_type 854
state_type 854
to_char_type() 854
to_int_type() 854
char_type
  for char_traits 854
chi_squared_distribution 917, 923
<chrono> 143, 153
cin 745, 751
  concurrency 752
  redirecting 822
class
  <<, >> 810
  enum class 32
classic()
  for locales 860, 868
classic_table()
  for ctype facets 895
clear()
  for containers 256, 260, 419
  for deques 287
  for forward lists 306
  for lists 295
  for maps and multimaps 340
  for sets and multisets 322
  for streams 759, 797, 800
  for strings 674, 705
  for unordered containers 370, 371
  for vectors 277
<climits> 115, 116
clock 143, 149
  adjustments 151
  current time 152
  duration 149
  is_steady 149
  now() 149
  period 149
  rep 149
  time_point 149
to_time_t() 153
clock() 158
clock_t 158

clog 745

close()
   for messages facets 905
   for streams 798

<cmath> 941

cntrl
   for ctype_base 894

code() for exceptions 48

<codecvt> 900

codecvt facet 897

codecvt_base 898
   result 898

codecvt_mode 900

codecvt_utf16 900

codecvt_utf8 900

codecvt_utf8_utf16 900

collate facet 904

collate locale category 904

collate regex constant 733

collection 165
   see container
   of collections 548

combine()
   for locales 866

command-line arguments 37, 797

commit-or-rollback 248

common_type trait 124, 131

compare
   lexicographical 548
   ranges 542

compare()
   for char_traits 689, 854
   for collate facets 904
   for strings 698

compare_exchange_strong()
   for atomics 1016

compare_exchange_weak()
   for atomics 1016

compare function
   for unordered containers 366

comparison operators 138

comparisons
   for shared_ptr 92
   for containers 259

for pairs 67
   for unique_ptr 112

compatibility 9

complex 925
   +, -, *, / 933, 938
   +=, -=, *=, /= 929, 933, 937
   << 927, 933, 938
   ==, != 932, 939
   <, <=, >, >= 933
   >> 927, 933, 938
   abs() 931, 938
   acos() 935, 940
   acosh() 935, 940
   and associative containers 932, 933
   arg() 931, 938
   asin() 935, 940
   asinh() 935, 940
   atan() 935, 940
   atanh() 935, 940
   conj() 929, 936
   constructor 929, 936
   cos() 935, 940
   cosh() 935, 940
  的例子 926
   exp() 935, 939
   header file 925
   imag() 931, 937
   input 927, 933, 938
   I/O 927, 933, 938
   log() 935, 940
   log10() 935, 940
   norm() 931, 938
   output 927, 933, 938
   polar() 929, 936
   pow() 935, 939
   proj() 929, 936
   real() 931, 937
   sin() 935, 940
   sinh() 935, 940
   sqrt() 935, 939
   tan() 935, 940
   tanh() 935, 940

type conversions 930

value_type 935

<complex> 925
complexity 10
amortized 10
compressing whitespaces 582
concurrency 55, 945
async() 946, 947, 974
atomics 1012
condition variable 1003, 1009
data race 982
deadlock 992, 995
for shared_ptrs 96
for queues 1006
for streams 56, 752
future 947, 975
guarantees 56
half-written data 985
high-level interface 946
lazy evaluation 951
lock-free programming 988
lock_guard 989, 999
locks 989
low-level interface 964
mutex order 1016, 1020
number of possible threads 980
of containers 985
packaged task 977
polling 954
promises 969, 977
recursive_mutex 993
reordering of statements 986
shared future 976
shared state 969, 973
speculative execution 954
spurious wakeups 1004
STL 56
synchronization 982
this_thread 981
thread 964, 979
timer 947, 981
unique_lock 1000
unsynchronized data access 984
volatile 988, 998
conditional trait 131
conditions in loops 760
condition variable 1003, 1004, 1009
condition_variable 1009
condition_variable_any 1011
constructor 1009
destructor 1009
native_handle() 1009
notify_all() 1009
notify_all_at_thread_exit() 1009
notify_one() 1009
spurious wakeups 1004
wait() 1009
wait_for() 1009
wait_until() 1009
condition_variable 1004
see condition variable
<condition_variable> 1003, 1004, 1009
condition_variable_any 1011
see condition variable
cnj() for complex 929, 936
connection_aborted 46
connection_already_in_progress 46
connection_refused 46
connection_reset 46
constant complexity 10
constant type 127
conestexpr 26, 119
conest_iterator
for containers 190, 260, 398
for strings 694
conest_local_iterator
for unordered containers 399
conest_mem_fun1_ref_t 498
conest_mem_fun1_t 498
conest_mem_fun_ref_t 498
conest_mem_fun_t 498
conest_pointer
for containers 260, 398
for strings 694
conest_pointer_cast() for shared_ptrs 94
conest_reference
for container adapters 645
for containers 260, 397
for strings 693
for vector<bool> 282
const_reverse_iterator 449
  for containers 398
  for strings 694
construct()
  for allocators 1024
constructor
  aliasing 95
  as template 36
  for priority_queues 646, 647
  for shared_ptrs 78, 93, 97
  for arrays 263, 264
  for complex 929, 936
  for condition variables 1009
  for container adapters 646
  for containers 255, 400, 430
  for deques 285
  for durations 147
  for forward lists 302
  for futures 975
  for lists 291, 292
  for locales 866
  for maps 333
  for multimaps 333
  for multisets 316, 327
  for packaged tasks 977
  for pairs 61
  for promises 977
  for sets 316
  for strings 687, 694, 695
  for threads 979
  for timepoints 155
  for tuples 69, 71
  for unique_ptrs 111
  for unordered containers 360
  for vectors 272, 273
  move constructor 21
consume_header 900
container 165, 167, 253, 631
  = 255, 258, 406
  ==, != 255, 404
  <, <=, >, >= 255, 404
  adapters 188, 631
  allocator_type 430
  and concurrency 985
array 261
  see array
assign() 407
at() 408
back() 409
begin() 189, 256, 410
bucket interface 429
C++98/C++03 example 193
call member function for elements 243, 491
cbegin() 192, 256, 410
cend() 192, 256, 410
clear() 256, 260, 419
comparisons 259
concurrency 56
const_iterator 190, 260, 398
const_pointer 260, 398
const_reference 260, 397
const_reverse_iterator 398
constructor 255, 400, 430
count() 404
crbegin() 214, 411
crend() 214, 411
C-style arrays 386
data() 409
deque 283
  see deque
destructor 255, 400, 403
difference_type 260, 398
element access 259
element requirements 244
emplace() 412, 414
emplace_back() 415
emplace_front() 415
emplace_hint() 414
empty() 255, 258, 403
end() 189, 256, 410
equal_range() 406
erase() 417, 418
find() 405
forward_list 300
  see forward_list
front() 409
get_allocator() 430
initialization 254
initializer list 256
insert() 411, 413, 416, 417
internal types 260
container (continued)
  iterator 190
  iterator category 410
  key_type 398
  list 290
  map 331
  mapped_type 399
  max_size() 255, 258, 403
  members 254, 397
  move semantics 257, 258
  multimap 331
    see multimap
  multiset 314
    see multiset
  of containers 548
  overview 392
  performance 394
  pointer 260, 398
  pop_back() 419
  pop_front() 419
  print elements 216
  push_back() 415
  push_front() 414
  rbegin() 214, 411
  reference 260, 397
  references as elements 132, 391
  reference semantics 245, 388
  rend() 214, 411
  requirements 254
  reserve() 428
  resize() 420
  reverse_iterator 398
  set 314
    see set
  shrink_to_fit() 428
  size() 255, 258, 403
  size operations 258
  size_type 260, 398
  swap() 250, 255, 258, 407
  terminology 168
  types 397
unordered container 355
    see unordered_map
    see unordered_multimap
    see unordered_multiset
    see unordered_set
  upper_bound() 405
  user-defined 385
  value semantics 245
  value_type 260, 397
  vector 270
    see vector
    when which 392
container adapter
  ==, != 649
  <, <=, >, >= 649
  constructor 646
  container_type 645
  emplace() 647
  empty() 647
  front() 648
  pop() 648
  push() 647
  reference 645
  size() 647
  size_type 645
  swap() 649
  top() 648
  value_type 645
container_type
  for container adapters 645
conversion
  absolute to relative values 516, 628
  between character encodings 897
  relative to absolute values 516, 627
copy()
  algorithm 557
  algorithm implementation 454
  for char_traits 854
  for strings 669, 700
copyable 244
copy and modify elements 563
copy and replace elements 573
copy_backward() 649
  algorithm 557
copy constructor
  as template 36, 62
  for containers 401

\texttt{copyfmt()}
  for streams 779, 780, 811, 817, 822

\texttt{copyfmt_event} 817

\texttt{copy_if()}
  algorithm 557

\texttt{copy_n()}
  algorithm 557

\texttt{cos()}
  for complex 935, 940
  global function 941

\texttt{cosh()}
  for complex 935, 940
  global function 941

\texttt{count()}
  algorithm 524, 831
  for containers 404
  for durations 147
  for maps and multimaps 335
  for sets and multisets 319
  for unordered containers 368

\texttt{count_if()}
  algorithm 524

\texttt{cout} 745, 751
  concurrency 752
  redirecting 822

\texttt{__cplusplus} 9

\texttt{crbegin() 214, 449}
  for arrays 266
  for containers 411
  for deques 286
  for lists 294
  for maps and multimaps 337
  for sets and multisets 321
  for strings 714
  for unordered containers 369
  for vectors 276

\texttt{crend() 214, 449}
  for arrays 266
  for containers 411
  for deques 286
  for lists 294
  for maps and multimaps 337
  for sets and multisets 321
  for strings 714
  for unordered containers 369
  for vectors 276

\texttt{cross_device_link} 46

\texttt{cstring} 163, 855

\texttt{C-string} 655

\texttt{ctime()} 153, 158

\texttt{<ctime>} 157, 884

\texttt{ctype facet} 891
 \texttt{classic_table()} 895
  \texttt{is()} 891
  \texttt{narrow()} 891
  \texttt{scan_is()} 891
  \texttt{scan_not()} 891
  \texttt{table()} 895
  \texttt{table_size 895}
  \texttt{tolower()} 891
  \texttt{toupper()} 891
  \texttt{widen()} 891

\texttt{ctype locale category} 891

\texttt{ctype_base} 893
  \texttt{mask} 893

\texttt{<ctype.h>} 896

\texttt{curr_symbol()} 800

\texttt{current_exception()} 52, 971

\texttt{current time} 152

\texttt{curr_symbol()}
  for moneypunct facets 874

\texttt{cyclic references} 84
Index D

D

data()
    for arrays 267, 409
    for containers 409
    for strings 669, 700
    for vectors 278, 409

data member adapter 494

data race 977, 982

date
    conversion to/from time_point 158

dateorder
    for time_base 889

date_order()
    for time_get facets 888

deadlock 992
    avoidance 995

deallocate()
    for allocators 1024

dec manipulator 785

dec stream flag 785

dec ratio unit 142

decay trait 131

deci ratio unit 142

decimal numeric representation 785

decimal_point()
    for moneypunct facets 874
    for numnumpunct facets 870

decltype 31, 32, 125, 232, 338, 379, 504, 822

declval 125

default_delete 106

default_error_condition()
    for error category 49
    for exceptions 49

defaultfloat manipulator 788

default initialization 37

default_random_engine 908, 916, 947

defer_lock 996, 1000
    for unique_locks 1000

deferred
    future status 954
    launch policy 951, 974

delete 114
    default_delete 106

for shared_ptrs 80, 82
    for unique_ptrs 107

den
    for ratios 140

denorm_absent 120

denorm_indeterminate 120

denorm_min()
    for numeric limits 118

denorm_present 120

deprecated
    binder 497
    function adapter 497

deqe 170, 283, 288
    see container
    = 287
    ==, != 286
    <, <=, >, >= 286
    [] 286
    assign() 287
    at() 286
    back() 286
    begin() 286
    cbegin() 286
    cend() 286
    clear() 287
    constructor 285
    crbegin() 286
    crow() 286
    destructor 285
    emplace() 287
    emplace_back() 287
    emplace_front() 287
    empty() 286
    end() 286
    erase() 287
    exception handling 288
    front() 286
    header file 283
    insert() 287
    max_size() 286
    pop_back() 287
    pop_front() 287
    push_back() 287
    push_front() 287
    rbegin() 286
deque (continued)
dequeue
    rend() 286
    resize() 287
    shrink_to_fit() 286
    size() 286
    swap() 287
<deque> 283
deriving
    exceptions 54
    from STL 251
destination_address_required 46
destination of algorithms 217
destroy()
    for allocators 1024
destroyable 244
destructor
    for shared_ptrs 93, 97
    for arrays 263
    for condition variables 1009
    for containers 255, 400, 403
    for deques 285
    for forward lists 302
    for futures 975
    for lists 291, 292
    for locales 866
    for maps 333
    for multimaps 333
    for multisets 316
    for packaged tasks 977
    for pairs 61
    for promises 977
    for sets 316
    for strings 696
    for threads 979
    for tuples 71
    for unique_ptrs 111
    for unordered containers 360
    for vectors 272, 273
detach()
    for threads 964, 979
detached thread 967
device_or_resource_busy 46, 999
dictionary
    with multimap 348
    with unordered_multimap 383
difference of two sets 618
difference_type
    for containers 260, 398
    for iterator_traits 467
    for strings 693
difftime() 158
digit
    for ctype_base 894
digits
    for numeric limits 117, 652
digits10
    for numeric limits 117
directory_not_empty 46
discard()
    for random-value engines 915, 916
discard_block_engine 916
discrete_distribution 917, 924
distance() 445, 469
distribution 907, 917
    arguments and parameters 918
    overview 917, 921
    serialization interface 918
    uniform_int_distribution 908
    uniform_real_distribution 908
div()
    global function 942
divides<>() 486
dmy date order 889
domain_error 41, 43
double
    I/O formats 787
    numeric limits 116
doubly linked list 173, 290
    see list
duplicates removing 578
duration 143, 144
    ++, -- 146
    +, -, *, /, % 146
    +=, -=, *=, /=, %= 146
    = 147
    ==, != 146
    <, <=, >, >= 146
    constructor 147
count() 147
duration_cast 147
for clocks 149
hours 145
max() 147
microseconds 145
milliseconds 145
min() 147
minutes 145
nanoseconds 145
period 147
rep 147
seconds 145
zero() 147
duration_cast 148, 151
for durations 147
dynamic_cast 42
dynamic_pointer_cast()
for shared_ptrs 94

eback()
for input buffers 839
ECMAScript regex grammar 733, 738, 739
egptr()
for input buffers 839
egrep regex grammar 732, 733, 739
element access
for arrays 265
for containers 259
for forward lists 304
for lists 292
for vectors 274
emplace()
for container adapters 647
for containers 412, 414
for deques 287
for lists 295
for maps and multimaps 340
for sets and multisets 322
for stacks 634
for unordered containers 370, 371
for vectors 277
emplace_after()
for forward lists 306, 424
emplace_back()
for containers 415
for deques 287
for lists 295
for vectors 277
emplace_front()
for containers 415
for deques 287
for forward lists 306
for lists 295
emplace_hint()
for containers 414
for maps and multimaps 340
for sets and multisets 322
for unordered containers 371
empty()
for arrays 264
for container adapters 647
for containers 175, 255, 258, 403
for deques 286
for forward lists 303
for lists 292
for maps and multimaps 335
for match results 720
for sets and multisets 318
for strings 670, 696
for unordered containers 367
for vectors 273
empty range 189
enable_if trait 131
enable_shared_from_this
for shared_ptrs 90
encoding()
for codecvt facets 898
encoding prefix for string literals 24
end()
as global function 386
for arrays 266
for buckets 374, 380, 430
for containers 189, 256, 410
for C-style arrays 257, 386
for deques 286
for forward lists 304
for initializer lists 18
for lists 294
for maps and multimaps 337
end() (continued)
  for match results 723
  for sets and multisets 321
  for strings 684, 714
  for unordered containers 369, 430
  for vectors 276
end stream position 800
endl manipulator 746, 774, 776, 846
end-of-file 750
end-of-stream iterator 213, 462
ends manipulator 746, 774, 808
engine for random values 909, 912
  <<, >> 916
  ==, != 916
  constructor 916
default_random_engine 908, 916
discard() 916
seed() 916
serialization interface 915, 916
state 912
without distribution 912
enumeration class 32
EOF 750, 758
  internationalized 854
eof()
  for char_traits 854
  for streams 759
eofbit 758
epoch 143, 152
eptr()  
  for output buffers 832
epsilon()  
  for numeric limits 117
eq()  
  for char_traits 689, 854
eq_int_type()  
  for char_traits 854
equal()  
  algorithm 542
  for istreambuf_iterator 830
equal_range()  
  algorithm 613
  for containers 406
  for maps and multimaps 335
  for sets and multisets 319
  for unordered containers 368
equal_to<>() 241, 486
equivalence criterion  
  as lambda 379
  for unordered containers 357, 366, 377
erase()  
  for containers 325, 417, 418
  for deques 287
  for lists 295
  for maps and multimaps 340
  for multisets 327
  for sets 325
  for sets and multisets 322
  for strings 675, 687, 705
  for unordered containers 370, 371
  for unordered sets 375
  for unordered sets and multisets 377
  for vectors 277
erase_after()  
  for forward lists 306, 312, 425
erase_event 817
errc 45
errno 45
<errno.h> 45
error  
  for codecvt_base 899
  error_backref 735
  error_badbrace 735
  error_badrepeat 735
  error_brace 735
  error_brack 735
  errorcategory 50
  error_code 45, 48, 49
  error_collate 735
  error_complexity 735
  error ctype 735
  error escape 735
  error handling 41
  for regular expressions 735
  in the STL 246
  error paren 735
  error_range 735
  error space 735
  error stack 735
evaluation order 983
event 817
event_callback 817
exa ratio unit 142
example code
  auxiliary functions 517
exception 41
  bad_alloc 41
  bad_array_new_length 41
  bad_cast 41
  bad_exception 41
  bad_function_call 41, 133
  bad_typeid 41
  bad_weak_ptr 41
classes 41
  code() 48
  current_exception() 52, 971
  deriving 54
domain_error 41
error_category 49
error_code 48, 49
exception 41
exception_ptr 971
failure 41, 45, 763
  for regular expressions 735
future_error 41, 45
header files 44
invalid_argument 41
ios_base::failure 41, 45, 763
length_error 41
logic_error 41
members 44
noexcept 24
out_of_range 41
overflow_error 41
range_error 41
rethrow_exception() 52, 971
runtime_error 41
safety 248
specification 24, 25, 42
system_error 41, 45
underflow_error 41
user-defined 53, 635
what() 45, 52
exception 41
  see exception
  <exception> 41, 42, 971
exception handling 41
  for arrays 268
  for deques 288
  for forward lists 311
  for lists 296
  for maps and multimaps 345
  for sets and multisets 325
  for unordered containers 375
  for vectors 278
  in the STL 248
  unique_ptr 98
exception_ptr 52, 971
exceptions()
  for streams 762
exchange()
  for atomics 1016
executable_format_error 46
  _Exit() 162
exit() 162
EXIT_FAILURE 162
EXIT_SUCCESS 162
exp()
  for complex 935, 939
  global function 941
expired()
  for shared_ptrs 97
explicit
  and initializer lists 16, 72
exponential_distribution 917, 922
extended regex grammar 733, 739
extending STL 250
extent trait 131
extractor
  for streams 745, 753
extreme_value_distribution 917, 923
F
fabs()
  global function 941
facet 864, 869
  categories 865
codecvt 897
codecvt_base 898
facet (continued)
  collate 904
  ctype 891
  for character classification 891
  for character encoding conversion 897
  for date formatting 884
  for internationalized messages 905
  for monetary formatting 874
  for numeric formatting 870
  for string collation 904
  for time formatting 884
  id 869
  messages 905
  money_base 877
  money_get 881
  moneypunct 874
  money_put 879
  num_get 873
  numpunct 870
  num_put 871
  overview 865
  time_get 887
  time_put 884

fail()
  for streams 759

failbit 758

failed()
  for ostreambuf_iterator 829

failure 41, 45, 763

false_name()
  for numpunct facets 870

false_type 125, 142

femto ratio unit 142

fetch_add()
  for atomics 1016

fetch_and()
  for atomics 1016

fetch_or()
  for atomics 1016

fetch_sub()
  for atomics 1016

fetch_xor()
  for atomics 1016

field width 781

file

access 791, 793
  opening 791
  positioning 799
  read and write 824

filebuf 791, 824

file descriptor 801, 835

file_exists 46

filename_too_long 46

file stream 791
  as return value 795
  class hierarchy 791
  constructor 794
  move semantics 795

file too large 46

fill()
  algorithm 568
  for arrays 265, 407
  for streams 782

fill character 781

fill_n()
  algorithm 568

filter 761, 772

find()
  algorithm 200, 528
  finding subrange 205
  for char_traits 689, 854
  for containers 405
  for maps and multimaps 335
  for multisets 327
  for sets 325
  for sets and multisets 319
  for strings 708, 709
  for unordered containers 368, 373
  for unordered sets 375
  for unordered sets and multisets 377
  return value 205

find_before() 308

find_before_if() 308

find_end()
  algorithm 537

find_first_not_of()
  for strings 709, 710

find_first_of()
  algorithm 538
  for strings 709, 710
find_if()  
algorithm 226, 350, 528
find_if_not()  
algorithm 528
finding algorithms 507, 528
find_last_not_of()  
for strings 710, 711
find_last_of()  
for strings 710, 711
find limit 529
first  
for pairs 61
first_type  
for pairs 60
fisher_f_distribution 917, 924
fixed manipulator 788
fixed stream flag 787
flags  
for streams() 934
flags()  
for streams 779, 780
flip()  
for vector<bool>::reference 281, 282
float  
I/O formats 787
numeric limits 116
float_denorm_style 119
floatfield 787
<float.h> 115, 116
float_round_style 119
floor()  
global function 941
flush() 947
for output streams 771
flush manipulator 746, 774, 846
fmod()  
global function 941
for 193
range-based 17
for_each() 519
algorithm 482
and maps 345
return value 482
versus transform() 509
format_default regex constant 733
format_first_only regex constant 730, 733
format flags 779
format_no_copy regex constant 730, 733
format_sed regex constant 730, 733
formatted I/O 779
formatting  
of bool 755, 781
of floating-point values 787
forward iterator 198, 436
advance() 441
distance() 445
next() 443
step forward 441
forward_iterator_tag 466
forward_list 175, 300, 312
see container  
= 303
==, != 303
<, <=, >, >= 303
assign() 303
before_begin() 304, 307, 312, 423
begin() 304
cbefore_begin() 304, 423
cbegin() 304
cend() 304
clear() 306
constructor 302
destructor 302
element access 304
emplace_after() 306, 424
emplace_front() 306
empty() 303
end() 304
erase_after() 306, 312, 425
exception handling 311
front() 304
header file 300
insert_after() 306, 312, 423, 424
iterators 304
max_size() 303
merge() 310, 312
pop_front() 306
push_front() 306, 312
remove() 305, 306
forward_list (continued)
  remove_if() 305, 306
  resize() 306
  reverse() 310
  size() 301
  sort() 310, 312
  special member functions 420
  splice_after() 309, 310, 425, 426
  swap() 303
  unique() 310, 312
  versus list 300
<forward_list> 300
fpos 799
frac_digits()
  for moneypunct facets 874
fractional arithmetics 140
freeze() 808, 809
frexp()
  global function 941
from_bytes() for wstring_convert<> 901
from_time_t()
  for system_clock 158
front()
  for arrays 265
  for container adapters 648
  for containers 409
  for deques 286
  for forward lists 304
  for lists 293
  for strings 671, 699
  for vectors 274
front_inserter 211, 212, 455, 457
fstream 791
  see file stream
<fstream> 791
function
  as argument 224
  as sorting criterion 228
function 31, 133
function adapter 241, 487
  see function object
deprecated 497
<functional> 44, 66, 70, 132, 133, 240, 356, 364, 486, 505
functional composition 243, 487
function_not_supported 46
function object 233, 351, 475
  as sorting criterion 476
  bind() 487, 496
  bindlist() 497
  bind2nd() 497
  bit_and<>() 486
  bit_or<>() 486
  bit_xor<>() 486
  by reference 480
  divides<>() 486
  equal_to<>() 486
  greater<>() 486
  greater_equal<>() 486
  header file 486
  less<>() 486
  less_equal<>() 486
  logical_and<>() 486
  logical_not<>() 486
  logical_or<>() 486
  mem_fn() 487
  mem_fun() 497
  mem_fun_ref() 497
  minus<>() 486
  modulus<>() 486
  multiplies<>() 486
  negate<>() 486
  not1() 487, 497
  not2() 487, 497
  not_equal_to<>() 486
  plus<>() 486
  predefined 239, 486
  ptr_fun() 497
  state 478, 485
  user-defined 495
  versus lambdas 483, 500, 504
function template 27
functor 233, 475
  see function object
future 947, 975
  constructor 975
  destructor 975
  error category 50
  get() 947, 975
  share() 975
shared 960, 976
valid() 975
wait() 953, 975
wait_for() 953, 975
wait_until() 953, 975
<future> 43, 950, 970, 972
future_already_retrieved 47
future_category() 50
future_errc 45
future_error 41, 43, 45
code() 48
error codes 47
future_status 954

G
gamma_distribution 917, 922
gcount()
  for input streams 769
general inserter 458
generate()
  algorithm 478, 569
generate_header 900
generate_n()
  algorithm 478, 569
generic
  error category 50
generic_category() 50
generic_category() 50
geometric_distribution 917, 922
get()
  for reference_wrappers 391
  for shared_ptrs 94
  for arrays 268
  for futures 947, 975
  for input streams 768, 772, 793
  for messages facets 905
  for money_get facets 881
  for num_get facets 873
  for pairs 61
  for shared futures 976
  for time_get facets 888
  for tuples 74
  for unique_ptrs 111
  future versus shared_future 963
get_allocator() 1024
  for containers 430
  for strings 715
get_buffer 839
  iterator 830
get_date()
  for time_get facets 888
get_deleter()
  for shared_ptrs 94
  for unique_ptrs 111
getenv()
  concurrency 56
get_future()
  for packaged task 972
  for packaged tasks 977
  for promises 971, 977
get_id()
  for this_thread 981
  for threads 967, 979
getline()
  for input streams 769
  for strings 677, 713
getloc() 688
  for stream buffers 827, 833
  for streams 790
get_money() 882
get_monthname()
  for time_get facets 888
get_temporary_buffer() 1029
get_time() 757, 890
get_weekday()
  for time_get facets 888
get_year()
  for time_get facets 888
giga ratio unit 142
global()
  for locales 864, 868
gmtime() 158
good() 762
  for streams 759
goodbit 758
gptr()
  for input buffers 839
grammar
  for regular expressions 738, 739
graph
  for ctype_base 894
greater<>() 325, 486, 525
greater_equal<>() 486
greedy 725, 733
grep regex grammar 733, 739
grouping()
    for moneypunct facets 874
    for numpunct facets 870

H

half-open range 189, 203
half-written data 985
happens-before relationship 1016
hardfail 758
hardware_concurrency() 980
has_denorm
    for numeric limits 118, 119
has_denorm_loss
    for numeric limits 118
has_facet() 867
hash()
    for collate facets 904
hasher
    for unordered containers 399
hash function
    as lambda 379
    for unordered containers 363, 377
hash_function()
    for unordered containers 362, 427
hashing policy 382
hash_map 355
hash_multimap 355
hash_multiset 355
hash_set 355
hash table 355
has_infinity
    for numeric limits 117
has_quiet_NaN
    for numeric limits 117
has_signaling_NaN
    for numeric limits 118
has_virtualDestructor trait 127
header file 40
   "algostuff.hpp" 517
<atomic> 1013, 1016
<bitset> 650
<cctype> 896
<cerrno> 45
<cfloat> 115, 116
<chrono> 143, 153
<climits> 115, 116
<cmath> 941
<codecvt> 900
<complex> 925
<condition_variable> 1003, 1004, 1009
<cstdlib> 14, 161
<cstdio> 82
<cstring> 163
<ctime>
<ctype.h> 896
deque> 283
<errno.h> 45
<exception> 41, 42, 971
extension 40
<float.h> 115, 116
for priority_queues 641
for shared_ptrs 78
for algorithms 505
for arrays 261
for bitsets 650
for complex 925
for deques 283
for exceptions 44
for forward lists 300
for function objects 486
for I/O 752
for lists 290
for maps and multimaps 331
for queues 638
for sets and multisets 314
for stacks 632
for streams 752
for strings 663
for unordered containers 356
for vectors 270
<forward_list> 300
<fstream> 791
heap algorithms 604
heapsort 512, 604
hecto ratio unit 142
hex manipulator 785
hex stream flag 785
hexadecimal numeric representation 785
hexfloat manipulator 788
high_resolution_clock 149
duration 149
is_steady 149
now() 149
period 149
rep 149
time_point 149
history of C++ 7
host_unreachable 46
hours
for durations 145

I

i18n

see internationalization
icase regex constant 733
for regular expressions 732
id
for threads 967, 979
identifier_removed 46
ifstream 791, 793
see file stream and input stream
ignore
for pairs 67
for tuples 72
ignore()
for input streams 770, 777, 927
illegal_byte_sequence 46
imag()
for complex 931, 937
imbue() 860, 934
for streams 790, 860
imbue_event 817
in()
for codecvt facets 898
in stream flag 796
inappropriate_io_control_operation 46
in_avail()  for input buffers  827
include file  see header file
includes()  algorithm  609
independent_bits_engine  916
index operator  see []
infinity()  for numeric limits  117
initialization
by input streams  257
default initialization  37
for arrays  262
list  15
narrowing  15
of containers  254
uniform  15
value initialization  15
zero initialization  37
initializer_list  15, 18
and arrays  262
and C-style arrays  172
and explicit  72
and range-based for loops  18
begin() and end()  18
explicit  16
for a container  256
for tuples  72
inner_product()  algorithm  625
in_place_merge()  algorithm  622
input  743
see stream, input stream
and concurrency  56
binary numeric representation  652
field width  783
hexadecimal numeric representation  785
line-by-line  677
numeric base  785
octal numeric representation  785
of addresses  756
of bool  755, 781
of char  755
of char*  755
of complex  927, 933, 938
of numeric types  755
of objects in a loop  761
of strings  677, 712, 783
of void*  756
of wchar_t  755
operator >>  754
redirecting  822
skip input  465
standard functions  768
input buffer  839
eback()  839
egetr()  839
gptr()  839
in_avail()  827
iterator  830
pbackfail()  840
sbumpc()  827, 839
setg()  840, 841
sgetc()  827, 839
sgetn()  827, 840
snextc()  827
sputbackc()  827, 839
sungetc()  827, 839
uflow()  839
underflow()  839
xsgetn()  840
input iterator  198, 435
advance()  441
distance()  445
next()  443
step forward  441
input_iterator_tag  466
input stream  772
buffer iterators  828
buffers  826
gcount()  769
get()  768
getline()  769
ignore()  770, 777, 927
iterator  212, 462
member functions  768
peek()  770
putback() 770
read() 769
readsome() 769
sentry 772
unget() 770

insert()
called by inserters 455
for containers 411, 413, 416, 417
for deques 287
for lists 295
for maps and multimaps 340, 341
for multisets 327
for sets 325
for sets and multisets 322, 323, 324
for strings 675, 703, 704, 705
for unordered containers 370, 371, 372, 382
for unordered sets 375
for unordered sets and multisets 377
for vectors 277

insert_after()
for forward lists 306, 312, 423, 424
INSERT_ELEMENTS() 517
inserter 210, 212, 454, 455, 458
for streams 745, 753
user-defined 471

io_errc 45
io_error 46
<iostream> 774, 882, 890
ios 750
see stream
<iomanip> 44, 767
ios_base 748
see stream
ios_base::failure 41, 44, 45, 763
code() 48
error codes 47
<iostream> 752
iostream 751
see stream, input stream, output stream and concurrency 56
error category 50
<iostream> 752, 844
iostream_category() 50
<iostream.h> 41
iostream_withassign 751
iota()
algorithm 571
ipfx() 772
irreflexive 315
is() for ctype facets 891
is_abstract trait 127
is_a_directory 46
isalnum() 896
isalpha() 896
is_arithmetic trait 126
is_array trait 126
is_assignable trait 128
is_base_of trait 128
isblank() 896
is_bounded for numeric limits 117
is_class trait 126
iscntrl() 896
is_compound trait 126
is_const trait 126
is_constructible trait 128
is_convertible trait 128
is_destructible trait 127
isdigit() 896
is_empty trait 127
is_enum trait 126
is_exact for numeric limits 117
is_floating_point trait 126
is_function trait 126
is_fundamental trait 126
isfx() 772
isgraph() 896
is_heap() algorithm 554
is_heap_until() algorithm 554
is_iec559 for numeric limits 117
is_integer for numeric limits 117
is_integral trait 126
is_literal_type trait 126
is_lock_free() for atomsics 1016
islower() 896
is_lvalue_reference trait 126
is_member_function_pointer trait 126
is_member_object_pointer trait 126
is_member_pointer trait 126
is_modulo
for numeric limits 117
is_move_assignable trait 127
is_move_constructible trait 128
is_nothrow_assignable 128
is_nothrow_constructible 128
is_nothrow_copy_assignable 127
is_nothrow_copy_constructible 127
is_nothrow_default_constructible 127
is_nothrow_destructible 127
is_nothrow_move_assignable 127
is_nothrow_move_constructible 127
is_object trait 126
ISO-Latin-1 851
is_open() for streams 798
is_partitioned() algorithm 552
is_permutation() algorithm 544
is_pod trait 126
is_pointer trait 126
is_polymorphic trait 127
isprint() 896
ispunct() 896
is_reference trait 126
is_rvalue_reference trait 126
is_same trait 126
is_scalar trait 126
is_signed for numeric limits 117
is_signed trait 126
is_sorted() algorithm 550
is_sorted_until() algorithm 550
isspace() 688, 896
is_specialized for numeric limits 117
is_standard_layout trait 126
is_steady
  for clocks 149
istream 744, 751
    see stream, input stream
<istream> 752
istreambuf_iterator 732, 828, 830
  equal() 830
istream iterator 212, 462
  end-of-stream 213, 462
  skip input 465
istream_withassign 751
istringstream 802
    see string stream
      move semantisc 806
is_trivial trait 126
is_triviallyAssignable 128
is_trivially_constructible 128
is_triviallyCopyable trait 126
is_triviallyCopyAssign 127
is_triviallyCopyConstructible 127
is_triviallyDefaultConstructible 127
is_triviallyDestructible 127
is_triviallyMoveAssign 127
is_triviallyMoveConstructible 127
istrstream 807
is_union trait 126
is_unsigned trait 126
isupper() 896
is_void trait 126
is_volatile trait 126
isxdigit() 896
iterator 166, 188, 199, 433
  * 188, 191, 435
++, -- 188, 191, 435, 437
+, - 438
+=, -= 438
+= versus advance() 442
-> 191, 435
= 188
==, != 188, 435, 436
<, <=, >, >= 199, 438
[] 438
adapters 210, 448
advance() 442, 586
arithemtic 438
auxiliary functions 441
back_inserter 212, 455
back_inserters 455
bidirectional 198, 437
categories 198, 433, 469
cue order 205
class iterator 471
class into reverse iterator 449
distance() 445
end-of-stream 213, 462
for arrays 266
for containes 190
for forward lists 304
for lists 293
for maps 337
for match results 720
for multismaps 337
for multisets 321, 327
for regular expressions 726
for sets 321
for stream buffers 828
for streams 212, 460, 465
for strings 684
for unordered contained 368
for vectors 275
forward 198, 436
front_inserter 212, 455
front_inserters 457
general inserters 458
input 198, 435
inserter 210, 212, 455
inserters 458
iter_swap() 446
move 216, 466
mutable 435
next() 307, 443
output 198, 433
past-the-end 189
prev() 444
random access 438
random-access 198
ranges 189
raw_storage_iterator 1029
reverse 214, 448
iterator (continued)
  step forward 441
  swapping values 446
  tag 466
  traits 466
  user-defined 471
iterator 471
  for containers 260, 398
  for strings 694
<iterator> 257, 386, 433, 828
iterator adapter 210, 448
  for streams 212, 460
inserter 210, 454
move 216, 466
reverse 214
user-defined 471
iterator_category 467
iterator_traits 467
iter_swap() 446
iword()
  for streams 815

J

join()
  for threads 964, 979
joinable()
  for threads 979

K

key_comp()
  for associative containers 427
  for maps and multimaps 335
  for sets and multisets 318
key_compare
  for associative containers 399
key_eq()
  for unordered containers 362, 427
key_equal
  for unordered containers 399
key_type
  for containers 398
kilo ratio unit 142
knuth_b 916
Koenig lookup 677, 812

L

labs()
  global function 942
lambda 28, 229, 499
  and algorithms 206, 229
  and maps 345
  and mutable 30, 501
  as equivalence criterion 379
  as hash function 379, 504
  as return value 31
  as sorting criterion 231, 232, 504
  capture 28, 29
  state 500
  type 31
  versus binders 499
  versus function object 498
  versus function objects 483, 500, 504
lambda as equivalence criterion
  for unordered containers 379
lambda as hash function
  for unordered containers 379
LANG 857
language features 13, 33
Latin-1 851
launch policy 951, 974
  async 951
  deferred 951
lazy evaluation 951
lazy initialization 1000
ldexp()
  global function 941
ldiv()
  global function 942
left manipulator 346, 783
left stream flag 782
length()
  for char_traits 854
  for codecvt facets 898
  for match results 720, 723
  for strings 670, 696
length_error 41, 43
less 572, 576
less<>() 241, 486
less_equal<>() 486
lexicographical_compare() 548
lexicographical comparison 548
<limits> 116
<limits.h> 115, 116
linear complexity 10
linear_congruential_engine 915
line-by-line input 677
list 173, 290, 298
see container
  = 293
  ==, != 292
  <, <=, >, >= 292
assign() 293
back() 293
begin() 294
cbegin() 294
cend() 294
clear() 295
clear() 295
destructor 291, 292
crbegin() 294
crend() 294
destructor 291, 292
element access 292
emplace() 295
emplace_back() 295
emplace_front() 295
empty() 292
end() 294
erase() 295
exception handling 296
front() 293
header file 290
insert() 295
iterators 293
max_size() 292
merge() 297, 298, 423
pop_back() 295
pop_front() 295
push_back() 295, 298
push_front() 295, 298
rbegin() 294
remove() 294, 420
remove_if() 294, 420
rend() 294
resize() 295
reverse() 297, 423
size() 292
sort() 297, 298, 422
special member functions 420
splice() 297, 298, 421, 422
swap() 293
unique() 297, 298, 421
versus forward_list 300
<list> 290
literals of type string 23, 655
little_endian 900
llabs() 942
lldiv() 942
load() 1012, 1016
load_factor() 362, 380, 427
locale 688, 857
() 868
==, != 868
as sorting criterion 868
"C" 861
classic() 860, 868
collate category 904
collate category 891
default constructor 864
destructor 866
defacets 864, 869
global() 864, 868
has_facet() 867
id 869
imbue() a stream 860
messages category 905
monetary category 874
name() 863, 868
numeric category 870
string collation 904
string comparisons 868
time category 884
use_facet() 867
<locale> 866
local_iterator
  for unordered containers  399
localtime()  158
lock()
  for shared_ptrs  97
  for unique_locks  1000
  for mutexes  989, 998
  for weak_ptrs  88
lock-free programming  988
lock_guard  989, 991, 999
locks  989, 991
  for condition variables  1004
lock_guard  989
  multiple  994, 997
  recursive  992
  unique_lock  996
  with timer  994
log()
  for complex  935, 940
  global function  941
log10()
  for complex  935, 940
  global function  941
logarithmic complexity  10
logical_and<>()  486
logical_not<>()  486
logical_or<>()  486
logic_error  41, 43
lognormal_distribution  917, 923
long
  input  755
  numeric limits  116
loop
  condition  760
  for reading objects  761
  range-based for  17
lower
  for ctype_base  894
lower_bound()
  algorithm  611
  for containers  405
  for maps and multimaps  335
  for sets and multisets  319
lowercase string characters  684
lowest()

for numeric limits  117
lt()
  for char_traits  689, 854

M

main()  37
make_error_code()  53
make_heap()
  algorithm  514, 604, 606, 644
make_pair()
  for pairs  61, 65, 341, 372
make_ready_at_thread_exit()
  for packaged tasks  977
make_shared()
  for shared_ptrs  78, 93
make_signed trait  130
make_tuple()
  for tuples  69, 70, 71
make_unsigned trait  130
manipulator  746, 774
boolalpha  781
dec  785
defaultfloat  788
del  746, 774, 776, 846
delns  746, 774, 808
dfixed  788
flush  746, 774, 846
get_money()  882
get_time()  890
hex  785
hexfloat  788
implementation  776
internal  783
left  346, 783
noboolalpha  781
no_showbase  786
no_showpoint  788
no_showpos  784
no_skipws  789
nounitbuf  789
nouppercase  784
oct  785
overview  774
put_money()  882
put_time() 890
resetiosflags() 780
right 783
scientific 788
setfill() 783
setiosflags() 780
setprecision() 788
setw() 346, 755, 783, 793
showbase 786
showpoint 788
showpos 784
skipws 789, 847
unitbuf 789
uppercase 784
user-defined 777
with arguments 778
ws 746, 774
map 331, 346, 350, 351
see container
= 336
==, != 335
<, <=, >, >= 335
[] 186, 343, 408
and for_each() 345
and lambdas 345
as associative array 185, 343
at() 186, 343, 408
begin() 337
cbegin() 337
cend() 337
clear() 340
constructor 333
count() 335
crbegin() 337
crend() 337
destructor 333
element access with bind() 494
emplace() 340
emplace_hint() 340
empty() 335
end() 337
equal_range() 335
erase() 340
exception handling 345
find() 335
header file 331
insert() 340, 341
iterators 337
key_comp() 335, 427
key_compare 399
lower_bound() 335
max_size() 335
modifying access 221
piecewise construction 342
rbegin() 337
removing elements 342
rend() 337
replace key 339
size() 335
sorting criterion 232, 331, 334, 351
swap() 336
upper_bound() 335
value_comp() 335, 427
value_compare 399
value_type 331, 345

<map> 331
mapped_type
  for containers 399
mask
  for ctype_base 893
match_any regex constant 733
match_continuous regex constant 733
match_not_bol regex constant 733
match_not_bow regex constant 733
match_not_eol regex constant 733
match_not_eow regex constant 733
match_not_null regex constant 733
match_prev_avail regex constant 733
match regular expressions 717
match_results
  for regular expressions 720
max() 134
  for distributions 918
  for durations 147
  for numeric limits 117, 777
  for timepoints 155
max_align_t 161
max_bucket_count()
   for unordered containers 362, 429
max_digits10
   for numeric limits 117
max_element()
   algorithm 200, 525
max_exponent
   for numeric limits 117
max_exponent10
   for numeric limits 117
maximum
   of elements 525
   of numeric types 115, 120
   of two values 134
timepoint 152
max_length()
   for codecvt facets 898
max_load_factor()
   for unordered containers 362, 380, 383, 427, 429
max_size()
   for arrays 264
   for containers 255, 258, 403
   for deques 286
   for forward lists 303
   for lists 292
   for maps and multimaps 335
   for sets and multisets 318
   for strings 670, 696
   for unordered containers 367
   for vectors 273
mdy date order 889
mega ratio unit 142
member
   as sorting criterion 228
member function
   adapter 491
   as template 34
member template 34
memchr() 163, 855
memcmp() 163, 855
memcpy() 163, 841, 855
mem_fn() 487, 494
mem_fun() 497
mem_fun1_ref_t 498
mem_fun1_t 498
mem_fun_ref() 243, 497
mem_fun_ref_t 498
mem_fun_t 498
memmove() 163, 855
<memory> 44, 76, 78
memory leak 98
memory order 1016, 1020
memory_order_acquire 1021
memory_order_Relaxed 1021
memory_order_release 1021
memory_order_seq_cst 1016, 1020
memset() 163, 855
merge()
   algorithm 614
   for forward lists 310, 312
   for lists 297, 298, 423
mersenne_twister_engine 915
message()
   for error category 49
   for exceptions 49
message_base 905
catalog 905
messages facet 905
close() 905
get() 905
open() 905
messages locale category 905
message_size 46
micro_ratio_unit 142
microseconds
   for durations 145
milli_ratio_unit 142
milliseconds
   for durations 145
min() 134
   for distributions 918
   for durations 147
   for numeric limits 117
   for timepoints 155
min_element()
   algorithm 200, 525
min_exponent
   for numeric limits 117
min_exponent
for numeric limits 117
minimum
of elements 525
of numeric types 115, 120
of two values 134
timepoint 152
minmax() 134
minmax_element()
algorithm 526
minstd_rand 916
minstd_rand0 916
minus<() 486
minutes
for durations 145
mirror elements 566
mismatch()
algorithm 546
mktime() 158
modf()
global function 941
modifying algorithms 509, 557
modifying elements 509, 557
modulus<>() 486, 573
monetary locale category 874
money_base 877
none 878
part 877
pattern 877
sign 878
space 878
symbol 878
value 878
money_put facet 879
put() 879, 880
monotonic_clock 149
move() 19
algorithm 561
for char_traits 854
move assignment 21
move_backward()
algorithm 561
move constructor 21
for containers 401
move iterator 216, 466
move semantics 19
and return values 22
for arrays 263
for containers 257, 258
for file streams 795
for strings 676
for string streams 806
moving elements 561
mt19937 916
mt19937_64 916
multibyte format 850
multimap 179, 331, 348
see container
= 336
==, != 335
<, <=, >, >= 335
begin() 337
cbegin() 337
cend() 337
clear() 340
constructor 333
count() 335
crbegin() 337
crend() 337
destructor 333
element access with bind() 494
emplace() 340
emplace_hint() 340
empty() 335
end() 337
equal_range() 335
erase() 340
exception handling 345
multimap (continued)
  find() 335
  header file 331
  insert() 340, 341
  iterators 337
  key_comp() 335, 427
  key_compare 399
  lower_bound() 335
  max_size() 335
  modifying access 221
  order of duplicates 180
  order of equivalent elements 343
  piecewise construction 342
  rbegin() 337
  removing elements 342
  rend() 337
  replace key 339
  size() 335
  sorting criterion 232, 331, 334, 351
  stable order 180
  swap() 336
  upper_bound() 335
  value_comp() 335, 427
  value_compare 399
  value_type 331
multiplies<() 241, 486, 563
multiset 177, 314, 327
  see container
    = 321
    ==, != 318
    <, <=, >, >= 318
    begin() 321
    cbegin() 321
    cend() 321
    clear() 322
    constructor 316, 327
    count() 319
    crbegin() 321
    crend() 321
    destructor 316
    emplace() 322
    emplace_hint() 322
empty() 318
end() 321
equal_range() 319
erase() 322, 327
exception handling 325
find() 319, 327
header file 314
insert() 322, 323, 324, 327
iterator 327
iterators 321
key_comp() 318, 427
key_compare 399
lower_bound() 319
max_size() 318
modifying access 221
order of duplicates 180
order of equivalent elements 325
rbegin() 321
rend() 321
size() 318
sorting criterion 232, 314, 316, 328
stable order 180
swap() 321
upper_bound() 319
value_comp() 318, 427
value_compare 399
value_type 331
multithreading 55, 945
  see concurrency
mutable
  for lambdas 30
mutable iterator 435
mutating algorithms 511, 583
mutex 989, 991, 998
  for condition variables 1004
  lock_guard 999
  multiple locks 994, 997
  try_lock() 994
  unique_lock 1000
  versus atomics 1012
mutex()
  for unique_locks 1000
<mutex> 1001, 1004
Index N 1073

N

name()  
for error category  49  
for locales  863, 868  
namespase  
ADL  677, 812  
Koenig lookup  677, 812  
placeholders  487  
posix  39, 802  
std  39  
std::chrono  144  
std::placeholders  243  
std::regex_constants  732  
std::rel_ops  138  
std::this_thread  981  
tr1  39  
using declaration  40  
using directive  40  
nano ratio unit  142  
nanoseconds  
for durations  145  
narrow()  
for ctype facets  891  
for streams  790  
narrowing initialization  15  
narrow stream  744  
native_handle()  
for condition variables  1009  
for mutexes  999  
for threads  979  
negate<>()  486  
negative_binomial_distribution  917, 922  
negative_sign()  
for moneypunct facets  874  
neg_format()  
for moneypunct facets  874  
nested class  
as template  37  
network_down  46  
network_reset  46  
network_unreachable  46  
new  43  
and shared_ptrs  77  
and unique_ptrs  99  
<new>  43  
newline  
internationalized  857  
newsgroups  1031  
next()  307, 443  
next_permutation()  
algorithm  587  
n-log-n complexity  10  
noboolalpha manipulator  781  
n_buffer_space  46  
n_child_process  46  
nocnv  
for codecvt_base  899  
nocreate stream flag  797  
nexcept  24, 137  
n_link  46  
n_lock_available  46  
n_message  46  
n_message_available  46  
none  
monetary pattern  878  
none_of()  
algorithm  555  
nonmodifying algorithms  507, 524  
o-op  204  
o_order date order  889  
o_protocol_option  46  
noreplace stream flag  797  
norm()  
for complex  931, 938  
normal_distribution  917, 923  
noshowbase manipulator  786  
nshawpoint manipulator  788  
noshawpos manipulator  784  
noskipws manipulator  789  
n_space_on_device  46  
n_state  47  
n_stream_resources  46  
nosubs regex constant  733  
n_such_device  46  
n_such_device_or_address  46  
n_such_file_or_directory  46  
n_such_process  46  
not1  529
not1() 487, 494, 497, 498
not2() 487, 494, 497
not_a_directory 46
not_a_socket 46
not_a_stream 47
not_connected 47
not_enough_memory 47
not_eof()
    for char_traits 854
not_equal_to<>() 486
notify_all()
    for condition variables 1004, 1009
notify_all_at_thread_exit()
    for condition variables 1009
notify_one()
    for condition variables 1004, 1009
not_supported 47
nounitbuf manipulator 789
nouppercase manipulator 784
now()
    for clocks 149
    for system_clock 152
npos
    for strings 658, 680, 694
NRVO 23
nth_element()
    algorithm 602
    versus partition() 514
NULL 161
    and strings 668
    versus nullptr 14
nullptr 14
nullptr_t 14, 33, 161
num
    for ratios 140
number of elements 524
numeric
    algorithms 515, 623
    base 785
    conversions for strings 713
    formatting 784, 870
    global functions 941
    input 755
    libraries 907
    limits 115
    <numeric> 505, 623
numeric locale category 870
numeric conversions
    for strings 681
numeric_limits 115, 652, 777
num_get facet 873
    get() 873
numpunct facet 870
    decimal_point() 870
    falsename() 870
    grouping() 870
    thousands_sep() 870
    truename() 870
num_put facet 871
    put() 871, 872
O
oct manipulator 785
oct stream flag 785
octal numeric representation 785
offsetof() 161
off_type
    for char_traits 854
    for streams 800
ofstream 791, 793
    see file stream and output stream
ok
    for codecvt_base 899
O(n) 10
once_flag 1000
open()
    for messages facets 905
    for streams 798
Open Closed Principle 385
openmode
    for streams 796
operation_canceled 47
operation_in_progress 47
operation_not_permitted 47, 999
operation_not_supported 47
operation_would_block 47
operator
    <<, >> 753
    dynamic_cast 42
for I/O 753
typeid 42
opfx() 772
optimize regex constant 733
order
  of duplicates in associative containers 180
  of duplicates in unordered containers 183
stable 180, 183
unordered 180
ordered collection 167
osfx() 772
ostream 744, 751
  see stream, output stream
<ostream> 752
ostreambuf_iterator 828, 829
  failed() 829
ostream iterator 212, 460
ostream_withassign 751
ostreamstream 802
  see string stream
ostreamstream 807
out()
  for codecvt facets 898
out stream flag 796
out_of_range 41, 43
output 743
  see stream, output stream
  adjustment 781
  and concurrency 56
  binary numeric representation 652
  defining floating-point notation 787
  field width 781
  fill character 781
  for shared_ptrs 94
  for pairs 62
  for tuples 74
  hexadecimal numeric representation 785
  numeric base 785
  octal numeric representation 785
  of addresses 756
  of bool 755, 781
  of complex 927, 933, 938
  of numeric values 784
  of strings 677, 712
  of void* 756
operator << 753
positive sign 784
redirecting 822
signs 784
standard functions 771
output buffer 832
epptr() 832
iterator 829
overflow() 832
pbase() 832
pbump() 839
pptr() 832
seekoff() 839
seekpos() 839
setp() 837
sputc() 826, 832
sputn() 826, 832
sync() 839
xsputn() 832
output iterator 198, 433
  iterator traits 468
output_iterator_tag 466
output stream 772
  buffer iterators 828
  buffers 826
  flush() 771
  iterator 212, 460
  member functions 771
  put() 771
  sentry 772
  write() 771
overflow() 833
  for output buffers 832
overflow_error 41, 43
overloading
  of I/O operators 810
  rvalue and lvalue references 22
  with functions as parameter 776
owner_before() 47
for shared_ptrs 94, 97
owner_dead 47
owns_lock()
  for unique_locks 1000
P

packaged_task 972, 977
    constructor 977
defstructor 977
get_future() 972, 977
make_ready_at_thread_exit() 977
reset() 977
swap() 977
valid() 977

pair 60, 324
    operator<< 62
    = 61
    ==, != 61
    <, <=, >, >= 61
    and tuple 75
    as element of (multi)maps 337
    as element of unordered (multi)maps 369
    assign to tuple 71
    comparisons 67
    constructor 61
defstructor 61
    first 61
    first_type 60
    get() 61
    ignore 67
    initialize tuple 71
    make_pair() 61, 65, 341, 372
    output 62
    piecewise construction 63
    second 61
    second_type 60
    swap() 61
    tie() 67
    tuple_cat() 73
    tuple_element 62
    tuple_size 62

param()
    for distributions 918

part
    for money_base 877
partial
    for codecvt_base 899
partial_sort()
    algorithm 514, 599
    partial_sort_copy()
    algorithm 600
    partial_sum()
    algorithm 627, 630
partition()
    algorithm 592
    versus nth_element() 514
partition_copy()
    algorithm 594
partition_point()
    algorithm 552
past-the-end iterator 189

pattern
    for money_base 877
    pbackfail()
        for input buffers 840
    pbase()
        for output buffers 832
    pbump()
        for output buffers 839
    peek()
        for input streams 770
performance 10
    of shared_ptrs 95
    of containers and algorithms 394
    of smart pointers 114
    of streams 844

period
    for clocks 149
    for durations 147
permission_denied 47
perror() 905
peta ratio unit 142
pico ratio unit 142
piecewise_constant_distribution 917, 924

piecewise construction
    and (multi)maps 342
    and unordered (multi)maps 373
    for pairs 63
piecewise_construct_t 63
piecewise_linear_distribution 917, 924
placeholders 243, 487
plus<>() 486, 629
POD 279

pointer
   and bind() 493
   auto_ptr 113
   I/O 756
   iterator traits 468
   NULL 161
   nullptr 14
   shared_ptr 76
   smart 76
      see smart pointer
   unique_ptr 98
   weak_ptr 84
pointer
   for containers 260, 398
   for iterator_traits 467
   for strings 693
poisson_distribution 917, 922
polar()
   for complex 929, 936
policy
   for unordered containers 359
polling a thread or background task 954
pop()
   for container adapters 648
pop_back()
   for containers 419
   for deques 287
   for lists 295
   for strings 675, 705
   for vectors 277
pop_front()
   for containers 419
   for deques 287
   for forward lists 306
   for lists 295
pop_heap()
   algorithm 605, 606, 644
pos_format()
   for moneypunct facets 874
position()
   for match results 720, 723
positioning
   in files 799
positive_sign()
protocol_error 47
protocol_not_supported 47
proxy
    for vector< bool > 282
ptrdiff_t 161
ptr_fun() 497
pubimbue()
    for stream buffers 827
pubseekoff()
    for stream buffers 827
pubseekpos()
    for stream buffers 827
pubsetbuf()
    for stream buffers 827
punct
    for ctype_base 894
pure abstraction 199
push()
    for container adapters 647
push_back()
    called by inserters 455
    for containers 415
    for deques 287
    for lists 295, 298
    for strings 674, 703
    for vectors 277
push_front()
    called by inserters 455
    for containers 414
    for deques 287
    for forward lists 306, 312
    for lists 295, 298
push_heap()
    algorithm 605, 606, 644
put()
    for money_put facets 879, 880
    for num_put facets 871, 872
    for output streams 771, 772, 793
    for time_put facets 884
putback()
    for input streams 770
put buffer 832
    iterator 829
putchar() 833
put_money() 882
put_time() 757, 890
pword()
    for streams 815

Q
quadratic complexity 10
queue 638
    back() 648
    header file 638
    user-defined version 641
    with concurrent access 1006
<queue> 638, 641
quick_exit() 162
quiet_NaN()
    for numeric limits 117

R
race condition 982
radix
    for numeric limits 117
rand() 570
    global function 942
<random> 908, 909
random access
    to container elements 169
    to files 799
random-access iterator 198, 438
    advance() 441
    distance() 445
    next() 443
    prev() 443
random_access_iterator_tag 466
random-number generator 908
    seed 910
    serialization interface 915, 916
    state 912
    values versus numbers 912
random numbers 907
    engines 912
    generator 908
    seed 910
    versus random values 912
versus random values 912
random_shuffle() algorithm 589
range 203
  change order of elements 583
  comparing 542
  copy 557
  copy and modify elements 563
  counting elements 524
  empty 189
  for iterators 189
  half-open 203
  in algorithms 203
  maximum 525
  minimum 525
  modifying 557, 568
  move 561
  multiple 207
  mutating 583
  notation 203
  numeric processing 623
  of values 115
  removing duplicates 578
  removing elements 218, 575
  replacing elements 571
  searching elements 507, 528
  sorting 596
  swapping elements 566
  transform elements 563
  valid 203, 205
range-based for loop 17, 193
range_error 41, 43
rank trait 131
ranlux24 916
ranlux24_base 916
ranlux48 916
ranlux48_base 916
ratio 140, 144
  den 140
  num 140
  ratio_add 141
  ratio_divide 141
  ratio_equal 141
  ratio_greater 141
  ratio_greater_equal 141
  ratio_less 141
  ratio_less_equal 141
  ratio_multiply 141
  ratio_not_equal 141
  ratio_subtract 141
  type 140
  units 142
  <ratio> 140
  ratio_add 141
  ratio_divide 141
  ratio_equal 141
  ratio_greater 141
  ratio_greater_equal 141
  ratio_less 141
  ratio_less_equal 141
  ratio_multiply 141
  ratio_not_equal 141
  ratio_subtract 141
  raw_storage_iterator 1029
raw string 719, 732
  literals 23
rbegin() 214, 448, 452
  for arrays 266
  for containers 411
  for deques 286
  for lists 294
  for maps and multimaps 337
  for sets and multisets 321
  for strings 714
  for unordered containers 369
  for vectors 276
rdbuf() 800, 820, 822, 846
rdstate() 759
reachable 203
read() 769
  for input streams 769
  global function 841
reading
  see input
read_only_file_system 47
readsome() 769
ready future status 954
real()
  for complex 931,937
reallocation
  for strings 670
  for vectors 270
recursive locks 992
recursive_mutex 993,998
recursive_timed_mutex 994,998
red-black tree 315
redirecting
  streams 822
ref() 132
  and bind() 491
  and make_pair() 66
  and make_tuple() 70
reference
  for rvalues 19
reference
  for container adapters 645
  for containers 260,397
  for iterator_traits 467
  for strings 693
  for vector<bool> 282
reference counting
  for strings 692
references
  cyclic 84
reference semantics
  for containers 245,388
reference_wrapper 132,391
  and make_pair() 66
  and make_tuple() 70
regex 717
  and algorithms 727
  awk grammar 739
  basic grammar 739
  basic_regex 719
  capture group 719
  case-insensitive 732
  constants 732
  ECMAScript grammar 738,739
  egrep grammar 732,739
  error handling and exceptions 735
  extended grammar 739
  flags 732
  grammars 738,739
  grep grammar 739
  icase regex constant 732
  initialization regex 719,741
  iterator 726
  match interface 717
  match_results 720
  regex_match() 717,740
  regex_replace() 730,740
  regex_search() 717,740
  replace interface 730
  search interface 717
  subexpressions 720
  sub_match 720
  token iterator 727
  wregex 719
regex_constants 732
regex_error 735
regex_iterator 726
regex_match()
  for regular expressions 717,740
regex_replace()
  for regular expressions 730,740
regex_search()
  for regular expressions 717,740
regex_token_iterator 727
register_callback()
  for streams 817
regular expression 717
  see regex
rehash()
  for unordered containers 362,428
relative to absolute values 516,627
release()
  for unique_locks 1000
  for unique_ptrs 100,111
rel_ops 138
remove()
  algorithm 575
  for forward lists 305,306
  for lists 294,420
remove_all_extents trait 131
remove_const trait 130
remove_copy()
  algorithm 577
remove_copy_if()  
    algorithm  577
remove_cv trait   130
remove_extent trait  131
remove_if()  
    algorithm  483, 575
    for forward lists  305, 306
    for lists  294, 420
remove_pointer trait  130
remove_reference trait  130
removing algorithms  511, 575
removing duplicates  578
removing elements  218, 511
rend()  214, 448, 452
    for arrays  266
    for containers  411
    for deques  286
    for lists  294
    for maps and multimaps  337
    for sets and multisets  321
    for strings  714
    for unordered containers  369
    for vectors  276
reordering of statements  986
rep  
    for clocks  149
    for durations  147
replace()  
    algorithm  571
    for strings  675, 687, 706, 707, 708
replace and copy elements  573
replace_copy()  
    algorithm  573
replace_copy_if()  
    algorithm  573
replace_if()  
    algorithm  571
replace regular expression  730
representation  
    binary  652
    decimal  785
    hexadecimal  785
    octal  785
requirements  
    for container elements  244
    for sorting criterion  314
    of containers  254
reserve()  
    for containers  428
    for strings  670, 672, 697
    for unordered containers  362
    for vectors  271, 273
reset()  
    for shared_ptrs  78, 93, 97
    for packaged tasks  977
    for unique_ptrs  111
resetiosflags() manipulator  780
resize()  
    for containers  176, 420
    for deques  287
    for forward lists  306
    for lists  295
    for strings  676, 706
    for vectors  277
resource_deadlock_would_occur  47, 999
resource leak  98
resource_unavailable_try_again  47
result  
    for codecvt_base  898
result_of trait  131
result_out_of_range  47
result_type  
    for distributions  918
rethrow_exception()  52, 971
return_temporary_buffer()  1029
return type  
    new syntax  29, 32
    of main()  37
return value  
    move semantics  22
    optimization (RVO)  23
reverse()  
    algorithm  200, 583
    for forward lists  310
    for lists  297, 423
    for strings  687
reverse_copy()  
    algorithm  583
reverse iterator 214, 448
    base() 452
convert into iterator 452
reverse_iterator
    for containers 398
    for strings 694
rfind()
    for strings 708, 709
right manipulator 783
right stream flag 782
rotate()
    algorithm 584
rotate_copy()
    algorithm 585
round_error()
    for numeric limits 117
round_indeterminate 120
round_style
    for numeric limits 117, 119
round_to_nearest 120
round_toward_infinity 120
round_toward_neg_infinity 120
round_toward_zero 120
runtime_error 41, 43
rvalue reference 19
    overloading 22
RVO 23

S

safe STL 247
sampling distribution 917
sbumpc()
    for input buffers 827, 839
scan_is()
    for ctype facets 891
scan_not()
    for ctype facets 891
scientific manipulator 788
scientific stream flag 787
scoped enumeration 32
search
    case-insensitive 732
search()
    algorithm 534, 684
searching algorithms 507, 528
search_n()
    algorithm 531
search_n_if()
    algorithm 533
search regular expressions 717
second
    for pairs 61
seconds
    for durations 145
second_type
    for pairs 60
seed 910
seed()
    for random-value engines 916
seekdir 800
seekg()
    for streams 799, 800, 825
seekoff()
    for output buffers 839
seekp() 809
    for streams 799, 825
seekpos()
    for output buffers 839
self-defined
    see user-defined
sentry 772
sequence
    see collection, container, range
sequence container 167, 169
sequential consistent memory order 1016, 1020
serialization interface
    for distributions 918
    for random-value engines 915, 916
set 314, 325
    see container
= 321
==, != 318
<, <=, >, >= 318
begin() 321
C++98/C++03 example 193
cbegin() 321
cend() 321
clear() 322
constructor 316, 325
count() 319
crbegin() 321
crend() 321
destructor 316
emplace() 322
emplace_hint() 322
empty() 318
end() 321
equal_range() 319
erase() 322, 325
exception handling 325
find() 319, 325
header file 314
insert() 322, 323, 324, 325
insert elements 323
iterators 321, 325
key_comp() 318, 427
key_compare 399
lower_bound() 319
max_size() 318
modifying access 221
rbegin() 321
rend() 321
size() 318
sorting criterion 232, 314, 316, 328
swap() 321
upper_bound() 319
user-defined sorting criterion 476
value_comp() 318, 427
value_compare 399
<set> 314
set_difference()
algorithm 618
set_exception()
for promises 971, 977
set_exception_at_thread_exit()
for promises 971, 977
setf()
for streams 779
setfill() manipulator 783
setg()
for input buffers 840, 841
set_intersection()
algorithm 617
setiosflags() manipulator 780
setlocale() 859
setp()
for output buffers 837
setprecision() manipulator 788
setstate()
for streams 759
set_symmetric_difference()
algorithm 619
set_union()
algorithm 616
set_value()
for promises 971, 977
set_value_at_thread_exit()
for promises 971, 977
setw() manipulator 346, 755, 783, 793
sgetc()
for input buffers 827, 839
sgetn()
for input buffers 827, 840
share()
for futures 975
shared_future 960, 976
see future
get() 976
shared pointer
see shared_ptr
shared_ptr 76, 92, 388
* 79, 94
-> 94
<< 94
= 93, 97
==, != 94
<, <=, >, >= 94
aliasing constructor 95
allocate_shared() 93
and arrays 80
and bind() 493
atomic_exchange() 97
atomic_is_lock_free() 97
atomic_load() 97
atomic_store() 97
bad_weak_ptr 89
bool() 94
comparisons 92
shared_ptr (continued)
  const_pointer_cast() 94
  constructor 78, 93, 97
  cyclic references 84
  deleter 80, 82
  destructor 93, 97
  dynamic_pointer_cast() 94
  enable_shared_from_this 90
  expired() 97
  get() 94
  get_deleter() 94
  header file 78
  lock() 97
  make_shared() 78, 93
  output 94
  owner_before() 94, 97
  performance 95, 114
  release() 94
  reset() 78, 93, 97
  static_pointer_cast() 94
  swap() 93, 97
  thread-safe interface 96
  unique() 94
  use_count() 94, 97

shared state 969, 973

short
  numeric limits 116

showbase manipulator 786
showbase stream flag 786
showpoint manipulator 788
showpoint stream flag 787
showpos manipulator 784
showpos stream flag 784

shrink_to_fit()
  for containers 428
  for deques 286
  for strings 670, 697
  for vectors 271, 273

shuffle()
  algorithm 589, 908
  and temporaries 911

shuffle_order_engine 916

sign
  monetary pattern 878

signaling_NaN()
weak_ptr 84
snextc() 827
for input buffers
sort() 827
algorithm 200, 228, 512, 596
for forward lists 310, 312
for lists 297, 298, 422
versus stable_sort() 597
sorted collection 167
sorted-range algorithms 608
sort_heap() 608
algorithm 514, 605, 606
sorting algorithms 511, 596
sorting criterion
as constructor parameter 316, 334
as template parameter 316, 334
at runtime 328, 351
for associative containers 232
for maps 351
for maps and multimaps 331, 334
for multimaps 351
for multisets 328
for sets 328
for sets and multisets 314, 316
for strings 351
function 228
function object 476
lambda 231
requirements 314
user-defined 228, 476
with locale 868
space
for ctype_base 894
monetary pattern 878
special characters
internationalized 857
speculative execution 954
splice() 422
for lists 297, 298, 421, 422
splice_after() 422
for forward lists 309, 310, 425, 426
spurious failure of try_lock() 994
spurious wakeups of condition variables 1004
sputbackc() 827, 839
sputc() 826, 832
for output buffers
sputn() 826, 832
for output buffers
sqrt() 935, 939
for complex 935
global function 941
srand() 942
global function
sregex_iterator 726
sregex_token_iterator 727
<sstream> 803
stable 180, 183
stable_partition() 514, 596
algorithm 592
versus sort() 597
stack 632
emplace() 634
header file 632
user-defined version 635
<stack> 632
Stack.hpp 635
standard error channel 745
redirecting 822
standard input channel 745
redirecting 822
standard operators
for I/O 753
standard output channel 745
redirecting 822
standard template library 165
see STL
state
of function objects 478
of streams 758
state_not_recoverable 47
state_type
for char_traits 854
static_pointer_cast() 94
for shared_ptrs
std namespace 39
<stddef.h> 161
stderr 745
<stdexcept> 43, 44
stdin 745
<stdlib.h> 162
stdout 745
steady_clock 149
duration 149
is_steady 149
now() 149
period 149
rep 149
time_point 149
STL 165
algorithms 166, 199, 217, 505
commit-or-rollback 248
concurrency 56
container adapters 631
containers 165, 167, 253, 397
element requirements 244
error handling 246
exceptions handling 248
extending 250
function objects 233, 475
functors 233, 475
inheritance 251
iterator adapters 210, 448
iterators 166, 188, 433
manipulating algorithms 217
predicates 226, 506
ranges 203
safe STL 247
transaction safety 248
stod() 682, 713
stof() 682, 713
stoi() 682, 713
stol() 682, 713
stold() 682, 713
stoll() 682, 713
store() 1012, 1016
stoul() 682, 713
stoull() 682, 713
str() 808
for match results 720, 723
for regex_iterator 726
for regex_token_iterator 728
for string streams 804, 934
stream 743, 746, 772
<< conventions 818
>> conventions 818
adjustfield 782
adjustment 781
and concurrency 56, 752
app flag 796
ate flag 796
bad() 759
badbit 758
basefield 785
basic_filebuf 824
beg position 800
binary flag 796
boolalpha flag 781
boolalpha manipulator 781
buffer iterators 828
buffers 826, 832
callback 817
cracter traits 853
class hierarchy 748
clear() 759, 797, 800
close() 798
connecting 819
copyfmt() 779, 780, 811, 817, 822
copyfmt_event 817
cur position 800
dec flag 785
dec manipulator 785
defaultfloat manipulator 788
defining floating-point notation 787
defining position 800
endif manipulator 746, 774, 776, 846
default of-file 750
ends manipulator 746, 774, 808
EOF 750
eof() 759
eofbit 758
erase_event 817
event 817
event_callback 817
exceptions() 762
fail() 759
failbit 758
failure 763
field width 781
file access 791
filebuf 824
fill() 782
fill character 781
fixed flag 787
fixed manipulator 788
flags() 779, 780
floatfield 787
flush manipulator 746, 774, 846
for char* 807
for file descriptors 801, 835
format flags 779
formatting 779
formatting of bool 755, 781
fpos 799
freeze() 808
getloc() 790
good() 759
goodbit 758
hardfail 758
header files 752
hex flag 785
hex manipulator 785
hexadecimal 785
hexfloat manipulator 788
imbue() 790, 860
imbue_event 817
in flag 796
input buffers 839
input functions 768
internal flag 782
internal manipulator 783
internationalization 790
is_open() 798
iterators 460
iword() 815
left flag 782
left manipulator 346, 783
manipulators 746, 774
member functions 767
narrow() 790
noboolalpha manipulator 781
nocreate flag 797
noreplace flag 797
noshownotice manipulator 786
noshownoticepoint manipulator 788
noshownoticepos manipulator 784
noskipws manipulator 789
nounderline manipulator 789
nounderlinepos manipulator 784
numeric bases 785
oct flag 785
oct manipulator 785
octal representation 785
off_type 800
open() 798
openmode 796
operator ! 760, 761
operator bool 760
operator void* 760
out flag 796
output buffers 832
output functions 771
performance 844
positioning 799
pos_type 799
precision() 787
pword() 815
rdbuf() 800, 820, 822, 846
rdstate() 759
read and write 824
read and write position 825
redirecting standard streams 822
register_callback() 817
resetiosflags() manipulator 780
right flag 782
right manipulator 783
scientific flag 787
scientific manipulator 788
seekg() 799, 800, 825
stream (continued)

seekp() 799, 825
sentry 772
setf() 779
setfill() manipulator 783
setiosflags() manipulator 780
setprecision() manipulator 788
setstate() 759
setw() manipulator 346, 755, 783, 793
showbase flag 786
showbase manipulator 786
showpoint flag 787
showpoint manipulator 788
showpos flag 784
showpos manipulator 784
skipws flag 789
skipws manipulator 789, 847
state 758
state and open() 798
str() 808
string access 802
synchronize streams 819
sync_with_stdio() 845
tellg() 799
tellp() 799
testing the state 760
tie() 819
trunc flag 796
unitbuf flag 789, 846
unitbuf manipulator 789
unsetf() 779
uppercase flag 784
uppercase manipulator 784
user-defined buffers 832
widen() 790
width() 782, 811
ws manipulator 746, 774
xalloc() 815
streambuf 750, 832
see stream buffer
<stringbuf> 752
stream buffer 826
see input buffer, output buffer
<<, >> 846
for file descriptors 801, 835
getloc() 827
pubimbue() 827
pubseekoff() 827
pubseekpos() 827
pubsetbuf() 827
user-defined 832
stream iterator 212, 460, 465
end-of-stream 213, 462
skip input 465
streamoff 800
streampos 799
streamsize 767
stream_timeout 47
strtime() 158, 884
strict weak ordering 314
string 655
+ 676, 711
++, -- for iterators 440
+= 674, 702
<< 677, 712
= 673
==, != 697
<, <=, >, >= 697
>> 677, 712
[] 671
allocator_type 715
append() 674, 702, 703
assign() 673, 700, 701
at() 671, 699
automatic type conversions 667
back() 671, 699
begin() 684, 714
capacity() 427, 670, 696
cbegin() 684, 714
cend() 684, 714
char* stream 807
character traits 689, 853
classes 664
clear() 674, 705
compare() 698
compare case-insensitive 351
comparisons 672
concatenation 676
const_iterator 694
const_pointer 694
const_reference 693
const_reverse_iterator 694
constructor 687, 694, 695
converting index into iterator 687
converting into char* 669, 700
converting iterator into index 687
copy() 669, 700
crbegin() 714
crend() 714
c_str() 669, 700
data() 669, 700
destructor 696
difference_type 693
empty() 670, 696
encoding prefix 24
end() 684, 714
erase() 675, 687, 705
find() 708, 709
find_first_not_of() 709, 710
find_first_of() 709, 710
find_last_not_of() 710, 711
find_last_of() 710, 711
front() 671, 699
get_allocator() 715
getline() 677, 713
header file 663
input 677, 712, 755, 783
insert() 675, 703, 704, 705
internationalization 689
iterator 694
iterators 684
length() 670, 696
literals 23, 655
locale dependent collations 904
locale dependent comparisons 673
lowercase characters 684
max_size() 670, 696
move semantics 676
npos 658, 680, 694
NULL 668
numeric conversions 681, 713
output 677, 712
pointer 693
pop_back() 675, 705
push_back() 674, 703
raw string literals 23, 719
rbegin() 714
realloction 670
reference 693
reference counting 692
regular expressions 717
rend() 714
replace() 675, 687, 706, 707, 708
reserve() 670, 672, 697
resize() 676, 706
reverse() 687
reverse_iterator 694
rfind() 708, 709
search functions 678
shrink_to_fit() 670, 697
size() 670, 696
size_type 680, 693
sorting criterion 351
stod() 682, 713
stof() 682, 713
stoi() 682, 713
stol() 682, 713
stold() 682, 713
stoll() 682, 713
stoul() 682, 713
stoull() 682, 713
str() 811
string stream 802
substr() 676, 711
substrings 676
swap() 674, 702
tokenization 727
to_string() 682, 713
to_wstring() 682, 713
traits_type 693
uppercase characters 684
value_type 693
<string> 663, 854
stringbuf 802
<string.h> 163, 855
string stream 802
app and ate 806
class hierarchy 802
move semantics 806
str() 804
stringstream 802
    see string stream
string termination character
    internationalized 857
strlen() 855
strstream 807
<strstream> 807
strstreambuf 807
struct tm 158
student_t_distribution 917, 924
sub_match
    for regular expressions 720
subclass operator
    see []
substr()
    for strings 676, 711
subtract_with_carry_engine 915
suffix()
    for match results 720, 722
sungetc()
    for input buffers 827, 839
swap() 136
    for shared_ptr 93, 97
    for arrays 263, 265
    for container adapters 649
    for containers 250, 255, 258, 407
    for deques 287
    for forward lists 303
    for lists 293
    for maps and multimaps 336
    for packaged tasks 977
    for pairs 61
    for promises 977
    for sets and multiset 321
    for strings 674, 702
    for tuples 71
    for unique_ptr 111
    for unordered containers 368
    for vectors 271, 274
swapping
    iterator values 446
    values 136
swapping elements 566
swap_ranges()
    algorithm 566
symbol
    monetary pattern 878
sync()
    for output buffers 839
synchronization of threads 982
sync_with_stdio()
    for streams 845
system
    error category 50
    system_category() 50
system_clock 149
    duration 149
    from_time_t() 158
    is_steady 149
    now() 149, 152
    period 149
    rep 149
    time_point 149
    to_time_t() 152, 158
system_error 41, 43, 45
    code() 48
    error conditions 45
    <system_error> 44

T

table()
    for ctype facets 895
table_size
    for ctype facets 895
tags
    for iterators 466
tan()
    for complex 935, 940
    global function 941
tanh()
    for complex 935, 940
    global function 941
tellg()
    for streams 799
tellp()
    for streams 799
template
    >> 13
    alias 27, 1024
constructor 36
copy constructor 62
default parameter 33
function 27
member templates 34
nested class 37
nontype parameters 33
typedef 27, 1024
typedef 27

variation 26, 68
zero initialization 37
tera ratio unit 142
terminate() 162
text_file_busy 47
this_thread 981
g et_id() 981
sleep_for() 947, 981
sleep_until() 981
yield() 955, 981
thousands_sep() for moneypunct facets 874
for numpunct facets 870
thread 964, 979
constructor 979
destructor 979
detach() 979
detached 967
get_id() 967, 979
hardware_concurrency() 980
ID 967, 979
join() 979
joinable() 979
native_handle() 979
<thread> 967, 981

threads 945, 979
see concurrency, thread
synchronization 982
tie()
for pairs 67
for streams 819
for tuples 70, 71, 72
time
conversion to/from time_point 158
time() 158
time locale category 884
time_base 889
dateorder 889
dmy 889
mdy 889
no_order 889
ymd 889
ydm 889
timed_mutex 994, 998
timed_out 47
time_get_facet 887
date_order() 888
g et() 888
g et_date() 888
g et_monthname() 888
g et_weekday() 888
g et_year() 888
<time.h> 157
timeout future status 954
time_point 143, 152
+,- 155
+=, -= 155
==, != 155
<, <=, >, >= 155
constructor 155
conversion to/from calendar time 158
current_time 152
epoch 152
for clocks 149
max() 155
min() 155
time_point_cast 155
time_since_epoch() 155
time_point_cast
for timepoints 155
time_put_facet 884
put() 884
timer 160, 947, 981
for locks 994
time_since_epoch()
for timepoints 155
time_t 158, 886
tinyness_before
for numeric limits 118
tm structure 158, 886
to_bytes() for wstring_convert<> 901
to_char_type()
  for char_traits  854

to_int_type()
  for char_traits  854

token iterator
  for regular expressions  727

tolower()  684, 896
  for ctype facets  891

too_many_files_open  47

too_many_files_open_in_system  47

too_many_links  47

too_many_symbolic_link_levels  47

top()
  for container adapters  648

to_string()  652
  for strings  682, 713

to_time_t()  757
  for clocks  152, 158

tr1  7
  namespace  39

traits
  for characters  689, 853
  for iterators  466
  for types  122

  see type traits

traits_type
  for strings  693

transaction safety  248

transform()
  algorithm  225, 240, 563, 564, 684
  for collate facets  904

  versus for_each()  509

transitive  315

traps
  for numeric limits  118

true_name()
  for numpunct facets  870

true_type  125, 142

trunc stream flag  796

try_lock
  for unique_locks  1000

try_lock()
  for unique_locks  1000

  for mutexes  994, 998

  multiple locks  995

  spurious failures  994

try_lock_for()  994
  for unique_locks  1000

  for mutexes  160, 998

try_lock_until()  994
  for unique_locks  1000

  for mutexes  160, 998

try_to_lock  996

tuple  806

  and arrays  268

  and initializer lists  72

tuple  68

  =  71

  ==, !=  71

  <, <=, >, >=  71

  and pair  75

  constructor  69, 71

  destructor  71

  get()  74

  ignore  72

  I/O  74

  make_tuple()  69, 70, 71

  output  74

  swap()  71

  tie()  70, 71, 72

  tuple_cat()  73

  tuple_element  73

  tuple_size  73

<tuple>  66, 68

tuple_cat()
  for tuples  73

tuple_element
  for pairs  62

  for tuples  73

  tuple interface

  for arrays  268

tuple_size

  for pairs  62

  for tuples  73
u16string 655, 664
    see string
u32string 655, 664
    see string
UCS-2 UCS-4 851
uflow()
    for input buffers 839
unary_function 497
unary predicate 226
underflow()
    for input buffers 839
underflow_error 41, 43
underlying_type trait 131
unexpected() 42
unget()
    for input streams 770
uniform initialization 15
uniform_int_distribution 908, 917, 921, 947
uniform_real_distribution 908, 917, 921
uninitialized_copy() 1028
uninitialized_copy_n() 1028
uninitialized_fill() 1028
uninitialized_fill_n() 1028
union set 616
unique()
    algorithm 578
    for shared_ptrs 94
    for forward lists 310, 312
    for lists 297, 298, 421
unique_copy()
    algorithm 580
unique_future 975
unique_lock 996, 1000
    for condition variables 1004
unique pointer
    see unique_ptr
unique_ptr 98, 822
    * 111
    -> 111
    = 102, 111
    ==, != 111
    <, <=, >, >= 111
    [] 111
    and arrays 105
    as member 103
    bool() 100, 111
    comparisons 112
    constructor 111
    deleter 107
    destructor 111
    get() 111
    get_deleter() 111
    initialization 100
    ownership transfer 101
    performance 114
    release() 100, 111
    reset() 111
    swap() 111
unlock()
    for unique_locks 1000
    for mutexes 989, 998
unordered collection 167
unordered container 167, 180
see container
begin() for buckets 374
bucket() 374
bucket_count() 374
bucket interface 374, 380, 429
bucket_size() 374
end() for buckets 374
equivalence criterion 377
hash function 377
modifying access 221
order of duplicates 183
terminology 168
user-defined inserter 471
unordered_map 183, 185, 355
see unordered container
= 368
==, != 367
[] 186, 374, 408
as associative array 185, 374
at() 186, 374, 408
begin() 369, 429
begin() for buckets 374
bucket() 429
bucket_count() 362, 380, 429
bucket interface 374
bucket_size() 429
cbegin() 369, 429
cend() 369, 430
clear() 370, 371
compare function 366
cst_local_iterator 399
constructor 360
count() 368
crbegin() 369
crend() 369
destructor 360
element access with bind() 494
emplace() 370, 371
emplace_hint() 371
empty() 367
end() 369, 430
end() for buckets 374
equal_range() 368
equivalence criterion 357, 366, 377
erase() 370, 371
exception handling 375
find() 368, 373
hasher 399
hash function 363, 377
hash_function() 362, 427
header file 356
insert() 370, 371, 372, 382
iterators 368
key_eq() 362, 427
key_equal 399
lambda as equivalence criterion 379
lambda as hash function 379
load_factor() 362, 380, 427
local_iterator 399
max_bucket_count() 362, 429
max_load_factor() 362, 380, 383, 427, 429
max_size() 367
modifying access 221
piecewise construction 373
policy 359
rbegin() 369
rehash() 362, 428
removing elements 342
rend() 369
reserve() 362
size() 367, 380
swap() 368
value_type 356
<unordered_map> 356
unordered_multimap 355, 383
see unordered container
= 368
==, != 367
[] 186, 374, 408
begin() 369, 429
begin() for buckets 374
bucket() 429
bucket_count() 362, 380, 429
bucket interface 374
bucket_size() 429
cbegin() 369, 429
cend() 369, 430
clear() 370, 371
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Line Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>compare function</td>
<td>366</td>
</tr>
<tr>
<td>const_local_iterator</td>
<td>399</td>
</tr>
<tr>
<td>constructor</td>
<td>360, 383</td>
</tr>
<tr>
<td>count()</td>
<td>368</td>
</tr>
<tr>
<td>crbegin()</td>
<td>369</td>
</tr>
<tr>
<td>crend()</td>
<td>369</td>
</tr>
<tr>
<td>destructor</td>
<td>360</td>
</tr>
<tr>
<td>element access with bind()</td>
<td>494</td>
</tr>
<tr>
<td>emplace()</td>
<td>370, 371</td>
</tr>
<tr>
<td>emplace_hint()</td>
<td>371</td>
</tr>
<tr>
<td>empty()</td>
<td>367</td>
</tr>
<tr>
<td>end() for buckets</td>
<td>374</td>
</tr>
<tr>
<td>equal_range()</td>
<td>368</td>
</tr>
<tr>
<td>equivalence criterion</td>
<td>357, 366, 377</td>
</tr>
<tr>
<td>erase()</td>
<td>370, 371</td>
</tr>
<tr>
<td>exception handling</td>
<td>375</td>
</tr>
<tr>
<td>find()</td>
<td>368, 373</td>
</tr>
<tr>
<td>hasher</td>
<td>399</td>
</tr>
<tr>
<td>hash function</td>
<td>363, 377</td>
</tr>
<tr>
<td>hash_function()</td>
<td>362, 427</td>
</tr>
<tr>
<td>header file</td>
<td>356</td>
</tr>
<tr>
<td>insert()</td>
<td>370, 371, 372, 382</td>
</tr>
<tr>
<td>iterators</td>
<td>368, 383</td>
</tr>
<tr>
<td>key_eq()</td>
<td>362, 427</td>
</tr>
<tr>
<td>key_equal</td>
<td>399</td>
</tr>
<tr>
<td>lambda as equivalence criterion</td>
<td>379</td>
</tr>
<tr>
<td>lambda as hash function</td>
<td>379</td>
</tr>
<tr>
<td>load_factor()</td>
<td>362, 380, 427</td>
</tr>
<tr>
<td>local_iterator</td>
<td>399</td>
</tr>
<tr>
<td>max_bucket_count()</td>
<td>362, 380, 383, 427, 429</td>
</tr>
<tr>
<td>max_load_factor()</td>
<td>362, 380, 383, 427, 429</td>
</tr>
<tr>
<td>max_size()</td>
<td>367</td>
</tr>
<tr>
<td>modifying access</td>
<td>221</td>
</tr>
<tr>
<td>order of duplicates</td>
<td>183</td>
</tr>
<tr>
<td>piecewise construction</td>
<td>373</td>
</tr>
<tr>
<td>policy</td>
<td>359</td>
</tr>
<tr>
<td>rbegiin()</td>
<td>369</td>
</tr>
<tr>
<td>rehash()</td>
<td>362, 428</td>
</tr>
<tr>
<td>removing elements</td>
<td>342</td>
</tr>
<tr>
<td>rend()</td>
<td>369</td>
</tr>
<tr>
<td>reserve()</td>
<td>362</td>
</tr>
<tr>
<td>size()</td>
<td>367, 380</td>
</tr>
<tr>
<td>stable order</td>
<td>183</td>
</tr>
<tr>
<td>swap()</td>
<td>368</td>
</tr>
<tr>
<td>value_type</td>
<td>356</td>
</tr>
<tr>
<td>unordered_multiset</td>
<td>182, 196, 355, 377</td>
</tr>
<tr>
<td>see unordered container</td>
<td>= 368</td>
</tr>
<tr>
<td>==, !=</td>
<td>367</td>
</tr>
<tr>
<td>begin()</td>
<td>369, 429</td>
</tr>
<tr>
<td>begin() for buckets</td>
<td>374</td>
</tr>
<tr>
<td>bucket()</td>
<td>429</td>
</tr>
<tr>
<td>bucket_count()</td>
<td>362, 380, 429</td>
</tr>
<tr>
<td>bucket interface</td>
<td>374</td>
</tr>
<tr>
<td>bucket_size()</td>
<td>429</td>
</tr>
<tr>
<td>cbegin()</td>
<td>369, 429</td>
</tr>
<tr>
<td>cend()</td>
<td>369, 429</td>
</tr>
<tr>
<td>clear()</td>
<td>370, 371</td>
</tr>
<tr>
<td>compare function</td>
<td>366</td>
</tr>
<tr>
<td>const_local_iterator</td>
<td>399</td>
</tr>
<tr>
<td>constructor</td>
<td>360, 377</td>
</tr>
<tr>
<td>count()</td>
<td>368</td>
</tr>
<tr>
<td>crbegin()</td>
<td>369</td>
</tr>
<tr>
<td>crend()</td>
<td>369</td>
</tr>
<tr>
<td>destructor</td>
<td>360</td>
</tr>
<tr>
<td>emplace()</td>
<td>370, 371</td>
</tr>
<tr>
<td>emplace_hint()</td>
<td>371</td>
</tr>
<tr>
<td>empty()</td>
<td>367</td>
</tr>
<tr>
<td>end() for buckets</td>
<td>374</td>
</tr>
<tr>
<td>equal_range()</td>
<td>368</td>
</tr>
<tr>
<td>equivalence criterion</td>
<td>357, 366, 377</td>
</tr>
<tr>
<td>erase()</td>
<td>370, 371, 377</td>
</tr>
<tr>
<td>exception handling</td>
<td>375</td>
</tr>
<tr>
<td>find()</td>
<td>368, 373, 377</td>
</tr>
<tr>
<td>hasher</td>
<td>399</td>
</tr>
<tr>
<td>hash function</td>
<td>363, 377</td>
</tr>
<tr>
<td>hash_function()</td>
<td>362, 427</td>
</tr>
<tr>
<td>header file</td>
<td>356</td>
</tr>
<tr>
<td>insert()</td>
<td>370, 371, 372, 377, 382</td>
</tr>
<tr>
<td>iterators</td>
<td>368, 377</td>
</tr>
<tr>
<td>key_eq()</td>
<td>362, 427</td>
</tr>
<tr>
<td>key_equal</td>
<td>399</td>
</tr>
<tr>
<td>lambda as equivalence criterion</td>
<td>379</td>
</tr>
<tr>
<td>lambda as hash function</td>
<td>379</td>
</tr>
<tr>
<td>load_factor()</td>
<td>362, 380, 427</td>
</tr>
<tr>
<td>local_iterator</td>
<td>399</td>
</tr>
<tr>
<td>max_bucket_count()</td>
<td>362, 429</td>
</tr>
</tbody>
</table>
unordered_multiset (continued)
    max_load_factor() 362, 380, 383, 427, 429
    max_size() 367
    modifying access 221
    order of duplicates 183
    policy 359
    rbegin() 369
    rehash() 362, 428
    rend() 369
    reserve() 362
    size() 367, 380
    stable order 183
    swap() 368
    value_type 356

unordered_set 198, 355, 375
    see unordered container
    = 368
    ==, != 367
    begin() 369, 429
    begin() for buckets 374
    bucket() 429
    bucket_count() 362, 380, 429
    bucket interface 374
    bucket_size() 429
    cbegin() 369, 429
    cend() 369, 430
    clear() 370, 371
    compare function 366
    const_local_iterator 399
    constructor 360, 375
    count() 368
    crbegin() 369
    crend() 369
    destructor 360
   emplace() 370, 371
   emplace_hint() 371
    empty() 367
    end() 369, 430
    end() for buckets 374
    equal_range() 368
    equivalence criterion 357, 366, 377
    erase() 370, 371, 375
    exception handling 375
    find() 368, 373, 375
    hasher 399
    hash function 363, 377
    hash_function() 362, 427
    header file 356
    insert() 370, 371, 372, 375, 382
    iterators 368, 375
    key_eq() 362, 427
    key_equal 399
    lambda as equivalence criterion 379
    lambda as hash function 379
    load_factor() 362, 380, 427
    local_iterator 399
    max_bucket_count() 362, 429
    max_load_factor() 362, 380, 383, 427, 429
    max_size() 367
    modifying access 221
    policy 359
    rbegin() 369
    rehash() 362, 428
    rend() 369
    reserve() 362
    size() 367, 380
    swap() 368
    value_type 356

unordered_set> 356
    unsetf() 688
        for streams 779
    unshift()  
        for codecvt facets 898
    unsynchronized data access 984
    upper
        for ctype_base 894
    upper_bound()
        algorithm 611
        for containers 405
        for maps and multimaps 335
        for sets and multisets 319
    uppercase manipulator 784
    uppercase stream flag 784
    uppercase string characters 684
    US-ASCII 851
    use_count()
        for shared_ptrs 94, 97
        for weak_ptrs 89
Index V

use_facet() 864, 867
user-defined
  <<, >> 810
  algorithm 308, 468
  allocator 1024
  container 385
  exception 635
  function object 495
  inserter 471
  iterator 471
  manipulators 777
  sorting criterion 228, 476
  stream buffers 832
uses_allocator trait 128
using declaration 40
using directive 40
UTF-8 UTF-16 UTF-32 851
  reading and writing 901, 903
utilities 59
<utility> 20, 60, 136, 138

V

valarray 943
valarray 943
valid() 943
  for futures 975
  for packaged tasks 977
valid range 203, 205
value
  for integral_constant 125
  monetary pattern 878
value()
  for exceptions 49
value_comp()
  for associative containers 427
  for maps and multimaps 335
  for sets and multisets 318
value_compare
  for associative containers 399
value initialization 15
value pair 60
value semantics
  for containers 245
value_too_large 47
value_type
  for allocators 1026
  for complex 935
  for container adapters 645
  for containers 260, 397
  for insert() 341, 372
  for integral_constant 125
  for iterator_traits 467
  for maps 345
  for maps and multimaps 331
  for strings 693
  for unordered containers 356
variadic template 26, 68
vector 169, 270, 279
  see container
++ , -- for iterators 440
= 274
==, != 273
<, <=, >, >= 273
[] 274
  as C-style array 278
assign() 274
at() 274
back() 274
begin() 276
capacity() 270, 273, 427
cbegin() 276
cend() 276
clear() 277
constructor 272, 273, 1027
contiguity of elements 278
crbegin() 276
crend() 276
data() 278
destructor 272, 273
element access 274
emplace() 277
eemplace_back() 277
empty() 273
end() 276
erase() 277
exception handling 278
for bool 281
front() 274
header file 270
vector (continued)
insert() 277
iterators 275
max_size() 273
pop_back() 277
push_back() 277
rbegin() 276
reallocation 276
removing elements 276
rend() 276
reserve() 271, 273, 1028
resize() 271
shrink capacity 271
shrink_to_fit() 271, 273
size() 270, 273
swap() 271, 274
<vector> 270
vector<bool> 281
and concurrency 985
const_reference 282
flip() 281, 282
reference 282
versions of C++ 7
void*
I/O 756
volatile
and concurrency 988, 998

W
wait()
for condition variables 1004, 1009
for futures 953, 975
wait_for() 160
for condition variables 1009
for futures 953, 975
wait_until() 160
for condition variables 1009
for futures 953, 975
wcerr 751
wchar_t 852, 858
input 755
numeric limits 116
wcin 751
wclog 751
wcout 751
wcregex_iterator 726
wcregex_token_iterator 727
weak pointer
see weak_ptr
weak_ptr 84, 96
bad_weak_ptr 89
lock() 88
use_count() 89
weibull_distribution 917, 922
wfilebuf 791
wfstream 791
what()
for exceptions 45, 52
whitespace
compressing 582
wide-character format 850
widen()
for ctype facets 891
for streams 790
width()
for streams 782, 811
wifstream 791
wios 750
wiostream 751
wistream 751
wistringstream 802
wofstream 791
wostream 751
wostringstream 802
wregex 719
write()
for output streams 771
global function 835, 837
writing
see output
wrong_protocol_type 47
ws manipulator 746, 774
wregex_iterator 726
wregex_token_iterator 727
wstreambuf 750, 832
see input buffer, output buffer
wstreampos 799
wstring 655, 664
   see string
wstringbuf 802
wstring_convert<> 901
wstringstream 802

X

xalloc()
   for streams 815
xdigit
   for ctype_base 894
xsgetn()
   for input buffers 840
xsputn()
   for output buffers 832

Y

ymd date order 889
yield()
   for this_thread 955, 981
yota ratio unit 142
yotta ratio unit 142

Z

zepto ratio unit 142
zero()
   for durations 147
zero initialization 37
zetta ratio unit 142